NTNU
Det skapende universitet

Adaptive Locking Protocols for Mobile Databases

Steinar Skjerven

Master i datateknikk
Oppgaven levert: Juni 2006
Hovedveileder: Mads Nygård, IDI
Medveileder(e): Hien Nam Le, IDI

Norges teknisk-naturvitenskapelige universitet
Institutt for datateknikk og informasjonsvitenskap

Oppgavetekst

Disconnections in communication may cause data inconsistency and delay database operations. Mobile resource constraints also contribute to the problem. Many locking protocols have been proposed to deal with this challenge, for example, speculative, altruistic, and online-offline locking protocols. However, there are still major limitations. This project will investigate adaptive locking protocols to support mobile databases to cope with different circumstances of mobile environments.

Oppgaven gitt: 16. januar 2006
Hovedveileder: Mads Nygård, IDI

Innhold

1 Sammendrag

Denne diplomoppgaven har undersøkt hvordan en adaptiv låseprotokoll kan hjelpe mobile databaser til å håndtere forskjellige problemer som eksisterer i det mobile miljøet. Diplomoppgaven har også undersøkt hvordan andre låseprotokoller løser problemer i det mobile miljøet. En av disse andre låseprotokollene er låseprotokollen online-offline, som den adaptive låseprotokollen har tatt utgangspunkt i. Den adaptive låseprotokollen har også hentet ideer fra de andre låseprotokollene. Design og implementering av online-offline er inkludert i diplomoppgaven. Designet og implementering av online-offline er utvidet til en adaptiv låseprotokoll.

2 Forord

Denne rapporten er resultatet av den avsluttende diplomoppgaven i sivilingeniørstudiet i datateknikk ved Norges teknisk-naturvitenskapelige universitet (NTNU). Diplomoppgaven er tatt hos databasegruppen på institutt for databaseteknikk og informasjonsvitenskap (IDI). Diplomoppgaven er skrevet av Steinar Skjerven, som er student ved IDI på NTNU. Diplomoppgaven er skrevet mellom 16. januar og 12. juni 2006.

Jeg vil takke min hovedveileder Mads Nygård for god veiledning gjennom hele diplomoppgaven. I tillegg vil jeg takke min biveileder Hien Nam Le for å ha gjort resultatet av hans PhD-these tilgjengelig og for nyttige tips.

Steinar Skjerven
12.06.2006

3 Innledning

Denne diplomoppgaven vil undersøke adaptive låseprotokoller som kan støtte mobile databaser til å håndtere forskjellige problemer som eksisterer i det mobile miljøet.

3.1 Formål

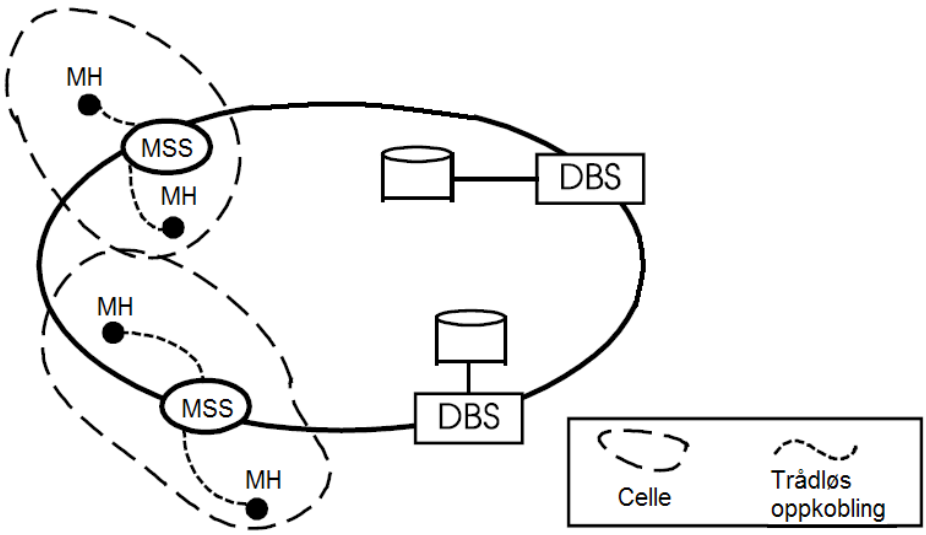
I denne diplomoppgaven skal adaptive låseprotokoller for mobile databaser undersøkes. Mobile databaser har ulike restriksjoner som ikke eksisterer i andre databasesystemer. I litteraturen er foreslått ulike låseprotokoller som håndterer noen av disse restriksjonene. Online-offline låseprotokoll er en av disse. Diplomoppgaven skal inneholde et forstudie som identifiserer restriksjonene til mobile databaser og undersøke de eksisterende låseprotokollene. Videre skal oppgaven utrede hvordan eksisterende adaptive låseprotokoller løser problemer i mobile databaser. Oppgaven skal også beskrive en adaptiv låseprotokoll og hvordan denne kan løse problemer i mobile databaser. Diplomoppgaven inkluderer også implementering og testing av låseprotokollen online-offline og utvidelse av denne til en adaptiv låseprotokoll.

4 Forstudie

Forstudiet gir en innledning av mobile databaser og ulike eksisterende låseprotokoller. Innledningen av mobile databaser inkluderer karakteristikker, utfordringer og løsninger for mobile databaser.

4.1 Mobile databaser

Uviklingen i trådløs nettverksteknologi og bærbare datamaskiner har ført til utviklingen av mobile databaser[5]. Mobile databaser består av mobile enheter (MH), mobile støttestasjoner (MSS) og databasetjenere (DBS). Arkitekturen er vist i figur 1. Mobile enheter er koblet til databasesystemet



Figur 1: Arkitektur mobile databaser

gjennom de mobile støttestasjonene via en trådløs oppkobling. Hver MSS tilbyr muligheter for trådløs oppkobling for mobile enheter innenfor et begrenset geografisk område. Dette området blir referert til som en celle. Alle cellene utgjør det totale området som mobile enheter fritt kan bevege seg innenfor. Når en mobil enhet beveger seg inn i en ny celle må ansvaret overføres fra MSS som har ansvaret for den gamle cellen til MSS i den nye cellen.

Mobile enheter kan variere fra tynne til fulle klienter. Når den mobile enheten er en tynn klient er tjeneren ansvarlig for all prosessering og kommunikasjon.

Den mobile enhetens eneste oppgave er da å vise utdata og sørge for nødvendig inndata til tjeneren. Når den mobile enheten er en full klient har nok ressurser til å emulere databasetjenerfunksjoner. Den kan da operere på egen hånd. Mobile enheter kan også dynamisk forandres mellom tynn og full klient.

4.2 Karakteristikker for mobile databaser

Mobile databaser har karakteristikker som skiller de fra tradisjonelle databaser. Disse karakteristikkene er:

- I motsetning til tradisjonelle databasesystemer er brukere i et mobile databaser ikke knyttet til en fast geografisk lokasjon. I stedet flytter punktet de er knyttet til nettverket seg etter hvordan brukeren beveger seg.
- Mobile databaser inneholder mobile enheter. Disse enhetene har begrenset batteritid og begrensede ressurser. Med begrensede ressurser menes begrenset minne, lagringsmuligheter, prosesseringskapasitet og skjermstørrelse.
- De mobile enhetene er koblet til databasetjeneren via en trådløs oppkobling. Dette gjør at de mobile enhetene ikke til en hver tid har kontakt med databasetjeneren. Dette kan skje ofte og i lange perioder. Båndbredden til oppkoblingen kan også variere. Mobile enheter opererer derfor i ulike modi. Disse modiene er: tilkoblet, delvis tilkoblet og frakoblet. I tillegg kan mobile enheter gå inn i en sovemodus, hvor ingen beregninger utføres. Frakoblinger kan både skje planlagt og være uforutsigbare. Med uforutsigbar frakobling menes en frakobling som ikke var planlagt.
- I mobile databaser finner man mobile transaksjoner. I [9] defineres en mobil transaksjon til å være en transaksjon hvor minst én mobil enhet er involvert. Mobile transaksjoner kan involvere både de mobile enhetene og databasesystemet. Mobile enheter kan både være enkle klienter og fremstå som selvstendige tjenere når de er i frakoblet modus. Hvis de mobile enhetene skal fremstå som tjenere må de inneha lagringsmuligheter for kopier at databaseelementer, muligheter for å utføre databaseoperasjoner på databaseelementene og andre databaseegenskaper. [9] definerer fem utføringsmodeller for mobile transaksjoner:

1. Mobile transaksjoner blir initiert av mobile enheter, men all utførelse skjer på databasesystemet.
2. Mobile transaksjoner blir initiert og utført på den mobile enheten.
3. Distribuert utførelse mellom én mobil enhet og databasesystemet.
4. Distribuert utførelse mellom flere mobile enheter.
5. Distribuert utførelse mellom flere mobile enheter og flere databasetjenere.

I utførelsesmodell 1 sendes query til databasetjeneren, som utfører forespørselen i queryet, og sender resultatet tilbake til den mobile enheten. Denne utførelsesmodellen er godt egnet lokasjonsavhengige query. Et lokasjonsavhengig query kan være å få informasjon om hotell innenfor en radius på fem kilometer. I utførelsesmodell 2 må de mobile enhetene ha databaseegenskaper. All utførelse av transaksjoner skjer på de mobile enhetene. De må være så selvstendige at de kan fortsette å arbeide, selv om de er frakoblet. Utførelsesmodell 3 er veldig fleksibel. Distribusjon av transaksjoner er ønskelig for eksempel når tilgjengeligheten av ressurser er lav på den mobile enheten. Målet til utførelsesmodell 4 er å tilby en peer-to-peer-tilnærming. I stedet for å få hjelp av databasetjeneren, kan en mobil enhet spørre en annen mobil enhet som ligger nærmere enn databasetjenere. Med nærmere menes mindre kommunikasjonskostnader. Den siste utførelsesmodellen er den fullstendig distribuerte varianten, der transaksjoner kan være distribuert mellom flere databasetjenere og mobile enheter. De to siste utførelsesmodellene er ikke aktuelle for dagens mobile databaser, da de ennå ikke er utviklet.

4.3 Utfordringer i mobile databaser

Karakteristikkene til mobile systemer gir nye utfordringer og problemer. Ulike utfordringer og problemer er:

1. Mobile enheter har mindre ressurser enn stasjonære enheter. I tillegg mobile enheter batteridrevet. Hvor få ressurser de mobile enhetene innehar påvirker designet av mobile systemer. For eksempel krever utførelse av transaksjoner på de mobile enhetene et minimum av ressurser.
2. Hvordan man skal håndtere lav og variabel båndbredde og de forskjellige oppkoblingsmodi er en utfordring. De mobile enhetene kan være

frakoblet i lange perioder og frakoblingene kan være frekvente. Dette kan føre til lav effektivitet i systemet, hvis frakobla transaksjoner får blokkere data. Frakoblinger kan også være uforutsigbare og komme uten at de er planlagt.

3. Mobile enheter er mobile. Hvordan man skal håndtere denne mobiliteten er en utfordring. Utfordringen ligger i skifte av celle, lokasjonsavhengige query og hvem som skal holde på data om lokasjonen til de mobile enhetene. Lokasjonsavhengige query er query som bruker data ut fra lokasjonen til den mobile enheten. Hovedproblemet til lokasjonsdata er å vite nåværende posisjon til en mobil enhet, i tillegg hvem skal være ansvarlig for denne informasjonen.

4. Mobile enheter kan frakobles databasesystemet. I den frakoblede perioden kan den mobile enheten fortsette å arbeide på lokale kopier av databasen. Flere mobile enheter kan ha kopi av samme databaseelement i sitt cache. En utfordring er hvordan den mobile enheten skal hente disse kopiene til cache og hvordan man skal sikre konsistens når flere mobile enheter arbeider med kopier av samme databaseelement. Arbeid i frakoblet modus krever også at det eksisterer mekanismer for å integrere arbeidet som ble utført i frakoblet modus, når den mobile enheten blir tilkoblet.

5. Mobile transaksjoner kan være langvarige. Hvordan man skal håndtere langvarige transaksjoner er en utfordring.

4.4 Løsninger for mobile databaser

Dette avsnittet presenterer noen løsninger for utfordringene i mobile databaser. Hvert punkt svarer til punkt i avsnitt 4.3. Løsningene er:

1. En strategi for å spare energi og håndtere lave båndbredder er å sende data til de mobile enhetene uten en forutgående forespørsel. Denne strategien kalles push-based. Den mostsatte strategien kalles pull-based, der data leveres når dataen spørres etter. Videre kan man la databasetjeneren utføre deler eller alt arbeid, hvis ressursene er begrenset.

2. En frakoblingsprotokoll kan designes som forbereder mobile enheter og systemet for en planlagt frakobling. Protokollen bør sikre at den mobile enheten har nok data til å kunne operere på egen hånd og den

bør informere alle interessenter om frakoblingen. Modusen delvis oppkoblet bør også ha en egen protokoll som forbereder systemet på lavere båndbredde. I tillegg bør det eksistere en protokoll som håndterer overgangen fra frakoblet modus til normal operasjonsmodus. Nødvendige forhåndsregler mot uforutsigbare frakoblinger bør også eksistere.

3. En overleveringsprotokoll[1] brukes for å håndtere prosessen rundt skifte av celle. Denne prosessen skal være transparent for den mobile enheten. Informasjon og ansvar for den mobile enheten overføres fra den gamle til den nye MSS.

4. Replikasjon av data brukes i mobile systemer. En av grunnene for dette er å øke selvstendigheten til de mobile enhetene. Når de har kopier av data i lokalt cache kan de arbeide uten at det kreves en oppkobling. Dette øker effektiviteten i det mobile systemet. Protokollene for replikasjon bør være tilpasset karakteristikkene til mobile systemer. For eksempel forskjellige grader av konsistens på kopier av data i lokal cache. Replikasjon muliggjør også nedlasting av data før det er behov for den, noe som øker effektiviteten i mobile systemer.

5. Låseprotokoller brukes for å håndtere problemer i mobile databaser. Låseprotokollene kan designes slik at langvarige transaksjoner ikke får blokkere databaseelementer når dette ikke er nødvendig. En annen hensikt til låseprotokoller er å sikre konsistensen i databasen når flere mobile enheter har kopi av samme databaseelement i lokalt cache.

4.5 Låseprotokoller

Målet for diplomen er en adaptiv låseprotokoll. En adaptiv låseprotokoll er en låseprotokoll som skal håndtere planlagte og uplanlagte frakoblinger, støtte caching av data slik at mobile enheter kan arbeide i frakoblet modus, støtte kooperative transaksjoner og støtte konfliktbevissthet for transaksjoner. I dette avsnittet presenteres ulike låseprotokoller som en adaptiv låseprotokoll kan baseres på. Disse ble valgt ut fra oppgaveteksten og etter diskusjon med biveileder.

[1] Overleveringsprotokoll = hand-off protocol.

4.5.1 Online-offline låseprotokoll

I dette avsnittet presenteres online-offlineprotokollen(OOL) som er beskrevet i [3].

Innledning
Låseprotokollens hovedmål er å sørge for høyere effektivitet i mobile systemer. Låseprotokollen støtter også kooperative operasjoner og øker bevisstheten i det mobile systemet. Et dataobjekt som er låst av en frakoblet mobil enhet kan minske effektiviteten i systemet og blokkere operasjoner til andre mobile enheter. For å oppnå målene introduseres flere typer låser og mekanismer for å alternere mellom låsetypene.

Det mobile systemet som brukes til å illustrere låseprotokollen består av en sentral database og mobile enheter. De mobile enhetene har muligheter til å cache små mengder data på lokal cache. Databasen inneholder arbeidsoppgaver som kalles mobile oppgaver og som lastes ned til det lokale cache på de mobile enhetene. Databaseoperasjoner blir initiert og utført av de mobile enhetene. Disse operasjonene kan utføres online eller offline.

Låsetyper
Låseprotokollen introduserer to typer låser: onlinelåser og offlinelåser. Det er totalt fire låsetyper:

- Write-intended-offlinelås (wioff):
  En transaksjon som holder en wiofflås har leserettigheter i frakoblet modus. Transaksjonen som holder låsen har også bevissthet om at andre transaksjoner kan endre databaseelementet, mens den er i frakoblet modus. Alle andre låser kan bli bevilget på en ressurs som har en wiofflås. Dette gjør systemet mer effektivt, siden en wiofflås ikke kan blokkere databaseelementet når transaksjonen er frakoblet i en lang periode.

- Write-offlinelås (woff):
  En transaksjon som ønsker å modifisere et databaseelement i frakoblet modus spør etter en wofflås. Transaksjonen har lese- og skriverettigheter til databaseelementet og bare denne transaksjonen kan oppdatere databaseelementet. Hvis databaseelementet blir modifisert, må låsen oppgraderes til en wonlås før resultatet kan integreres i databasen. En wofflås kan bevilges selv om det er ronlåser på dataelementet. Dette

kan gjøres siden databaseelementet ikke oppdateres før transaksjonen blir tilkoblet igjen.

- Read-onlinelås (ron):
  En ronlås er en vanlig leselås.

- Write-onlinelås (won):
  En wonlåsen er en vanlig skrivelås.

Låsematrise og låseregler
Kompatibilitetsmatrisen for alle låsene er vist i tabell 1.

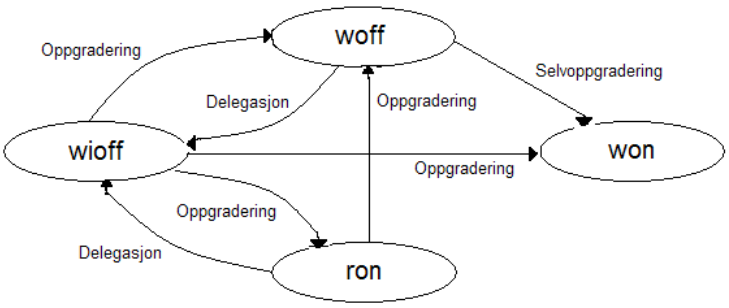
	$X_{(i,wioff)}$	$X_{(i,ron)}$	$X_{(i,woff)}$	$X_{(i,won)}$
$X_{(j,wioff)}$	Y	Y	N	N
$X_{(j,ron)}$	Y	Y	N	N
$X_{(j,woff)}$	Y	Y	N	N
$X_{(j,won)}$	Y	N	N	N

Tabell 1: Kompatibilitetsmatrise for online-offlinelåseprotokollen

Denne kompatibilitetsmatrisen og alle andre kompatibilitetsmatriser i diplomoppgaven leses slik at låsene til venstre er de forespurte, mens låsene på toppen er eksisterende låser på databaseelementet. Y indikerer at låsene er kompatible og den forespurte låsen blir bevilget. N indikerer at låsene ikke er kompatible og den forespurte låsen blir avvist. $X_{(j,won)}$ representerer en wonlåseforspørsel fra den mobile enheten M_j på databaseelement X. $X_{(i,wioff)}$ representerer at den mobile enheten M_i holder en wiofflås på databaseelement X. Fra kompatibilitetsmatrisen kan man se at wiofflåser ikke er i konflikt med andre wiofflåser eller ronlåser. Dette gjør at mobile enheter kan være oppmerksomme på modifikasjoner av databaseelementer. I tillegg blir alle låseforespørsler bevilget, hvis databaseelementet er wiofflåst. Dette øker effektiviteten til det mobile systemet, siden wiofflåser ikke blokkerer andre låseforespørsler. Man kan også se at wofflåser kan bli bevilget når databaseelementet er ronlåst. Dette kan gjøres siden verdien ikke umiddelbart blir oppdatert i databasen.

Andre mekanismer
Låseprotokollen inneholder alternerende låseoperasjoner. Med disse alternerende låseoperasjoner kan en mobil enhet ta over eller frigi låser. Det er to

typer alternerende låseoperasjoner. Den ene er oppgradering og den andre er delegasjon. Figur 2 viser de alternerende låseoperasjonene. Det er to situa-



Figur 2: Alternerende låseoperasjoner

sjoner hvor en mobil enhet oppgraderer låser. Disse to situasjonene er:

- når en mobil enhet ønsker å ta over en lås som en annen mobil enhet holder.
- når den mobile enheten ønsker å oppgradere sin egen lås.

Det er tre situasjoner hvor en mobil enhet delegerer låser. Disse tre situasjonene er:

- når den mobile enheten er ferdig med en operasjon og ønsker å avgi låsen.
- når mobile enheter gjør ronlåser om til wiofflåser før den mobile enheten frakobles.
- når den mobile enheten holder en wofflås og verdien på elementet ikke har forandret seg. I denne situasjonen blir wofflåsen delegert tilbake til en wiofflås.

Diskusjon

Databasetjeneren håndterer alle låseforespørsler og kan holde følge med til hvilken tid og i hvilken rekkefølge de er gjort. På denne måten kan databasetjeneren sikre at alle mobile enheter som forespør etter wiofflåser og ronlåser leser konsistent data. Når en mobil enhet, M_i, oppretter oppkoblingen etter en frakobling vil databasetjeneren sjekke om det er noen operasjoner i konflikt. Alle operasjoner som skjedde før M_i fikk bevilget sin wofflås eller wonlås

er korrekte. Alle operasjoner som skjedde fra M_i fikk sin wofflås eller wonlås til databaseelementet er oppdatert i databasen kan være korrekt avhengig av riktighetskriteriumet til det mobile systemet. Hvis riktighetskriteriumet sier at operasjonene er i konflikt må korrigerende mekanismer iverksettes. Alle operasjoner som henter etter at databaseelementet er oppdatert i databasen er i konflikt og korrigerende mekanismer må iverksettes.

4.5.2 Altruistisk låseprotokoll

I dette avsnittet presenteres en altruistisk låseprotokoll(AL) som er beskrevet i [8].

Innledning
I mobile databaser eksisterer det langvarige transaksjoner (LLT). En av grunnene til at det eksisterer LLT er frakoblinger. LLT holder databaseelementer i relativt lange perioder. Altruistisk låsing er en utvidelse av tofaselåsing. Altruistisk låsing gjør at LLT kan avgi låser på databaseelementer når det er klart at databaseelementene ikke skal aksesseres mer. Andre transaksjoner trenger da ikke vente så lenge på låser som de måtte gjort i tofaselåsing. Låseprotokollen garanterer serialiserbarhet og gir ingen restriksjoner på hvilken rekkefølge databaseelementene må aksesseres.

Låsetyper
Altruistisk låsing bruker de to tradisjonelle låsetypene:

- Skrivelås.
- Leselås.

Skrivelåser og leselåser på databaseelementer må først skaffes med en låseoperasjon. Når transaksjonen er ferdig med databaseelementet gir den fra seg låsen med en opplåsingsoperasjon. Låseoperasjonen og opplåsningsoperasjonen kalles parallellitetskontrolloperasjoner. Låseprotokollen innfører ikke noen nye låsetyper, men den innfører derimot en ny parallellitetskontrolloperasjon, som kalles donasjonsoperasjon. Donasjonsoperasjonen brukes til å informere tidsplanleggeren[2] om at transaksjonen som holder en lås på et databaseelement er ferdig med databaseelementet. Transaksjonen vil ikke

[2]Tidsplanlegger = scheduler.

aksessere dette databaseelementet mer. Donasjonsoperasjonen er lik opplåsingsoperasjonen, men skiller seg fra den ved at transaksjonen fortsatt kan låse flere databaseelementer. Donasjonsoperasjonen er en frivillig operasjon. Det kreves ikke at den brukes, selv om transaksjoner er ferdig med å aksessere et databaseelement. De tre parallellitetskontrolloperasjonene som brukes i altruistisk låsing er:

- Låseoperasjon.
- Opplåsningsoperasjon.
- Donasjonsoperasjon.

Låsematrise og låseregler
Kompatibilitetsmatrisen for skrivelåser og leselåser er vist i tabell 2.

	Leselås	**Skrivelås**
Leselås	Y	N
Skrivelås	N	N

Tabell 2: Kompatibilitetsmatrise for altruistisk låsing

Altruistisk låsing innførte ingen nye låsetyper, men derimot en ny parallellitetskontrolloperasjon. Det er tre regler rundt bruken av donasjonsoperasjonen:

1. Transaksjoner kan bare donere databaseelementer som de allerede har en lås på.
2. Transaksjoner kan ikke aksessere databaseelementer de har donert.
3. Transaksjoner må låse opp databaseelementet uansett om de har brukt donasjonsoperasjonen eller ikke.

I tillegg til disse tre reglene settes det også restriksjoner for transaksjoner som aksesserer donerte databaseelementer. Grunnen til dette er at det kan oppstå historier som ikke er serialiserbare. Hvis man ser for seg en transaksjon, T_1, som aksesserer et databaseelement, A, og senere et databaseelement, B. Når T_1 har aksessert A brukes donasjonsoperasjonen. En annen transaksjon, T_2, aksesserer det donerte databaseelementet og ønsker også å aksessere B. Hvis T_2 aksesserer B før T_1 har aksessert og oppdatert B vil historien ikke være serialiserbar. Restriksjonene er definert i to regler. Den første er:

- Altruistisk låseregel 1
  To transaksjoner kan ikke samtidig holde låser i konflikt på et databaseelement, hvis ikke den ene transaksjonen har donert databaseelementet først.

Hvis en transaksjon låser et databaseelement som en annen transaksjon har donert, men ennå ikke låst opp sier man at transaksjonen er i kjølvannet av[3] av den donerende transaksjonen. Både databaseelementet og transaksjonen kan beskrives som å være i kjølvannet av den donerende transaksjonen. En transaksjon er fullstendig i kjølvannet av en transaksjon, hvis alle databaseelementer den låser er i kjølvannet av den andre transaksjonen. En transaksjon, T_2, står i gjeld til[4] en transaksjon, T_1, hvis og bare hvis låsene til de to transaksjonene er i konflikt eller en lås fra en tredje transaksjon er i konflikt med begge. Den andre regelen er:

- Altruistisk låseregel 2
  Hvis en transaksjon, T_a, står i gjeld til en annen transaksjon, T_b, må den være fullstendig i kjølvannet til T_b helt til T_b ufører sin første opplåsingsoperasjon.

Andre mekanismer
Transaksjoner kan informere tidsplanleggeren om hvilke databaseelementer den muligens kommer til å aksessere. Hvorfor dette er ønskelig diskuteres i neste avsnitt. En merkeoperasjon benyttes for å gi nødvendig informasjon til tidsplanleggeren. Flere transaksjoner kan holde merkeoperasjonen samtidig på et databaseelement. Det kreves ikke at transaksjoner benytter merkeoperasjonen, men de som gjør det kalles avmerkingstransaksjoner[5]. Det er to regler rundt bruken av merkeoperasjonen:

1. En avmerkingstransaksjon kan ikke låse et databaseelement uten å først ha merket det.

2. Når en transaksjon først har donert et databaseelement, kan den ikke merke flere databaseelementer. Donering og merking foregår altså i to faser.

[3] I kjølvannet av = in the wake.
[4] Stå i gjeld til = indebted to.
[5] Avmerkingstransaksjoner = marking transactions.

Diskusjon
Det er to mulige utvidelser av den altruistiske låseprotokollen. Begge går ut på å utvide transaksjonenes kjølvann. Den første utvidelsen er å la transaksjoner kunne donere databaseelementer uten at den holder en lås på databaseelementet. Dette kan gjøres selv om andre transaksjoner holder en lås på eller har donert databaseelementet. En slik donasjon refereres til som en utvidet donasjon. En transaksjon som utfører en utvidet donasjon må også utføre en opplåsingsoperasjon. Dette gjøres for å fortelle andre at databaseelementet ikke lenger er i kjølvannet til transaksjonen. En transaksjon må være sikker på at den ikke må aksessere databaseelementet den donerer. Bruken av en utvidet donasjon gjør det mer sannsynlig at en transaksjon som kjører i kjølvannet av en transaksjon ikke vil bli blokkert. Men det blir også mer sannsynlig at transaksjoner som ikke kjører i kjølvannet vil bli blokkert. Dette gjør at utvidede donasjoner ikke er fornuftig i alle situasjoner. For å bestemme om det er fornuftig må transaksjonenes aksesseringsmønster studeres. Den andre utvidelsen går ut på at det er tidsplanleggeren som automatisk utfører utvidede donasjoner på vegne av transaksjoner. Dette er aktuelt når transaksjoner som kjører i kjølvannet av en transaksjon ønsker å låse et databaseelement som ikke er i kjølvannet. Men når man tillater dette kan det skapes uønskede kjølvann. Hvis transaksjonen må aksessere akkurat dette databaseelementet på et senere tidspunkt, må transaksjonen aborteres. Grunnen er at transaksjoner ikke kan låse databaseelementer som de allerede har donert. For å overkomme dette problemet brukes merkeoperasjonen. Tidsplanleggeren ekspanderer ikke transaksjonens kjølvann med disse databaseelementene som er merket og eliminerer dermed muligheten for fremtidige konflikter. Tidsplanleggeren kan nå donere et databaseelement, a på vegne av en transaksjon, T, hvis den følger disse tre reglene:

1. T er en avmerkingstransaksjon.
2. T har utført minst én donasjonoperasjon.
3. T har ikke merket a.

Hvis en transaksjon donerer et databaseelement den har skrevet til, kan andre transaksjoner se dette skrevne databaseelementet og committe. Hvis transaksjonen som donerer det skrevne databaseelementet aborterer, oppstår det et problem. For å unngå dette problemet kan man vente med å donere skrevne databaseelementer til transaksjonen har committet. Leste databaseelementer kan doneres til enhver tid. Dette kalles steng altruistisk låsing. Streng altruistisk låsing er ikke effektiv når det eksisterer mange lange transaksjoner som

inneholder mange skriveoperasjoner, men hvis den de lange transaksjonene består av mange leseoperasjoner kan streng altruistisk låsing være ideell. Et alternativ til streng altruistisk låsing er å forsinke commit til transaksjoner som har sett et skrevet donert databaseelement, til den donerende transaksjoner committer. Selv om denne varianten fører til mer parallellitet, vil den også føre til lange forsinkelser ved commit og innføre kaskaderende aborteringer[6].

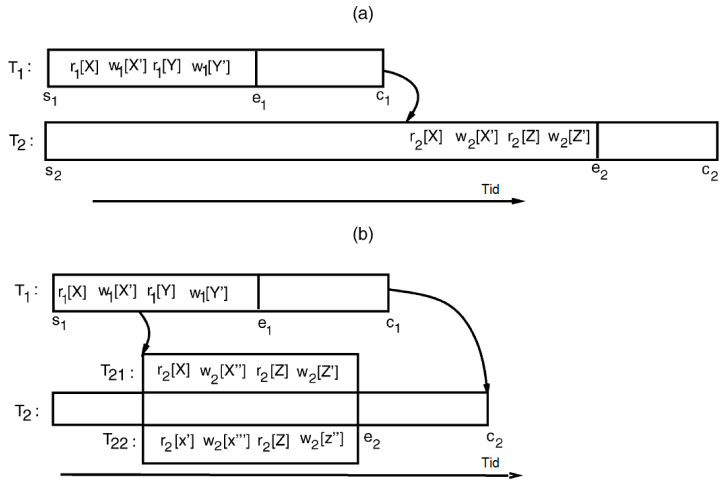
4.5.3 Spekulativ låseprotokoll

I dette avsnittet presenteres en spekulativ låseprotokoll(SL) som er beskrevet i [7].

Innledning
SL utvider tofaselåsing for å forbedre effektiviteten av transaksjonsprosessering i distribuerte databasesystemer. I SL produserer transaksjoner korresponderende etterkopier[7] av databaseelementer under utførelsen av transaksjonen. I tofaselåsing slippes låsene på databaseelementer først etter tofasecommitprotokollen er utført, selv om det produseres etterkopier. SL øker parallelliteten ved å la transaksjoner slippe låsen på et databaseelement når det har produsert en korresponderende etterkopi. Ventende transaksjoner kan da aksessere denne etterkopien. I SL aksesserer ventende transaksjoner både førkopi og etterkopi av databaseelementer og utfører spekulative utførelser. Én utførelse blir valgt basert på om de foregående transaksjonene committet eller aborterte. Utførelse av flere utførelser øker parallelliteten uten at det går ut over serialiserbarhetskravet. I den naive varianten av SL eksploderer antallet spekulative utførelser med antallet konflikter. Det presenteres flere varianter av SL som reduserer antallet spekulative utførelser som utnytter at sjansen for at transaksjoner committer er større enn at de aborterer.

Figur 3a viser transaksjonsprosessering med tradisjonell tofaselåsing for distribuerte databaser. For en transaksjon, T_i, noterer s_i, e_i og c_i starten på utførelsen, slutten på utførelsen og slutten på commitprosesseringen. Figur 3b viser transaksjonsprosesseringen i SL. Når transaksjon T_1 produserer etterkopien X', aksesserer T_2 både X og X' og starter de spekulative utførelsene T_{21} og T_{22}. T_2 committer først etter commitprosesseringen til $T1$ er ferdig. Hvis T_1 committer beholdes T_{22} og hvis T_1 aborterer beholdes T_{21}.

[6]Kaskaderende abortering = cascading abort.
[7]Etterkopi = after-image.



Figur 3: Tradisjonell tofaselåsing og SL

Låsetyper
I spekulativ låsing er det tre typer låser:

- Leselås
  En vanlig leselås.
- Utførelsesskrivelås
  Låsen transaksjoner spør etter når de ønsker lese- og skriverettigheter til et databaseelement.
- Spekulasjonsskrivelås
  Utførelsesskrivelåsen blir forandret til en spekulasjonsskrivelås etter at etterkopien er inkludert i respektivt dataobjekttre.

Låseprotokollen spesifiserer at det ikke foregår noen konversjon fra leselåser til utførelsesskrivelåser. Et dataobjekttre i denne låseprotokollen er en struktur for å holde på versjoner av databaseelementer, som er produsert av spekulative transaksjoner. I dette treet er den committede versjonen av databaseelementet roten, og resten av nodene er versjoner som ikke er committet.

Låsematrise og låseregler
Tabell 3 viser kompatibilitetsmatrisen til SL.

Flere transaksjoner kan samtidig holde leselåser og spekulasjonsskrivelåser på et databaseelement. Dette er ikke mulig for utførelsesskrivelåser. SL sikrer

	Lese	**Utførelsesskrive**	**Spekulasjonsskrive**
Lese	Y	N	SY
Utførelsesskrive	SY	N	SY

Tabell 3: Kompatibilitetsmatrise for SL

konsistensen ved å forme commitavhengigheter mellom transaksjoner. Hvis T_i former en commitavhengighet med T_j, vil T_i committe etter termineringen av T_j. SY, som står for spekulativ ja, indikerer at transaksjonen som forespør en lås utfører spekulative utførelser og danner en commitavhengighet med transaksjonene som holder leselåsene og utførelsesskrivelåsene på databaseelementet. Det er to commitavhengighetsregler:

1. Hvis T_i får en utførelsesskrivelås på et databaseelement som T_j holder enten en leselås eller spekulasjonsskrivelås på, vil T_i forme en commitavhengighet med T_j.
2. Hvis T_i får en leselås på et databaseelement som T_j holder en spekulasjonsskrivelås på, vil T_i forme en commitavhengighet med T_j.

Andre mekanismer
Ingen andre mekanismer anses som viktige for denne låseprotokollen.

Diskusjon
Det eksisterer flere varianter av SL. De fire variantene som diskuteres er: SL(n), SL(0), SL(1) og SL(2). Situasjonen som disse variantene analyseres under består av en transaksjon, T_i, som er i konflikt med n transaksjoner. SL(n) er den ekstreme pessimistiske varianten av SL og utfører alle spekulative utførelser. Dette gjør T_i robust mot n transaksjonsaborteringer. SL(0) er den ekstreme optimistiske varianten av SL. I denne varianten utfører en transaksjon bare en utførelse ved å lese bare etterkopien av de påfølgende transaksjonene. T_i er i SL(0) robust mot null transaksjonsaborteringer. T_i må altså abortere, hvis bare en av de påfølgende transaksjonene må abortere. SL(1) og SL(2) benytter seg av informasjonen om at transaksjoner har større sannsynlighet for å committe enn abortere. I SL(1) er transaksjoner robust mot én transaksjonsabortering. For å få til dette er det nok å støtte $n + 1$ utførelser for en transaksjon. I SL(2) er transaksjoner robust mot to transaksjonsaborteringer. For å få til dette er det nok å støtte $\Sigma n + 1$ utførelser for en transaksjon.

Spekulative utførelser krever muligens også ekstra ressurser på de mobile enhetene.

4.5.4 Prewrites låseprotokoll

I dette avsnittet presenteres en låseprotokoll(PL) som er beskrevet i [4].

Innledning
Prewrites hovedmål er å øke tilgjengeligheten til data. Dette gjøres ved å introdusere en preskriveoperasjon før en skriveoperasjon. En preskriveoperasjon oppdaterer ikke databaseelementet, men gjør databaseelementets fremtidige verdi synlig frem til endelig commit. Verdien som blir gjort synlig garanteres å committe. Når en transaksjon har lest alle verdier og deklarert alle preskriveverdier kan transaksjonen precommitte på den mobile enheten. Ansvaret for resten av utførelsen av transaksjonen, som å oppdatere databaseelementene, blir nå flyttet over til en MSS. Det introduseres også en preleseoperasjon som returnerer en preskriveverdi i motsetning til en leseoperasjon som returner en skriveverdi. Transaksjoner blir serialisert basert på preskriverekkefølgen.

Låsetyper
Prewrite bruker en låseprotokoll som inneholder fire låsetyper:

- Preleselås:
  Brukes for preleseoperasjoner.
- Preskrivelås:
  Brukes for preskriveoperasjoner.
- Leselås:
  Er en vanlig leselås.
- Skrivelås:
  Er en vanlig skrivelås.

Låseprotokollen bygger på tofaselåsing. Tofaselåsing sier at en transaksjon ikke kan få en lås bevilget etter at transaksjonen har avgitt en lås. Tofaselåsing alene er ikke nok for å garantere korrekt utførelse av transaksjoner i Prewrite. I tillegg introduseres det et krav som sier at transaksjoner ikke kan

få låser bevilget etter at en lås er oppgradert til en annen låsetype. Preskrivelåser blir oppgradert til skrivelåser før databaseelementet i databasen skal oppdateres.

Låsematrise og låseregler
Kompatibilitetsmatrisen for alle låsene er vist i tabell 4.

	Prelese	**Lese**	**Preskrive**	**Skrive**
Prelese	Y	Y	N	N
Lese	Y	Y	Y	N
Preskrive	N	Y	N	Y
Skrive	Y	N	Y	N

Tabell 4: Kompatibilitetsmatrise for prewrites låseprotokoll

Andre mekanismer
Ingen andre mekanismer anses som viktige for denne låseprotokollen.

Diskusjon
Det vil ikke eksistere vranglåser[8] mellom transaksjoner som er precommittet. Grunnen til dette er at preskrivelåser og skrivelåser blir anskaffet på et ordnet vis. Transaksjoner som er precommittet vil derfor ikke aborteres grunnet en vranglås og er garantert å committe.

4.5.5 Browselåsprotokoll

I dette avsnittet presenteres en låseprotokoll(BL) som er beskrevet i [1]. I artikkelen presenteres et rammeverk for å håndtere frakoblinger i et mobilt system. Rammeverket presenterer flere modi for å sjekke ut databaseelementer. For en av modusene presenteres en låseprotokoll som inneholder browselåser.

Innledning
Det mobile systemet som presenteres er en fullstendig distribuert og replisert database som er bygd opp av bare mobile enheter. En mobil enhet kan kobles

[8]Vranglås = deadlock.

fra de andre mobile enhetene. Dette kan gjøres i fire ulike modi: avkobling, utsjekking, relaksert utsjekking og optimistisk utsjekking[9]. Tofaselåsing blir brukt til parallellitetskontroll lokalt. En metode for global synkronisering blir også brukt, men artikkelen spesifiserer ikke hvilken. Låseprotokollen med browselåser presenteres for modusen: relaksert utsjekking. Låseprotokollens mål er å øke tilgjengeligheten av data.

Låsetyper
Det er tre typer låser i denne låseprotokollen.

- Leselås.
  En vanlig leselås.
- Skrivelås.
  En vanlig skrivelås.
- Browselås
  En browselås tillater en mobil enhet som holder låsen å lese skitten data. Den tillater ikke skriving og garanterer ikke konsistent lesing.

Låsematrise og låseregler
Kompatibilitetsmatrisen er vist i tabell 5.

	Leselås	**Skrivelås**	**Browselås**
Leselås	Y	N	Y
Skrivelås	N	N	Y
Browselås	Y	Y	Y

Tabell 5: Kompatibilitetsmatrise for browselåser

For relaksert utsjekking partisjonerer den mobile enheten som skal kobles fra databasen ved å sjekke ut databaseelementer. Databaseelementer kan sjekkes ut av bare én mobil enhet om gangen. Den mobile enheten har lese- og skriverettigheter til partisjonen den sjekker ut. Andre mobile enheter som er tilkoblet har muligheter til å browse databaseelementene i denne partisjonen. Når en transaksjon browser et utsjekket databaseelement kan den lese skitten data og har da samtidig gitt opp serialiserbarhet. Tre regler må bli fulgt:

[9] Avkobling = sign-off og utsjekking = check-out.

1. Bare de databaseelementene som er låst av en pseudotransaksjon når den mobile enheten frakobles, kan bli modifisert av transaksjoner i den frakobla mobile enheten.

2. Databaseelementene som er låst av en pseudotransaksjon kan bli browset, men ikke modifisert av andre mobile enheter.

3. Databaseelementer som ikke er låst av en pseudotransaksjon når den mobile enheten frakobles kan leses av transaksjoner på den frakobla mobile enheten.

En pseudotransaksjon er en transaksjon som holder skrivelåser på databaseelementer som en mobil enhet ønsker å modifisere når den er frakoblet. Skrivelåsene som en pseudotransaksjon holder er en ordinær skrivelås, bortsett fra at pseudotransaksjonen som holder disse skrivelåsene ikke kan aborteres på grunn av en vranglås. Hvis en pseudotransaksjon ikke klarer å skaffe skrivelåser til alle ønskede databaseelementer, frakobles den mobile enheten med oppdateringsrettigheter til subsettet som den har skrivelåser til.

Andre mekanismer
Ingen andre mekanismer anses som viktige for denne låseprotokollen.

Diskusjon
ANSI/ISO SQL-92-spesifikasjonen definerer fire isolasjonsnivåer for transaksjoner: READ UNCOMMITTED, READ COMMITTED, REPEATABLE READ og SERIALIZABLE. Browselåsprotokollen svarer til nivå to, altså READ COMMITTED, hvis browselåser bare blir benyttet på skrivelåser som blir holdt av pseudotransaksjoner. Dette er fordi en browse tillater lesing av gamle dataverdier og ikkegjentakbar lesing hvis databaseelementet blir browset etter at den frakobla mobile enheten blir tilkoblet. Verdien som leses vil likevel ikke være ucommittet og en virkelig skitten verdi. Hvis browselåser tillates på databaseelementer som er lås av vanlige transaksjoner og pseudotransaksjoner, vil isolasjonsnivået være nivå én.

Problemer med denne låseprotokollen er at den ikke støtter mobilitet og uplanlagte frakoblinger.

4.6 Sammendrag av låseprotokollene

Tabell 6 gir en oppsummering av viktige aspekter ved låseprotokollene.

	Mål	**Låsetyper**	**Annet**	**Problemer**
OOL	Konfliktbevissthet, kooperative transaksjoner og effektivitet	Ron, won, woff, wioff	Alternerende op.	-
AL	Effektivitet	Lese- og skrivelås	Donasjon og regler	Kaskaderende aborteringer
SL	Effektivitet	Lese, spekulasjonsskrive og utførelseskrive	Flere varianter av SL og spekulative utførelser	Ressurskrav
PL	Tilgjengelighet	Lese, skrive, prelese og preskrive	-	-
BL	Tilgjengelighet	Lese, skrive og browse	Pseudotransaksjoner	Ikke støtte for mobilitet og uplanlagte frakoblinger

Tabell 6: Sammendrag av låseprotokoller

5 En adaptiv låseprotokoll

I dette kapitteles vil en adaptiv låseprotokoll undersøkes og presenteres.

5.1 Formålet til en adaptiv låseprotokoll

Frakoblinger kan skape inkonsistens og forsinke databaseoperasjoner. En adaptiv låseprotokoll vil forsøke å løse disse utfordringene, i tillegg til andre utfordringer ved mobile databaser. En av utfordringene som ble identifisert i avsnitt 4.3 var hvordan mobile enheter skal cache kopier av databaseelementer og hvordan de skal sikre konsistens når flere mobile enheter arbeider på kopier av samme databaseelement. Det er to tilnærminger til caching av databaseelementer. Den pessimistiske tillater bare én mobil enhet å ha kopi av et databaseelement til enhver tid. Dette fører til lav tilgjengelighet og effektivitet. Den optimistiske tilnærmingen tillater alle mobile enheter til å ha en kopi av alle databaseelementer. Konflikter blir korrigert når den mobile enheten tilkobles. Dette fører til høy tilgjengelighet, men øker frekvensen av konflikter. Korrigering av konflikter er kostbart i form av prosesseringskapasitet og båndbredde. En løsning for å sikre konsistens i databasen er å bruke en låsemekanisme, som beskrevet i avsnitt 4.4. Den adaptive låseprotokollen skal sikre konsistens når alle mobile enheter kan ha kopi av et databaseelement til en hver tid. For å støtte konsistens til transaksjoner som arbeider i frakoblet modus, må disse transaksjonene være klar over potensielle konflikter. Dette kalles konfliktbevissthet [10].

Målene til den adaptive låseprotokollen er:

- å støtte planlagte og uplanlagte frakoblinger.
- å støtte caching av data slik at mobile enheter kan arbeide i frakoblet modus.
- å støtte konfliktbevissthet for å sikre konsistens i databasen.
- å støtte kooperative transaksjoner.
- å sikre høy effektivitet og tilgjengelighet.

[10] Konfliktbevissthet = Conflict awareness.

5.2 Diskusjon av låseprotokollene fra forstudiet og utvidelser til en adaptiv låseprotokoll

I avsnitt 4.5 ble ulike låseprotokoller som den adaptive låseprotokollen kan baseres på beskrevet. Disse låseprotokollene var:

- Online-offline
- Altruistisk låseprotokoll
- Spekulativ låseprotokoll
- Prewrite
- Browselåsprotokoll

Under utarbeidelse av formålet til oppgaven ble det sagt at den adaptive låseprotokollen skulle baseres på online-offline, men at flere eksisterende låseprotokoller skulle undersøkes. Grunnen til at andre låseprotokoller skulle undersøkes var for å vurdere om den adaptive låseprotokollen kunne utnytte noen av mekanismene som eksisterer i disse låseprotokollene. Under følger en vurdering av hver av låseprotokollene. Vurderingene er basert på målene til den adaptive låseprotokollen fra avsnitt 5.1.

Online-offline
Online-offline støtter alle målene som ble satt opp for den adaptive låseprotokollen. Av denne grunn vil den adaptive låseprotokollen baseres på online-offline. Den adaptive låseprotokollen vil utvide online-offline ved å utvide støtten for konfliktbevissthet og kooperative transaksjoner.

Altruistisk låseprotokoll
Altruistisk låsing øker effektiviteten og tilgjengeligheten ved å innføre en donasjonsoperasjon. Den støtter ikke frakoblinger, konfliktbevissthet eller kooperative transaksjoner. Den altruistiske låseprotokollen kan derfor ikke brukes til en adaptiv låseprotokoll.

Spekulativ låseprotokoll
Spekulativ låsing øker effektiviteten ved å innføre spekulative utførelser. Den støtter ikke frakoblinger, konfliktbevissthet eller kooperative transaksjoner.

Den spekulative låseprotokollen kan derfor ikke brukes til en adaptiv låseprotokoll.

Prewrite
Prewrite støtter planlagte frakoblinger. Mobile enheter kan arbeide i frakoblet modus etter precommit. Prewrites hovedmål er å øke tilgjengeligheten til data. Prewrite støtter derimot ikke konfliktbevissthet og kooperative transaksjoner. Prewrites låseprotokoll kan derfor ikke brukes til en adaptiv låseprotokoll.

Browselåsprotokoll
Browselåsprotokollen støtter planlagte frakoblinger, caching av data og arbeid i frakoblet modus. Den støtter ikke konfliktbevissthet og kooperative transaksjoner. Browselåsprotokollen kan derfor ikke brukes til en adaptiv låseprotokoll. Men det er noen elementer ved denne protokollen som er interessante for en adaptiv låseprotokoll. Den innfører en ny type lås som kalles browselås. Den tillater tilkobla transaksjoner å lese verdier til databaseelementer som frakobla transaksjoner holder en skrivelås på. For browselåsprotokollen gjøres dette uten at konsistens garanteres. Denne typen lås er interessant for den adaptive låseprotokollen, hvis konsistens kan garanteres. Browselåsprotokollen innfører også en pseudotransaksjon som ligger hos databasetjeneren, selv når den mobile enheten er frakoblet. En slik type transaksjon kan brukes for å støtte konfliktbevissthet i den adaptive låseprotokollen.

Den adaptive låseprotokollen vil baseres på online-offline. Online-offline vil utvides slik at den støtter alle målene til den adaptive låseprotokollen. I tillegg vil den adaptive låseprotokollen bruke ideer fra browselåsprotokollen. Hvordan online-offline utvides og hvilke ideer som brukes fra browselåsprotokollen diskuteres videre i avsnitt 5.4.7.

5.3 Eksisterende adaptive låseprotokoller

Et av målene for diplomoppgaven som ble satt opp i formålet var å undersøke eksisterende adaptive låseprotokoller. Det ble derimot ikke funnet noen artikler som nevner en adaptiv låseprotokoller. Av denne grunn er ingen eksisterende adaptive låseprotokoller beskrevet.

5.4 Beskrivelse av en adaptiv låseprotokoll for mobile databaser

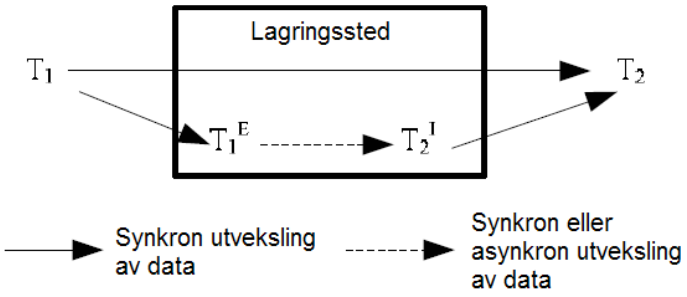
Dette avsnittet skal beskrive en adaptiv låseprotokoll. Den adaptive låseprotokollen er basert på arbeidet til Hien Nam Le fra hans PhD-oppgave [2].

5.4.1 Innledning

Den adaptive låseprotokollen baseres på online-offline og skal utvide denne. Målene til den adaptive låseprotokollen er å støtte planlagte og uplanlagte frakoblinger, konfliktbevissthet, kooperative transaksjoner og at mobile enheter kan arbeide i frakoblet modus. Låseprotokollen skal prøve å øke effektiviteten og tilgjengeligheten av data.

5.4.2 Den adaptive låseprotokollens miljø

Den adaptive låseprotokollen tar utgangspunkt i transaksjonsmodellen MO-WAHS, som er presentert i [6]. Denne modellen tillater mobile enheter å danne dynamiske arbeidsgrupper som kalles mobile affiliasjoner. Mobile enheter i den mobile affiliasjonen kan kommunisere direkte med hverandre, for eksempel med hjelp av Bluetooth. Mobile transaksjoner kan utveksle data gjennom et mobilt lagringssted i den mobile arbeidsgruppen. Dette lagringsstedet kalles eksport-import(EI). Utveksling av data kan skje både synkront og asynkront på et adaptivt vis. Utvekslingen av data er synkron, hvis de mobile transaksjonene er koblet til hverandre gjennom EI. Samtidig som de mobile transaksjonene utveksler data synkront, initieres og utføres mobile eksporteringstransaksjoner og importeringstransaksjoner som backuptransaksjoner. Eksporterings- og importeringstransaksjoner utveksler data på et synkront eller asynkront vis. Hvis de mobile transaksjonene som utveksler data synkront mister forbindelsen på grunn av for eksempel en frakobling, kan utvekslingen av data fortsette gjennom eksporterings- og importeringstransaksjonene. På denne måten er delingsmekanismen adaptiv. Utveksling av data i denne modellen er vist i figur 4 for de mobile transaksjonene T_1 og T_2, eksporteringstransaksjon T_1^E og importeringstransaksjon T_2^I.



Figur 4: Adaptiv utvekslingsmekanisme

5.4.3 Caching av data

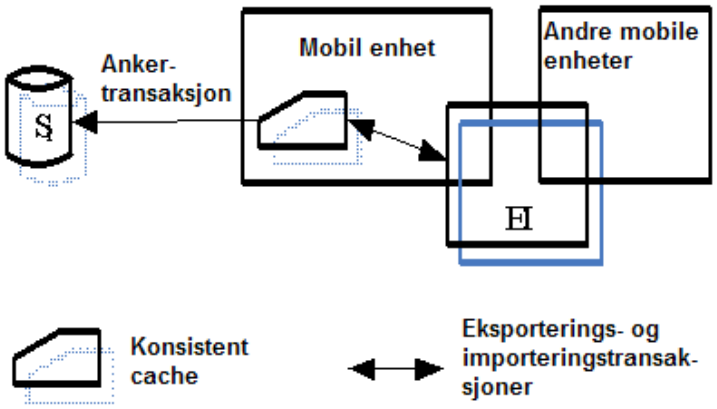
Før en mobil enhet frakobles fra databasetjeneren, caches konsistente delte databaseelementer lokalt på den mobile enheten. Lokale transaksjoner, som kalles offlinetransaksjoner, på den mobile enheten kan utføre databaseoperasjoner på disse databaseelementene i frakoblet modus. De delte databaseelementene kan være cachet med bare leserettigheter eller skriverettigheter. Samtidig kan de samme databaseelementene være cachet hos andre mobile enheter, som fører til potensielle konflikter. Offlinetransaksjoner på mobile enheter må være klar over disse potensielle konfliktene.

5.4.4 Støtte av konfliktbevissthet

For å støtte at offlinetransaksjoner er klar over potensielle konflikter krever låseprotokollen at offlinetransaksjonene sender sine låseforespørsler til databasetjeneren via en ankertransaksjon, T^A. Ankertransaksjonen til en mobil enhet holder alle låsehandlinger til alle konsistente delte databaseelementer som er cachet på den mobile enheten. En ankertransaksjon virker altså som en proxytransaksjon for andre transaksjoner på de andre mobile enhetene. Figur 5 viser en ankertransaksjon i det mobile systemet. S_i er en databasetjener.

5.4.5 Datacachingmodi

For hver mobile enhet MH_i er det en ankertransaksjon T_i^A som virker som en proxy for alle lokale transaksjoner t_i til den mobile enheten. En lokal transak-



Figur 5: Ankertransaksjon i det mobile systemet

sjon kalles en offlinetransaksjon. Den mobile enheten kan gå inn i en datacachingfase når den ønsker å frakoble. Under denne fasen vil ankertransaksjonen sende låseforespørsler slik at den mobile enheten kan cache de nødvendige databaseelementene for prosessering i frakoblet modus. Ankertransaksjonen vil konkurrere med onlinetransaksjoner og andre ankertransaksjoner. Onlinetransaksjoner er transaksjoner som er håndtert av transaksjonsbestyrerene på databasetjenerene. For onlinetransaksjonene gjelder datacachingmodus uten konflikter. Dette er for strengt for offlinetransaksjoner. Derfor er det i tillegg nødvendig med to andre datacachingmodus som tillater konflikter. Datacachingmodi med konflikt tillater transaksjoner å erverve låser med konflikt på delte databaseelementer så lenge de er bevisste på den potensielle konflikten og det er mulig å finne en serialiserbar tidsplan for disse transaksjonene. Det er opp til det mobile databasesystemet å oppdage konflikter. Når en konflikt blir oppdaget vil det bestemme en serialiserbar tidsplan og gi beskjed til de involverte transaksjonene om konflikten. De tre datacachingmodiene er datacachingmodus uten konflikter, datacachingmodus med leseskrivekonflikter og datacachingmodus med skrivelesekonflikter.

Datacachingmodus uten konflikter

For datacachingmodus uten konflikter sørger databasetjenerene for at ingen låseforespørsler med konflikter kan erverves før den mobile enheten frakobles systemet. Den mobile enheten kan likevel ende opp med konflikterende låser på delte databaseelementer, hvis den utfører operasjoner etter utveksling av delt data med andre mobile enheter når den er frakoblet.

Datacachingmodus med leseskrivekonflikter
Datacachingmodus med leseskrivekonflikter tillater en ankertransaksjon å erverve skrivelås på et databaseelement som er leselåst av en annen ankertransaksjon eller en onlinetransaksjon. Grunnen er at offlinetransaksjonen ikke trenger å utføre oppdateringsoperasjonen med det samme. Den kan skje etter at onlinetransaksjonen som holder leselåsen har committet og da er det mulig å legge offlinetransaksjonen etter onlinetransaksjonen i en serialiserbar tidsplan.

Datacachingmodus med skrivelesekonflikter
En onlinetransaksjon eller en annen offlinetransaksjon kan ha behov for å lese et databaseelement som en offlinetransaksjon holder en skrivelås på. Datachachingmodus med skrivelesekonflikter tillater dette så lenge transaksjonene er serialiserbare med transaksjonen som oppdaterer databaseelementet frakoblet.

5.4.6 Deling data i frakoblet modus

Et av målene for den adaptive låseprotokollen var å støtte kooperative transaksjoner. I den adaptive låseprotokollen kan offlinetransaksjoner danne mobile affiliasjoner for å dele data. Dette gjøres gjennom EI, som ble beskrevet i avsnitt 5.1. EI blir dannet når det er behov for å dele data mellom offlinetransaksjoner på forskjellige frakobla mobile enheter. Når en offlinetransaksjon har utført utvekslingen av data, kan den forlate EI. Den siste offlinetransaksjonen som forlater EI, sletter EI. Dataen i EI vil bli integrert i databasen av de deltakende transaksjonene.

Eksporteringstransaksjoner og importeringstransaksjoner er mobile transaksjoner som støtter utveksling av data mellom offlinetransaksjoner i EI. Eksporteringstransaksjoner samhandler med importeringstransaksjoner for å dele data. Definisjonene av disse transaksjonene er:

- Eksporteringstransaksjon
  Rollen til en eksporteringstransaksjon er å støtte offlinetransaksjoner i å
  1. dele delvise eller committede resultater med andre offlinetransaksjoner, så snart som mulig.
  2. å lagre delvise resultater for å hindre tap på grunn av feil.

Offlinetransaksjonen vil initiere én eller flere eksporteringstransaksjoner, avhengig av hvordan den vil dele sin data. Commit av eksporteringstransaksjoner er avhengig av målet til transaksjonen.

- Importeringstransaksjon
  En importeringstransaksjon tillater offlinetransaksjoner å skaffe ønsket informasjon fra andre offlinetransaksjoner. Offlinetransaksjonen initierer én eller flere importeringstransaksjoner for å skaffe denne dataen. Commit av importeringstransaksjonen kan eller kan ikke være avhengig av transaksjonen som initierer den og de assosierende eksporteringstransaksjonene.

5.4.7 Utvidelser av eksisterende låseprotokoller til en adaptiv låseprotokoll for mobile databaser

Dette avsnittet skal beskrive hvordan man kan utvide låseprotokollene som ble beskrevet i avsnitt 4.5 til en adaptiv låseprotokoll.

Utvidelse av låseprotokollen online-offline
Den adaptive låseprotokollen vil ta utgangspunkt i online-offline. De fire låsetypene i online-offline er wonlås, ronlås, wiofflås og wofflås. Wonlåser og ronlåser brukes av onlinetransaksjoner. En wonlås er det samme som en ordinær skrivelås. En ronlås er det samme som en ordinær leselås. Wioff og wofflåser brukes av offlinetransaksjoner. En wiofflås gir offlinetransaksjoner leserettigheter i frakoblet modus og gir samtidig offlinetransaksjoner konfliktbevissthet om potensielle konflikter. En wofflås gir offlinetransaksjoner lese- og skriverettigheter i frakoblet modus. Den adaptive låseprotokollen vil bruke disse låsetypene og reglene for disse. Den adaptive låseprotokollen vil også benytte seg av de alternerende operasjonene som er beskrevet for online-offline.

For datacachingmodus med leseskrivekonflikt tillates det at ankertransaksjoner spør etter skrivelåser på databaseelementer som er leselåst av onlinetransaksjoner eller andre ankertransaksjoner. Offlinetransaksjonen som spør etter skrivelåsen skal oppdatere databaseelementet i frakoblet modus. Dette er det samme som en wofflås i online-offline brukes til. Offlinetransaksjoner som spør etter skrivelåser på et leselåst databaseelement, vil derfor spørre etter en wofflås.

For datacachingmodus med skrivelesekonflikt tillates det at onlinetransaksjoner eller ankertransaksjoner spør etter leselåser på databaseelementer som er

skrivelåst av en ankertransaksjon. Dette er ikke det samme som en wiofflås. En wiofflås fra online-offline bevilges ikke når det eksisterer en skrivelås på databaseelementet. For dette tilfellet må det defineres en ny type lås. Browselåser fra browselåsprotokollen har samme egenskaper som kreves av en slik type lås.

Utvidelse av browselåsprotokollen
For datacachingmodus med skrivelesekonflikt tillates det at onlinetransaksjoner eller ankertransaksjoner spør etter leselåser på databaseelementer som er skrivelåst av en ankertransaksjon. Dette er det samme som browselåser. For browselåser i browselåsprotokollen gis serialiserbarheten opp. Dette skal ikke gjøres i den adaptive låseprotokollen.

Proxytransaksjoner ble introdusert i browselåsprotokollen og ble kalt pseudotransaksjoner der. Ankertransaksjonene i den adaptive låseprotokollen bygger på disse, men skiller seg fra de på følgende måter:

- Settet av låser kan bli alternert når den mobile enheten tilkobler databasen.
- En ankertransaksjon kan bli initiert når den mobile enheten tilkobler databasen. Grunnen til dette er at en mobil enhet ikke trenger å ha cachet noen databaseelementer når den frakobler, men den kan få databaseelementer fra andre mobile enheter i frakoblet modus.
- Ankertransaksjonen lagrer potensielle konflikter mellom databaseelementer i cache til forskjellige mobile enheter.
- Ankertransaksjonen kan støtte mobilitet til transaksjoner.

5.4.8 Låsetyper og regler

Låsetypene i den adaptive låseprotokollen er:

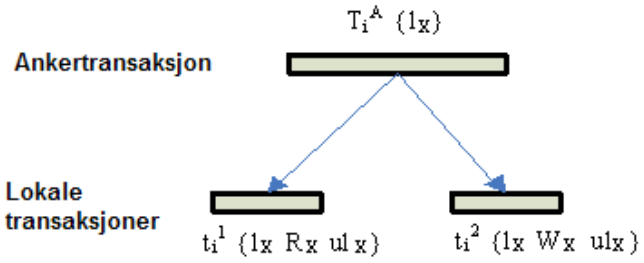
- Ronlås
- Wonlås
- Wiofflås
- Wofflås

- Browselås

En browselås gir onlinetransaksjoner leserettigheter i tilkoblet modus og offlinetransaksjoner leserettigheter i frakoblet modus. Transaksjoner som holder browselåser er bevisste på skrivelesekonflikten som oppstår når de får bevilget en browselås. Browselåser kan bevilges når det eksisterer en wofflås på databaseelementet. Transaksjoner som browser et databaseelement leser den umodifiserte verdien av databaseelementet. Hvis wofflåsen oppgraderes til en wonlås og den modifiserte verdien integreres i databasen, vil transaksjonene som browset databaseelementet lese den nye modifiserte verdien.

Reglene til en ankertransaksjon T_i^A på en mobil enhet MH_i er:

- En ankertransaksjon T_i^A sender låseforespørsler til en databasetjener S_i for å få lese- og skrivelåser på settet av databaseelementer som kreves under den frakobla prosesseringen.
- Hvis disse låseforespørselene blir bevilget, blir de korresponderende databaseelementene cachet lokalt på den mobile enheten. Ankertransaksjonen vil holde lese- og skriveoperasjonene på disse databaseelementene. Figur 6 illustrerer dette.

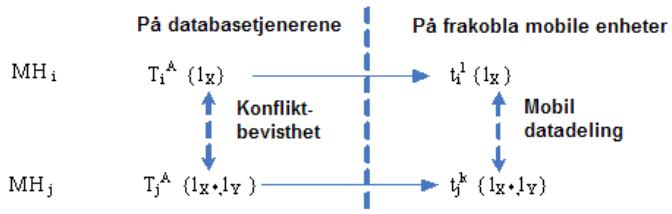


Figur 6: Rollen til en ankertransaksjon

- Når den mobile enheten frakobles, vil ankertransaksjonen fortsatt oppholde seg på databasetjeneren. Kopier av låsenehandlingene vil bli lagret på den mobile enheten som den tar med seg i frakoblet modus. Ankertransaksjonen håndteres av transaksjonsbestyrerene[11] på databasetjeneren og vil ikke bli abortert av noen grunn. Hvis ankertransaksjonen spør etter en lås som er i konflikt vil dette bli lagret på begge ankertransaksjonene og på begge mobile enhetene. Ved å gjøre dette

[11]Transaksjonsbestyrer = transaction manager.

støtter låseprotokollen konfliktbevissthet for offlinetransaksjoner. Figur 7 illustrerer dette. Her lagrer ankertransaksjonene T_i^A og T_j^A den potensielle konflikten på det delte databaseelementet X, for eksempel lese- og skriveoperasjoner mellom de to cachene på de to mobile enhetene MH_i og MH_j.



Figur 7: Ankertransaksjonen støtter konfliktbevissthet

- Når en mobil enhet MH_i tilkobles vil låsehandlingene som holdes av ankertransaksjonen T_i^A bli kombinert med andre låser som er et resultat av deling av data mellom mobile enheter for å bestemme det endelige låsesettet. De lokale transaksjonene t_i^l og t_j^k i figur 7 deler data. Initiale låser kan bli alternert og være forskjellige fra de nærende låsene som holdes av ankertransaksjonene. Etter at ankertransaksjonen T_i^A har bestemt det endelige låsesettet vil resultatet av den lokale transaksjonen t_i^l bli integrert i databasen. Når integreringsprosessen er utført vil T_i^A frigi låsene og comitte.

5.4.9 Kompatibilitetsmatriser

Kompatibilitetsmatrise for datacachingmodus uten konflikt er vist i tabell 7.

	Leselås	**Skrivelås**
Leselås	Y	N
Skrivelås	N	N

Tabell 7: Kompatibilitetsmatrise for datacachingmodus uten konflikter

I denne kompatibilitetsmatrisen er det en onlinetransaksjon eller en ankertransaksjon som forespør etter en lås på et databaseelement som enten andre onlinetransaksjoner eller ankertransaksjoner holder låser på.

	Leselås	**Skrivelås**
Leselås	Y	N
Skrivelås	LS	N

Tabell 8: Kompatibilitetsmatrise for datacachingmodus med leseskrivekonflikter

Kompatibilitetsmatrise for datacachingmodus med leseskrivekonflikt er vist i tabell 8.

I denne kompatibilitetsmatrisen er det en ankertransaksjon som forespør etter en lås på et databaseelement som enten andre onlinetransaksjoner eller ankertransaksjoner holder låser på. LS i kompatibilitetsmatrisen står for at leseskrivekonflikter er tillatt. En leseskrivekonflikt er definert som:

- Leseskrivekonflikt
  Hvis en onlinetransaksjon t_k eller en ankertransaksjon T_i^A holder en leselås på et databaseelement X, og en ankertransaksjon T_j^A forespør etter en skrivelås på databaseelementet, vil databasetjeneren bevilge låsen til T_j^A. En leseskrivekonflikt er angitt som $X(T_{ij}^A, rw)$.

Kompatibilitetsmatrise for datacachingmodus med skrivelesekonflikt er vist i tabell 9.

	Leselås	**Skrivelås**
Leselås	Y	SL
Skrivelås	N	N

Tabell 9: Kompatibilitetsmatrise for datacachingmodus med skrivelesekonflikter

I denne kompatibilitetsmatrisen er det en onlinetransaksjon eller en ankertransaksjon som forespør etter en lås på et databaseelement som en ankertransaksjoner holder låser på. SL i kompatibilitetsmatrisen står for at skrivelesekonflikter er tillatt. En leseskrivekonflikt er definert som:

- Skrivelesekonflikt
  Hvis en ankertransaksjon T_i^A holder en skrivelås på et databaseelement X, og en onlinetransaksjon t_k eller en ankertransaksjon T_j^A forespør etter en leselås på databaseelementet, vil databasetjeneren bevilge låsen

og den umodifiserte verdien av X vil bli returnert. En skrivelesekonflikt er angitt som $X(T_{ij}^{A}, wr)$.

Den adaptive låseprotokollen tillater at databaseelementer caches både med leseskrivekonflikter og skrivelesekonflikter. Ved å kombinere alle kompatibilitetsmatriser for alle datacachingmodi får man kompatibilitetsmatrisene for den adaptive låseprotokollen. Onlinetransaksjoner og offlinetransaksjoner spør etter forskjellige låser på databaseelementene. Onlinetransaksjoner vil bare spørre om ronlåser, wonlåser og browselåser. Kompatibilitetsmatrisene for onlinetransaksjoner i den adaptive låseprotokollen er vist i tabell 10.

	Ron	**Won**	**Wioff**	**Woff**	**Browse**
Ron	Y	N	Y	N	N
Won	N	N	Y	N	N
Browse	N	N	N	Y	Y

Tabell 10: Kompatibilitetsmatrise for onlinetransaksjoner i den adaptive låseprotokollen

Fra kompatibilitetsmatrisen kan man se at onlinetransaksjoner får bevilget browselås på databaseelementer som har en wofflås eller en browselås. Datacachingmodus med skrivelesekonflikt krever at onlinetransaksjoner kan browse databaseelementer som er skrivelåst av en offlinetransaksjon, altså et databaseelement som er wofflåst. Browselåser er også kompatible med andre browselåser, siden de er bare bevilget når det eksisterer en wofflås på databaseelementet. For ron- og wonlåser er det likt online-offline.

Offlinetransaksjoner vil spørre etter wiofflåser, wofflåser og browselåser. Kompatibilitetsmatrisene for offlinetransaksjoner i den adaptive låseprotokollen er vist i tabell 11.

	Ron	**Won**	**Wioff**	**Woff**	**Browse**
Wioff	Y	N	Y	N	N
Woff	Y	N	Y	N	N
Browse	N	N	N	Y	Y

Tabell 11: Kompatibilitetsmatrise for offlinetransaksjoner i den adaptive låseprotokollen

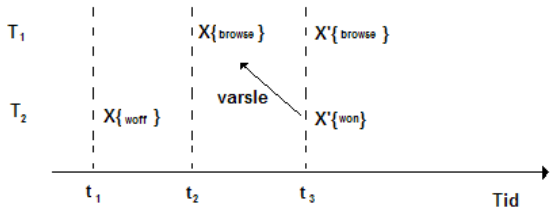
Fra kompatibilitetsmatrisen kan man se at offlinetransaksjoner får bevilget wofflåser på databaseelementer som har ronlåser eller wiofflåser. Dette er kra-

vet til datacachingsmodus med leseskrivekonflikter. Datacachingmodus med skrivelesekonflikt krever at offlinetransaksjoner kan browse databaseelementer som er skrivelåst av en annen offlinetransaksjon, altså et databaseelement som er wofflåst. Browselåser er også kompatible med andre browselåser, siden de er bare bevilget når det eksisterer en wofflås på databaseelementet. For wioff- og wofflåser er det likt online-offline.

5.4.10 Datakonsistens for browselåser

Databasetjeneren må sørge for at transaksjoner som browser databaseelementer leser konsistente verdier. Når en transaksjon får en browselås og browser et databaseelement returneres den umodifiserte verdien. Denne verdien er konsistent helt til transaksjonen som holdt wofflåsen oppgraderer denne til en wonlås og integrerer den modifiserte verdien i databasen. Integrering av verdi skjer ved commit av transaksjonen som holdt wofflåsen. Transaksjoner som har browset dette databaseelementet og ikke har committet kan nå lese den nye verdien. Onlinetransaksjoner som holder browselås får beskjed fra databasetjeneren når verdien modifiseres. Hvis det er en offlinetransaksjon får ankertransaksjonen til offlinetransaksjonen beskjed. Offlinetransaksjoner kan lese den nye verdien når den mobile enheten tilkobles. Disse to situasjonene er vist i figur 8 og 9.

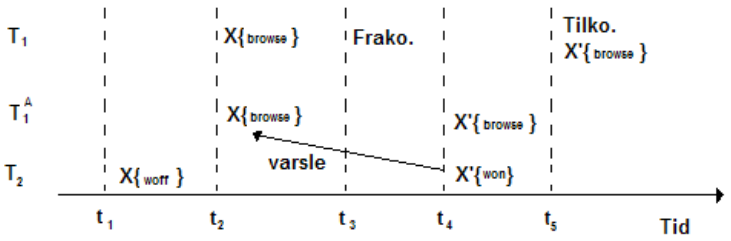
I figur 8 får offlinetransaksjon T_2 bevilget en wofflås på databaseelement X på tidspunkt t_1. På t_2 får onlinetransaksjon T_1 bevilget en browselås og får returnert den umodifiserte verdien av X. Denne verdien er konsistent til t_3, da T_2 tilkobles og integrerer den nye verdien i databasen. Dette blir T_1 varslet om og leser den modifiserte verdien.



Figur 8: Onlinetransaksjon T_1 får browselås på databaseelement X

I figur 9 får offlinetransaksjon T_2 bevilget en wofflås på databaseelement X

på tidspunkt t_1. På t_2 får offlinetransaksjon T_1 bevilget en browselås og får returnert den umodifiserte verdien av X. Denne browselåsen blir også lagret hos ankertransaksjonen T_1^A til T_1. Denne verdien er konsistent til t_4, da T_2 tilkobles og integrerer den nye verdien i databasen. Dette kan ikke T_1 blir varslet om, siden den koblet fra på tidspunkt t_3. Beskjeden om dette gis derfor til ankertransaksjonen. Når T_1 tilkobles vil den få beskjed fra ankertransaksjonen og den kan nå lese den modifiserte verdien.



Figur 9: Offlinetransaksjon T_1 får browselås på databaseelement X

Hvis transaksjonen som oppgraderer sin wofflås til wonlås aborteres etter at wonlåsen er bevilges, vil dette ikke affektere konsistensen til transaksjonene som har browset dette databaseelementet. Grunnen til dette er at disse transaksjonene ikke leser verdien før den er committet. Transaksjonene trenger heller ikke å vente på at transaksjonen committer. Hvis de ønsker å committe selv, committer de med den umodifiserte verdien og serialiseres før transaksjonen som holdt wofflåsen.

5.4.11 Konsistens for deling data i frakoblet modus

Det er to forskjellige måter å dele data mellom mobile transaksjoner i mobile affiliasjoner: dele-for-lesing og dele-for-skriving[12]. For dele-for-lesing er delte databaseelementer bare tilgjengelig med leserettigheter for andre offlinetransaksjoner. Den offlinetransaksjonen som delte databaseelementet holder fortsatt wofflåsen på databaseelementet. For dele-for-skriving skriver transaksjonen den nye dataverdien inn i EI og frigir wofflåsen. Andre transaksjoner får da både lese- og skriverettigheter på databaseelementet.

[12]Dele-for-lesing = Share-to-Read og Dele-for-skriving = Share-to-Write.

Dele-for-lesing
Når transaksjon T_i på den mobile enheten MH_i ønsker å dele sine dataverdier med dele-for-lesing, initierer den eksporteringstransaksjonen T_i^E, som skriver disse verdiene til EI på vegne av transaksjon T_i. Alle transaksjoner T_j som leser slik ny data vil bli serialisert etter T_i og T_i^E. Eksporteringstransaksjon T_i^E former en commitavhengighet til transaksjon T_i.

Dele-for-skriving
Når transaksjon T_i på den mobile enheten MH_i ønsker å dele sine dataverdier med dele-for-skriving, initierer den eksporteringstransaksjonen T_i^E, som skriver de nye dataverdiene til EI og frigir låsen på vegne av T_i. Eksporteringstransaksjonen fungerer som en delegasjonstransaksjon av den originale transaksjonen i dette tilfellet. Andre offlinetransaksjoner kan nå erverve seg lese- og skriverettigheten på dette databaseelementet ved å utføre korresponderende importeringstransaksjoner. Alle transaksjoner T_j som leser eller skriver disse databaseelementene vil bli serialisert etter T_i og T_i^E. Siden T_i ikke lenger holder skriverettigheten på databaseelementet, må enten T_i^E eller transaksjon T_k som oppdaterer databaseelement ta ansvar for å committe verdien i databasen. Transaksjon T_i^E og T_k danner en tvunget-commit-ved-abort-avhengighet [13]. Denne avhengigheten sier at hvis T_k aborteres, må T_i^E committe. Hvis T_k aborterer, må T_i^E integrere den delte dataverdien i databasen.

5.4.12 Andre mekanismer

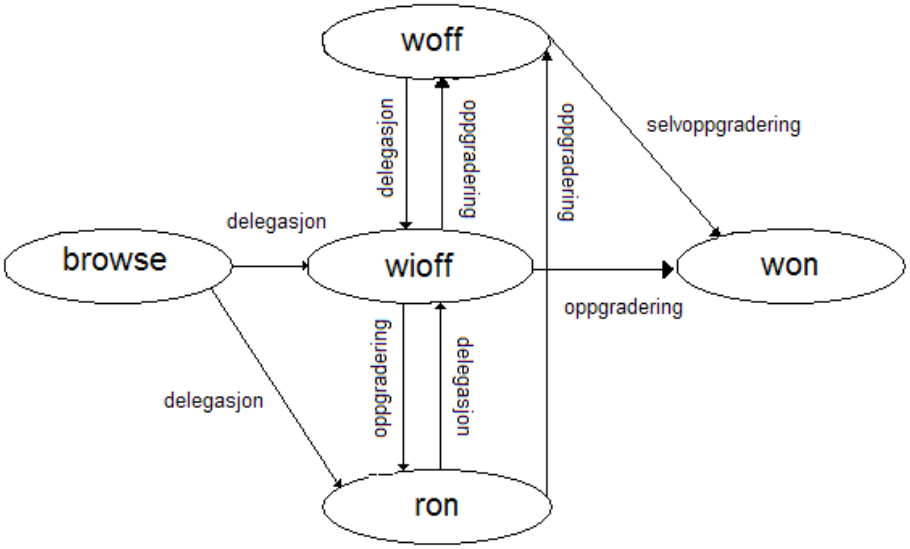
I online-offline finnes det alternerende låseoperasjoner. Alternerende låseoperasjoner er situasjoner, hvor låser oppgraderes eller delegeres. Det finnes også situasjoner hvor browselåser må delegeres. Disse situasjonene er:

- når en transaksjon som holder en wofflås tilkobles og ikke har modifisert verdien på databaseelementet. Etter reglene i online-offline skal da wofflåsen delegeres til en wiofflås. Browselåser som er bevilget på databaseelementet må også delegeres. Browselåser til onlinetransaksjoner delegeres til ronlåser og browselåser til offlinetransaksjoner delegeres til wiofflåser.
- når wofflåsen eller wonlåsen, som wofflåsen er oppgradert til, frigis må browselåsene delegeres. Browselåser til onlinetransaksjoner delegeres til

[13] Tvunget-commit-ved-abort-avhengighet = Forced-Commit-on-Abort Dependency.

ronlåser og browselåser til offlinetransaksjoner delegeres til wiofflåser.

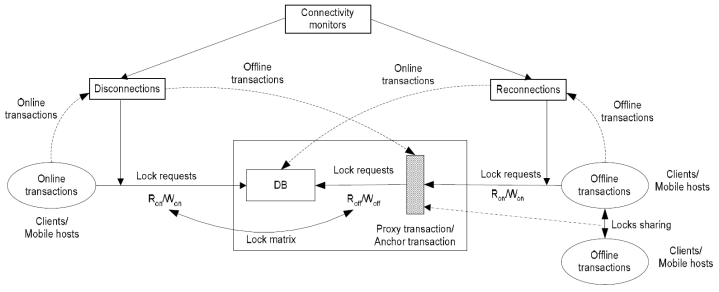
Figur 10 viser de alternerende låseoperasjonene for den adaptive låseprotokollen.



Figur 10: Alternerende låseoperasjoner for den adaptive låseprotokollen.

5.4.13 Modell for låseprotokollen

Figur 11 viser en modell for låseprotokollen [2]. Denne modellen oppsummerer



Figur 11: Modell for den adaptive låseprotokollen

den adaptive låseprotokollen og legger grunnlaget for implementeringen.

6 Konstruksjonsdokumentasjon

En del av denne diplomoppgaven er å designe, implementere og teste låseprotokollen online-offline i Java. Videre skal denne implementeringen utvides til en implementering av en adaptiv låseprotokoll. I dette kapittelet beskrives overordnet design, detaljert design og sekvensdiagram til online-offline. I dette kapittelet beskrives også overordnet design, detaljert design og sekvensdiagram til den adaptive låseprotokollen. Siden designet av den adaptive låseprotokollen utvider designet av online-offline, blir bare utvidelsene beskrevet og ikke det fullstendige designet.

6.1 Overordnet systembeskrivelse for online-offline

Dette avsnittet inneholder en overordnet systembeskrivelse for implementeringen av online-offline.

6.1.1 Designstrategi

Designet av låseprotokollen har tatt utgangspunkt i det mobile miljøet som ble beskrevet i artikkelen. Det mobile miljøet som er beskrevet består av en sentral databasetjener og et sett mobile enheter. Som beskrevet i 4.1 består også et mobilt miljø av mobile støttestasjoner. Mobile støttestasjoner ble inkludert i designet, men fungerer bare som et mellomledd for å ikke gjøre det ulikt det mobile miljøet som ble presentert i artikkelen.

Strategien for designet var å dele implementeringen inn i mest mulig enkeltstående moduler. For større moduler vil kommunikasjonen gå gjennom grensesnittklasser. For mindre moduler er det ikke lagt opp til grensesnittklasser, men direkte kommunikasjon med klassene i modulen. En grensesnittklasse er en klasse som ligger i forkant av hver modul. Kommunikasjon inn til modulen skal gå gjennom denne. Begrunnelsen for et slikt design er å begrense antallet avhengigheter mellom modulene. Ved at andre moduler ikke er avhengig av hvordan en modul er implementert internt kan de betrakte den som en sort boks.

6.1.2 Moduler

For å simulere det mobile miljøet og for å implementere låseprotokollen ble følgende moduler valgt:

- *dataBase*
- *mobileHost*
- *transaction*
- *lock*
- *operation*
- *input*

Modulen *dataBase*
I det mobile miljøet som låseprotokollen skal implementeres i finnes det en databasetjener. Databasetjeneren skal tilby de mobile enhetene alle nødvendige funksjoner. Den må ha ressurser som de mobile enhetene kan låse. Den må ha fasiliteter for å starte, committe og abortere transaksjoner. De mobile enhetene må også kunne utføre lese- og skriveoperasjoner på ressursene i databasen. Databasetjeneren må også kunne ta i mot låseforespørsler på ressursene og håndtere disse som beskrevet i online-offlineprotokollen. De mobile enhetene skal også kunne initiere planlagte frakoblinger og koble til databasen igjen. De mobile enhetene kan også koble fra uplanlagt. Databasetjeneren må ha fasiliteter for å oppdage mobile enheter som er frakoblet uplanlagt. Videre må databasetjeneren logge operasjonene som utføres av de mobile enhetene på ressursene. Som spesifisert i online-offline må databasetjeneren også ha en låselogg og en konverteringslogg. Låseloggen holder rede på hvilke låseforespørsler databasetjeneren mottar og til hvilken tid de ankommer. Denne loggen skal sikre at det ikke oppstår konflikter og at de mobile enhetene leser konsistente verdier. Konverteringsloggen holder rede på til hvilken lås en lås skal konverteres til ved en delegasjon.

Modulen *dataBase* vil inneholde alle klasser for å kunne tilby funksjonaliteten som er beskrevet. All kommunikasjon inn til modulen vil gå gjennom grensesnittklassen `DatabaseServer`. Kommunikasjonen går via klassen `MobileSupportStation` som også ligger i denne modulen. Den mobile støttestasjonen er altså et mellomledd mellom databasetjeneren og de mobile enhetene.

Låseprotokollen er implementert i klassen `LockManager`. Denne klassen håndterer alle låseforespørsler og andre situasjoner som innebærer låsene på ressursene til databasetjeneren. Et eksempel på en annen situasjon er, når den mobile enheten ønsker å koble fra databasen.

Modulen *mobileHost*
Det mobile miljøet som låseprotokollen skal implementeres i, inneholder mobile enheter. De mobile enhetene skal ha fasiliteter for å initiere og utføre transaksjoner. For å utføre transaksjonene sender de mobile enhetene låseforespørsler til databasetjeneren. De skal kunne initiere planlagte frakoblinger, koble fra uplanlagt og koble til databasetjeneren igjen. De mobile enhetene inneholder et cache, der de kan cache ressurser de har fått bevilget låser på. Et cache må til for å kunne simulere at de mobile enhetene utfører databaseoperasjoner lokalt i frakoblet modus. Siden de mobile enhetene kan utføre databaseoperasjoner lokalt, må det også eksistere en logg som logger disse operasjonene. De mobile enhetene må i tillegg ha en låselogg, som holder på tidspunktet de fikk bevilget wiofflåser. Dette gjøres for å kunne oppdage konflikter når de mobile enhetene tilkobles etter en frakobling.

Modulen *mobileHost* vil inneholde alle klasser for å kunne tilby funksjonaliteten som er beskrevet. All kommunikasjon inn til modulen vil gå gjennom grensesnittklassen `mobileHost`.

Modulen *transaction*
Mobile enheter skal kunne utføre transaksjoner. Denne modulen inneholder én klasse: `Transaction`. Denne modulen dekker funksjonaliteten til en transaksjon.

Modulen *lock*
De forskjellige låsetypene i låseprotokollen er samlet i modulen *Lock*. Alle låsene implementerer grensesnittklassen `Lock`.

Modulen *operation*
De forskjellige operasjonene som transaksjoner kan utføre er samlet i modulen *operation*. Alle de forskjellige operasjonene implementerer grensesnittklassen `Operation`.

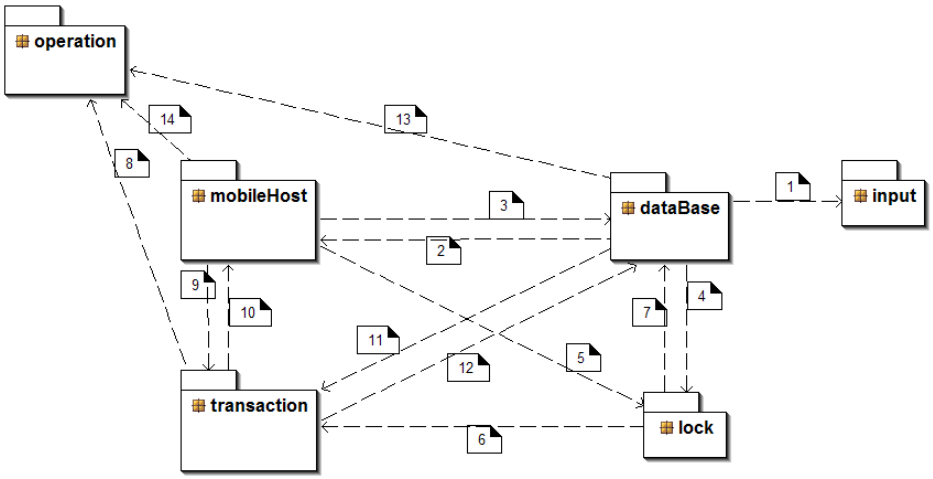
Modulen *input*
For å simulere handlingene til de mobile enhetene ble det valgt å lage inndatafiler. Disse inndatafilene, som beskriver hva de mobile enhetene skal gjøre til spesifiserte tider. Inndatafilene muliggjør testing av låseprotokollen for alle tenkelige situasjoner. Klasser for å lese inn tekstfiler finnes i denne modulen.

6.1.3 Avhengighetsbeskrivelse

En avhengighet mellom to klasser finnes hvis:

- en klasse sender en melding til en annen klasse.
- en klasse har en annen klasse som en del av sin informasjon.
- en klasse nevner en annen klasse som en parameter i en operasjon.

Figur 12 viser avhengighetene mellom modulene. Hva de forskjellige avhen-



Figur 12: Avhengigheter mellom modulene.

gighetene innebærer er:

1. Modulen *dataBase* benytter seg av modulen *input*, når den ønsker å lese inn inndatafilen.
2. Modulen *dataBase* kommuniserer med modulen *mobileHost* gjennom grensesnittklassen `MobileHost`. Kommunikasjonen foregår via klassen

`MobileSupportStation` i modulen *dataBase*, som er et bindeledd mellom databasetjeneren og de mobile enhetene.

3. Modulen *mobileHost* kommuniserer med modulen *dataBase* gjennom grensesnittklassen `DatabaseServer`. Kommunikasjonen foregår via klassen `MobileSupportStation` i modulen *dataBase*, som er et bindeledd mellom databasetjeneren og de mobile enhetene.

4. Modulen *dataBase* benytter seg av låsene i modulen *lock* gjennom grensesnittet `Lock`. Klassen `LockManager` i modulen *dataBase*, vil også aksessere de forskjellige låsetypene når den får låseforespørsler.

5. Modulen *mobileHost* benytter seg av låsene i modulen *lock* gjennom grensesnittet `Lock`.

6. Låsene i modulen *lock* vil holde på hvilken ressurs i modulen *dataBase* de låser.

7. Låsene i modulen *lock* vil holde på hvilken transaksjon i modulen *transaction* som holder låsen.

8. Transaksjoner vil benytte seg av operasjonene i modulen *operation* når den utfører operasjoner.

9. Modulen *mobileHost* vil benytte seg av modulen *transaction*, når den utfører transaksjoner.

10. Transaksjoner hører til mobil enhet.

11. Modulen *dataBase* vil benytte seg av *transaction* når den håndterer transaksjoner.

12. Transaksjoner holder låser på ressurser i modulen *dataBase*.

13. Modulen *dataBase* vil benytte seg av klasser i modulen *operation* når det utføres databaseoperasjoner.

14. Modulen *mobileHost* vil benytte seg av klasser i modulen *operation* når det utføres databaseoperasjoner.

6.1.4 Viktige overordnede designbeslutninger

De mobile enhetene kan kobles til og fra databasetjeneren. Det ble valgt å bruke en boolsk verdi, `connectionStatus`, for å emulere tilkoblingsstatusen til de mobile enhetene. Hvis `connectionStatus` er satt til `false` er den mobile enheten frakoblet. Hvis en mobil enhet ønsker å kommunisere med databasetjeneren sjekkes det først om tilkoblingsstatusen er tilkoblet. Hvis den ikke er det må den utføre den ønskede handlingen lokalt.

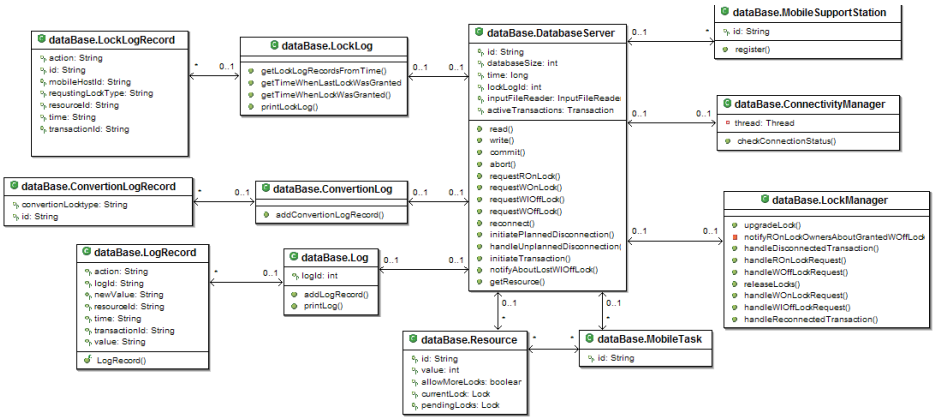
I artikkelen har databasetjeneren mobile oppgaver som de mobile enhetene kan laste ned og utføre. Disse mobile oppgavene er inkludert i designet, men brukes ikke av de mobile enhetene, da disse ikke er nødvendige for å teste låseprotokollen.

6.2 Detaljert design for online-offline

I dette avsnittet beskrives klassene i hver modul.

6.2.1 Klasser i *dataBase*-modulen

Modulen *dataBase* og dens klasser er vist UML-klassediagramet i figur 13. Modulens avhengigheter kan man se i figur 12.



Figur 13: UML-klassediagram for modulen *dataBase*

DatabaseServer

`DatabaseServer` inneholder typiske databasefunksjoner.

`DatabaseServer` holder på instanser av `Resource` og `MobileTask`. Den har en `LockManager`, `ConnectivityManager`, `Log`, `LockLog` og `ConvertionLog`.

Hvis en mobil enhet ønsker å initiere en transaksjon kalles metoden *`initiateTransaction()`* i `DatabaseServer`. For å utføre databaseoperasjoner kalles metodene *`read()`*, *`write()`*, *`commit()`* og *`abort()`* for å lese ressurser, skrive ressurser, committe transaksjoner og abortere transaksjoner. Hvis en mobil enhet ønsker en lås på en ressurs kalles metodene *`requestROnLock()`*, *`requestWOnLock()`*, *`requestWIOffLock()`* og *`requestWOffLock()`*. `DatabaseServer` spør da sin `LockManager` om de mobile enhetene kan få denne låsen. Mobile enheter kan også initiere en planlagt frakobling ved å kalle metoden *`initiatePlannedDisconnection()`*. Når den mobile enheten tilkobles igjen kalles metoden *`reconnect()`*. Metoden *`handleUnplannedDisconnection()`* kalles av `ConnectivityManager` hvis den oppdager at en mobil enhet har frakoblet uplanlagt. En transaksjon kan miste en `WIOffLock` på en ressurs, hvis for eksempel en annen transaksjon får en `WOffLock` på den samme ressursen. Den mobile enheten får beskjed om dette, ved at `LockManager` kaller metoden *`NotifyAboutLostWIOffLock()`*.

ConnectivityManager

`DatabaseServer` har en instans av `ConnectivityManager` som har ansvaret for å oppdage mobile enheter som har blitt frakoblet uplanlagt. `ConnectivityManager` holder på alle mobile enheter som er tilkoblet og frakoblet. `ConnectivityManager` implementerer grensesnittet `Runnable` og vil starte en egen tråd som med passende mellomrom kaller metoden *`checkConnectionStatus()`*.

Metoden *`checkConnectionStatus()`* sjekker om noen av de mobile enhetene har blitt frakoblet uplanlagt. Hvis det oppdages en frakoblet mobil enhet kalles metoden *`handleUnplannedDisconnection()`* i `DatabaseServer`.

ConvertionLog

`DatabaseServer` må ta vare på til hvilken lås en lås skal konverteres til når en delegasjon utføres. Dette gjøres i `ConvertionLog`. `ConvertionLog` holder på en samling `ConvertionLogRecords`. Når `LockManager` håndterer en låseforespørsel legger den inn en `LocalLogRecord` i `LocalLog`. Samtidig legges det inn en `ConvertionLogRecord` for denne låsen i samlingen av

ConvertionLogRecords, hvis låsen ble bevilget.

ConvertionLogRecord

ConvertionLogRecord er en post i ConvertionLog som tar vare på til hvilken lås en lås skal konverteres til ved en delegasjon. En ConvertionLogRecord har samme ID som tilsvarende LockLogRecord.

LockLog

DatabaseServer må ta vare på alle låseforespørsler. Dette gjøres i LockLog. LockLog holder på en samling instanser av LockLogRecord. Når LockManager mottar en låseforespørsel kaller den *addLockLogRecord()* som legger til en LockLogRecord i samlingen av LockLogRecords.

LockLogRecord

LockLogRecord er en post i LockLog som tar vare på tiden, IDen til den mobile enheten, IDen til transaksjonen, IDen til ressursen, hvilken type lås som ble forespurt, nåværende låser på ressursen, hvilende[14] låser på ressursen og hva som ble gjort i med låseforespørselen.

LockManager

LockManager håndterer alle låseforespørsler og bestemmer om de skal bevilges eller ikke. Dette gjøres basert på kompatibilitetsmatrisen og reglene for online-offline som er beskrevet i avsnitt 4.5.1.

LockManager skal håndtere alle låseforespørsler som databasetjeneren mottar. Dette gjøres i metodene *handleROnLockRequest()*, *handleWOnLockRequest()*, *handleWIOffLockRequest()* og *handleWOffLockRequest()*. Låsene kan alterneres til andre låser. Dette gjøres i metodene *upgradeLock()* og *delegateLock()*. Når en transaksjon ønsker å frigi sine låser kalles *releaseLocks()*. LockManager skal også håndtere transaksjoner som frakobles og tilkobles. Dette håndteres i metodene *handleDisconnectedTransaction()* og *handleReconnectedTransaction()*.

Log

DatabaseServer må ta vare på alle databaseoperasjoner som utføres. Dette gjøres i Log. Log holder på en samling LogRecords. Når en databaseoperasjon

[14]Hvilende = pending.

utføres kalles *addLogRecord()* som legger til en LogRecord i samlingen av LogRecords.

LogRecord
LogRecord er en post i Log som tar vare på til hvilken tid, hvilken transaksjon som utfører operasjonen, hvilken ressurs operasjonen utføres på og hva som utføres.

MobileSupportStation
En MobileSupportStation holder på instanser av MobileHost fra modulen *mobileHost*, som den er ansvarlig for. De mobile enhetene registrerer seg hos den mobile støttestasjonen gjennom metoden *register()*.

MobileTask
DatabaseServer inneholder MobileTasks som MobileHosts kan utføre.

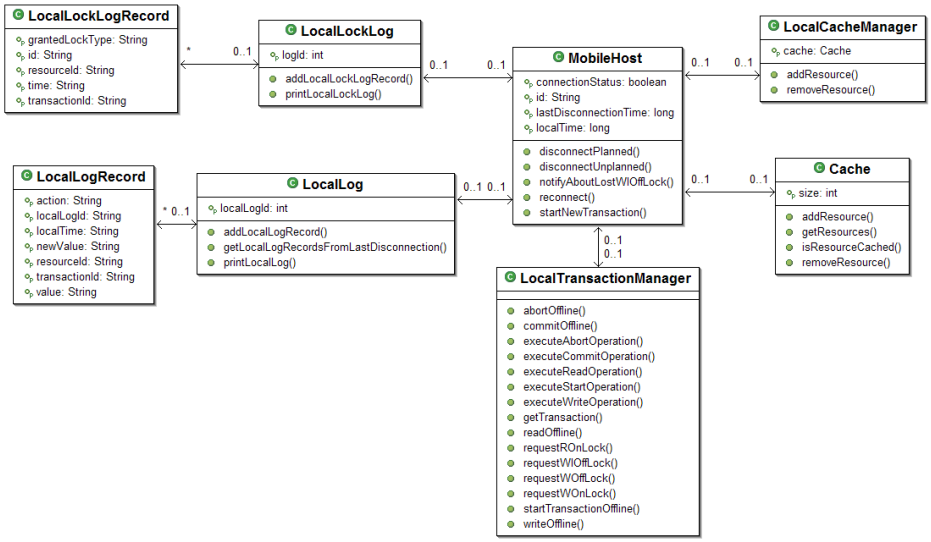
Resource
DatabaseServer inneholder instanser av Resource som transaksjoner kan utføre databaseoperasjoner på. Hver ressurs har en verdi. Resource holder også på en samling over nåværende låser og hvilende låser i attributtene currentLocks og pendingLocks. Hvis en WOffLock eller en WOnLock bevilges på ressursen settes attributten allowMoreLocks til usant.

6.2.2 Klasser i *mobileHost*-modulen

Modulen *mobileHost* og dens klasser er vist UML-klassediagramet i figur 14. Modulens avhengigheter kan man se i figur 12.

MobileHost
MobileHost representerer en mobil enhet og inneholder funksjonene til en mobil enhet.

MobileHost har en instans av Cache, LocalLockLog, LocalLog, LocalCacheManager og LocalTransactionManager. Den kobler til en MobileSupportStation som er mellomleddet til DatabaseServer. MobileHost har også en attributt, connectionStatus, som sier om den mobile enheten er tilkoblet eller frakoblet.



Figur 14: UML-klassediagram for modulen *mobileHost*

`MobileHost` sine handlinger kommer fra inndatafiler, som `DatabaseServer` i modulen *dataBase* leser inn. Den kan frakoble planlagt og uplanlagt og tilkobles ved å kalle *`disconnectPlanned()`*, *`disconnectUnplanned()`* og *`reconnect()`*. Den kan også starte transaksjoner i *`startNewTransaction()`*. Den får også beskjed fra `DatabaseServer` når en av transaksjonene på den mobile enheten mister en `WIOffLock` gjennom *`NotifyAboutLostWIOffLock()`*. En transaksjon må også få beskjed når en `WOffLock` bevilges på en ressurs som har en `ROnLock`. Dette gjøres gjennom metoden *`notifyAboutGrantedWOffLockOnROnLockedResource()`*.

Cache

Hver `MobileHost` har en instans av `Cache` som tar vare på alle ressurser som den mobile enheten ønsker å cache. `Cache` har en størrelse som forteller hvor mange ressurser den kan inneholde.

`Cache` inneholder metoder for å legge til og fjerne ressurser i cachet.

LocalCacheManager

`LocalCacheManager` håndterer cachet til de ulike mobile enhetene. `LocalCacheManager` bestemmer om en ressurs skal legges til cache når de ulike transaksjonene får låser på ressurser og bestemme om ressursen skal fjernes

fra cache når transaksjoner frigir låsene. Dette gjøres gjennom metodene *addResource()* og *removeResource()*.

LocalLockLog
Hver MobileHost må logge når transaksjoner får WIOffLocks på ressurser. LocalLockLog brukes til dette. LocalLockLog holder på en samling instanser av LocalLockLogRecord. Når en transaksjon får en WIOffLock kalles metoden *addLocalLockLogRecord()* som legger til en ny LocalLockLogRecord i samlingen av LocalLockLogRecords.

LocalLockLogRecord
LocalLockLogRecord er en post i LocalLogLog som tar vare på til hvilken tid og på hvilken ressurs en transaksjon får en WIOffLock.

LocalLog
Mobile enheter kan utføre databaseoperasjoner når de er frakoblet. LocalLog logger disse databaseoperasjonene. LocalLog holder på en samling instanser av LocalLogRecord. Når en transaksjon utfører en databaseoperasjon frakoblet kalles metoden *addLocalLogRecord()*, som legger til en ny LocalLogRecord i samlingen av LocalLogRecords.

LocalLogRecord
LocalLogRecord er en post i LogLog som tar vare på til hvilken tid og på hvilken ressurs en transaksjon utfører en databaseoperasjon frakoblet. Den tar også vare på hvilken type operasjon som ble utført. Hvis det ble utført en skriveoperasjon vil den også ta vare på nye og gammel verdi til ressursen.

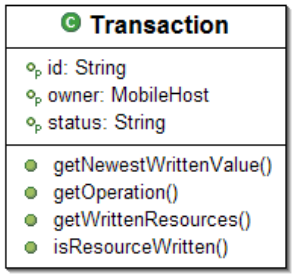
LocalTransactionManager
LocalTransactionManager håndterer transaksjonene til en MobileHost. Den utfører databaseoperasjoner for transaksjonene, både tilkoblet og frakoblet. LocalTransactionManager spør også etter låser for transaksjonene.

Når for eksempel det leses en leseoperasjon fra inndatafilen for en transaksjon som LocalTransactionManager har ansvaret for, vil metoden *executeReadOperation()* kalles. *ExecuteReadOperation()* sjekker om transaksjonen har påkrevd lås og tilkoblingsstatusen til den mobile enheten før den kan utføre leseoperasjonen. Hvis for eksempel en transaksjon har en WIOffLock på

en ressurs og den mobile enheten er frakoblet, må den utføre en leseoperasjon fra en ressurs i cachet til den mobile enheten. Dette gjøres gjennom metoden *`readOffline()`*. Hvis transaksjonen har en `ROnLock` og tilkoblingsstatusen er tilkoblet vil `LocalTransactionManager` kalle *`read()`* i `DatabaseServer`. Når det leses en låseforespørsel etter en `ROnLock` leses fra inndatafilen vil *`requestROnLock()`* kalles som vil spørre `DatabaseServer` om en `ROnLock` kan bevilges denne transaksjonen.

6.2.3 Klasser i *transaction*-modulen

Modulen *transaction* og dens klasser er vist UML-klassediagramet i figur 15. Modulens avhengigheter kan man se i figur 12.

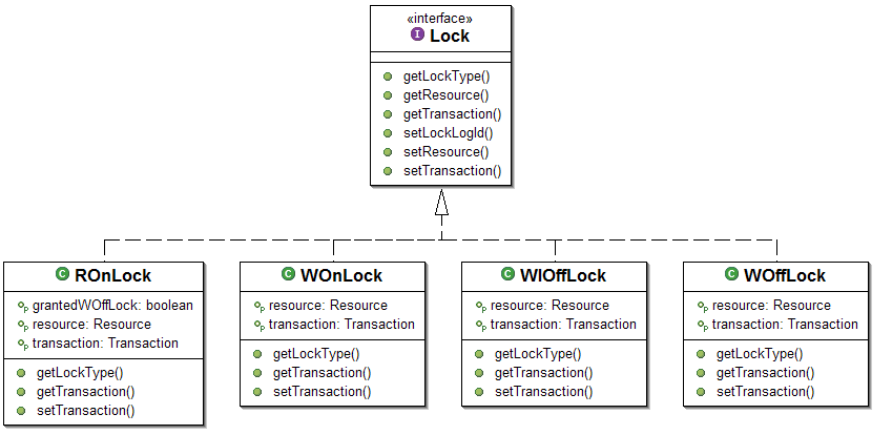


Figur 15: UML-klassediagram for modulen *transaction*

Transaction
`Transaction` er klassen for transaksjoner. Den holder på en samling operasjoner som beskriver hvilke operasjoner denne transaksjonen skal utføre. En transaksjon har en status og holder på hvilke låser den har fått bevilget. Den holder også på de verdiene den har skrevet på ressurser. Når transaksjonen committes, integreres disse verdiene i databasen.

6.2.4 Klasser i *lock*-modulen

Modulen *lock* og dens klasser er vist UML-klassediagramet i figur 16. Modulens avhengigheter kan man se i figur 12.



Figur 16: UML-klassediagram for modulen *lock*

Lock

`Lock` er grensesnittet til de ulike låsetypene. Disse låsetypene er `ROnLock`, `WOnLock`, `WIOffLock` og `WOffLock`. Låsene må holde på hvilken ressurs de låser og hvilken transaksjon som holder låsen.

ROnLock

`ROnLock` implementerer grensesnittet `Lock`. Hvis for eksempel en transaksjon får bevilget en `ROnLock` på lages det en instans av denne klassen og legges til i samlingen over låser i `Resource`. Wofflåser kan bevilges på ressurser selv om det eksisterer ronlåser. Hvis en wofflås bevilges vil attributten `grantedWoffLock` settes til sant hos alle ronlåser på den ressursen.

WIOffLock

`WIOffLock` implementerer grensesnittet `Lock`. Hvis en transaksjon får bevilget en `WIOffLock` på lages det en instans av denne klassen og legges til i samlingen over låser i `Resource`.

WOffLock

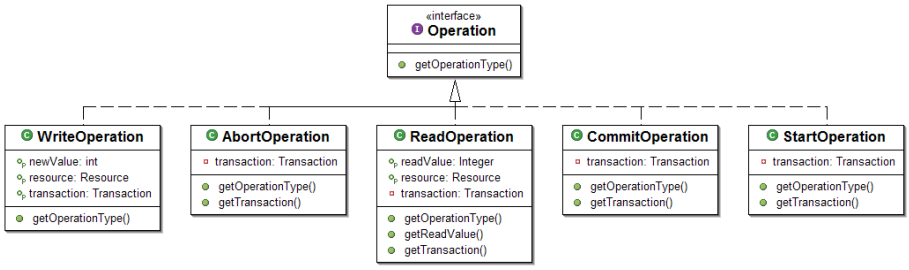
`WOffLock` implementerer grensesnittet `Lock`. Hvis en transaksjon får bevilget en `WOffLock` på lages det en instans av denne klassen og legges til i samlingen over låser i `Resource`.

WOnLock

`WOnLock` implementerer grensesnittet `Lock`. Hvis en transaksjon får bevilget en `WOnLock` på lages det en instans av denne klassen og legges til i samlingen over låser i `Resource`.

6.2.5 Klasser i *operation*-modulen

Modulen *dataBase* og dens klasser er vist UML-klassediagramet i figur 17. Modulens avhengigheter kan man se i figur 12.



Figur 17: UML-klassediagram for modulen *operation*

Operation

`Operation` er grensesnittet for de ulike operasjonene. Disse operasjonene er `StartOperation`, `ReadOperation`, `WriteOperation`, `CommitOperation` og `AbortOperation`.

AbortOperation

`AbortOperation` implementerer grensesnittet `Operation`. Hvis en transaksjon ønsker å utføre en abortoperasjon legges en instans av denne klassen til samlingen av operasjoner som transaksjonen har utført.

CommitOperation

`CommitOperation` implementerer grensesnittet `Operation`. Hvis en transaksjon ønsker å utføre en abortoperasjon legges en instans av denne klassen til samlingen av operasjoner som transaksjonen har utført.

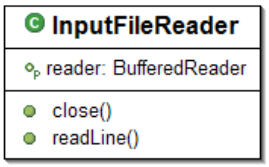
ReadOperation
`ReadOperation` implementerer grensesnittet `Operation`. Hvis en transaksjon ønsker å utføre en leseoperasjon legges en instans av denne klassen til samlingen av operasjoner som transaksjonen har utført.

StartOperation
`StartOperation` implementerer grensesnittet `Operation`. Hvis en transaksjon ønsker å utføre en startoperasjon legges en instans av denne klassen til samlingen av operasjoner som transaksjonen har utført.

WriteOperation
`WriteOperation` implementerer grensesnittet `Operation`. Hvis en transaksjon ønsker å utføre en skriveoperasjon legges en instans av denne klassen til samlingen av operasjoner som transaksjonen har utført.

6.2.6 Klasser i *input*-modulen

Modulen *input* og dens klasser er vist UML-klassediagramet i figur 18. Modulens avhengigheter kan man se i figur 12.



Figur 18: UML-klassediagram for modulen *input*

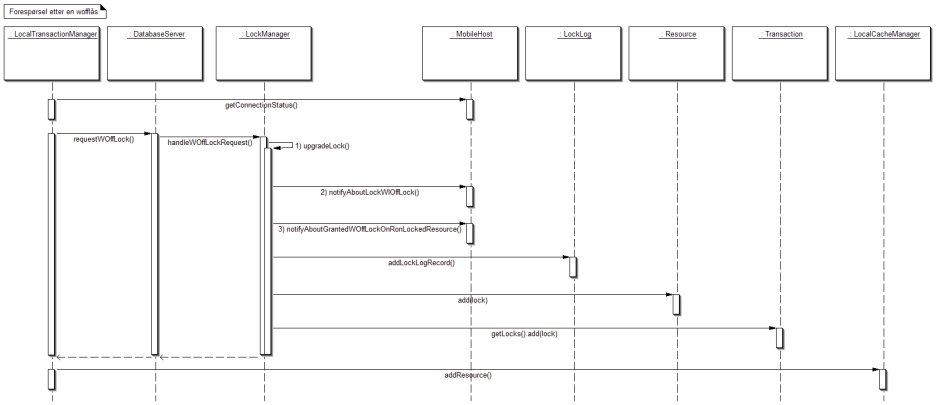
InputFileReader
`InputFileReader` er en klasse for å lese inn testfiler, som inneholder hvilke transaksjoner, databaseoperasjoner og låseforespørsler de mobile enhetene skal utføre. En testfil kan også si når den mobile enheten skal frakobles planlagt og uplanlagt og tilkobles igjen.

6.3 Sekvensdiagram for online-offline

I dette avsnittet vil sekvensdiagrammer for sekvenser som anses som viktige for låseprotokollen online-offline fremlegges.

6.3.1 Forespørsel etter WOffLock

Figur 19 viser hvordan en transaksjon spør etter en `WOffLock`. `LocalTransactionManager` i modulen *mobileHost* får informasjon om når transaksjoner ønsker låser fra inndatafilen. Først sjekkes det om den mobile enheten er til-

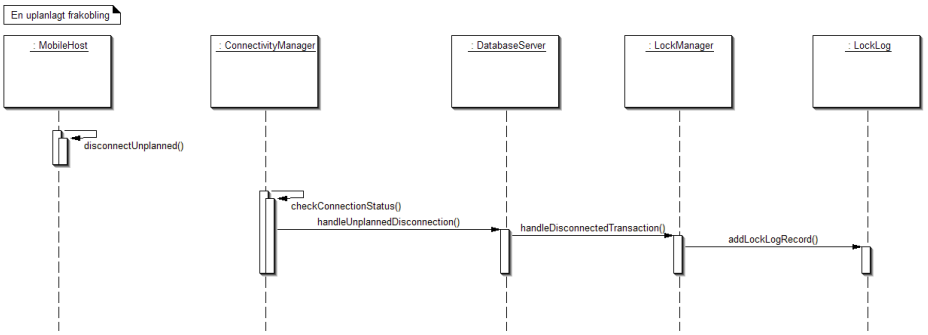


Figur 19: Forespørsel etter WOffLock

koblet databasen. Hvis den er det kan låseforespørselen utføres. `LockManager` håndterer låseforespørselen. 1), 2) og 3) hender bare hvis noen forutsetninger er oppfylt. 1) skjer hvis transaksjonen allerede har en `WIOffLock` eller en `ROnLock` på ressursen. 2) skjer hvis noen transaksjoner mister sin `WIOffLock` på ressursen på grunn av den forespurte `WOffLock`. 3) skjer hvis noen andre transaksjoner har en `ROnLock` på ressursen. Det legges en ny post i `LockLog` uansett om låsen bevilges eller ikke. Hvis låsen bevilges legges den nye låsen i enten `currentLocks` eller `pendingLocks` på ressursen og i låsene til transaksjonen. Til slutt får `LocalTransactionManager` beskjed om låsen ble bevilget eller ikke. Hvis den ble bevilges legges ressursen til i cachet på den mobile enheten.

6.3.2 En uplanlagt frakobling

Figur 20 viser hvordan en mobil enhet utfører en uplanlagt frakobling. Den

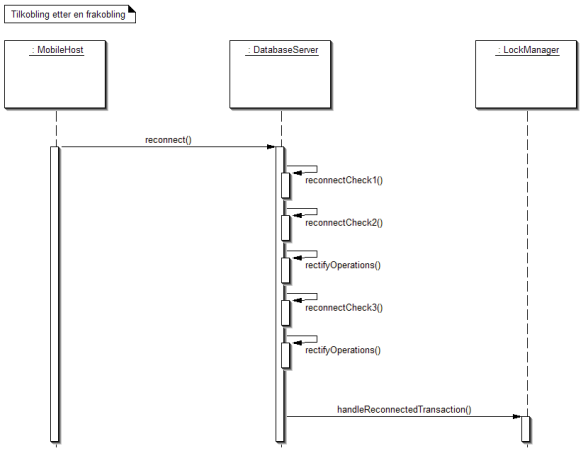


Figur 20: En uplanlagt frakobling

mobile enheten får beskjed fra inndatafilen når den skal frakoble uplanlagt. Dette oppdager `ConnectivityManager`, som gir beskjed til `DatabaseServer`. `DatabaseServer` ber `LockManager` håndtere de frakobla transaksjonene som den mobile enheten har. `LockManager` håndterer transaksjonene etter reglene i online-offline.

6.3.3 Tilkobling etter en frakobling

Figur 21 viser hva som skjer når mobile enheter tilkobles etter en frakobling. Den mobile enheten får beskjed fra inndatafilen når den skal tilkobles.



Figur 21: Tilkobling etter en frakobling

Når dette skjer kaller den mobile enheten *`reconnect()`* i `DatabaseServer`. Denne metoden sjekker om operasjoner som er utført i frakoblet modus er i

konflikt med andre operasjoner i *reconnectCheckN()*, som er private metoder. Disse sjekkene er beskrevet i avsnitt 4.5.1. Hvis en operasjon i konflikt oppdages kalles *rectifyOperation()* som er en korrigerende mekanisme. Andre nødvendige tilkoblingssjekker utføres også. I tillegg kalles metoden *handleReconnectedTransaction()* i `LockManager`. I denne metoden sjekkes det om verdien til wofflåste ressurser har blitt modifisert. Hvis den ikke har blitt modifisert, skal wofflåsen gjøres om til en wiofflås.

6.4 Overordnet systembeskrivelse for den adaptive låseprotokollen

Dette avsnittet inneholder en overordnet systembeskrivelse for implementeringen av den adaptive låseprotokollen.

6.4.1 Designstrategi

Strategien for utvidelsen var å fortsette den strategien som ble satt for online-offline. De modulene som ble valgt for online-offline er beholdt og i tillegg er en ny modul innført. Denne modulen er kalt *mobileAffiliation*. I det mobile miljøet som er beskrevet for den adaptive låseprotokollen, kan mobile enheter opprette mobile affiliasjoner der de kan dele data. Den nye modulen inneholder de nødvendige klassene for at mobile enheter kan opprette mobile affiliasjoner og dele data.

6.4.2 Moduler

Modulene i den adaptive låseprotokollen er:

- *dataBase*
- *mobileHost*
- *transaction*
- *lock*
- *operation*
- *input*
- *mobileAffiliation*

Modulen *dataBase*
I den adaptive låseprotokollen finnes det onlinetransaksjoner. Disse transaksjonene initieres av mobile enheter og utføres på databasetjeneren. I tillegg til funksjonaliteten som ble beskrevet for online-offline, må databasetjeneren ha en transaksjonsbestyrer som håndterer disse transaksjonene. Browselåser, som er en ny låsetype i forhold til online-offline, ble innført i den adaptive

låseprotokollen. Databasetjeneren må inkludere denne låsetypen og håndtere låseforespørsler etter reglene som ble beskrevet for denne låsetypen. I tillegg må databasetjeneren sikre konsistens for transaksjoner som holder disse låsene.

Modulen *mobileHost*
I den adaptive låseprotokollen skilles det mellom onlinetransaksjoner og offlinetransaksjoner. De mobile enhetene skal her ha fasiliteter for å initiere begge typene, men bare utføre offlinetransaksjoner.

Modulen *transaction*
I den adaptive låseprotokollen er det flere typer transaksjoner. Denne modulen skal ha klasser for å dekke onlinetransaksjoner, offlinetransaksjoner, ankertransaksjoner, importeringstransaksjoner og eksporteringstransaksjoner.

Modulen *lock*
Denne modulen inkluderer browselås, i tillegg til låsene fra online-offline.

Modulen *operation*
Denne modulen er lik modulen fra online-offline.

Modulen *input*
Denne modulen er lik modulen fra online-offline.

Modulen *mobileAffiliation*
I det mobile miljøet som er beskrevet for den adaptive låseprotokollen, kan offlinetransaksjoner opprette mobile affiliasjoner der de kan dele data. Denne modulen inneholder fasiliteter for dette.

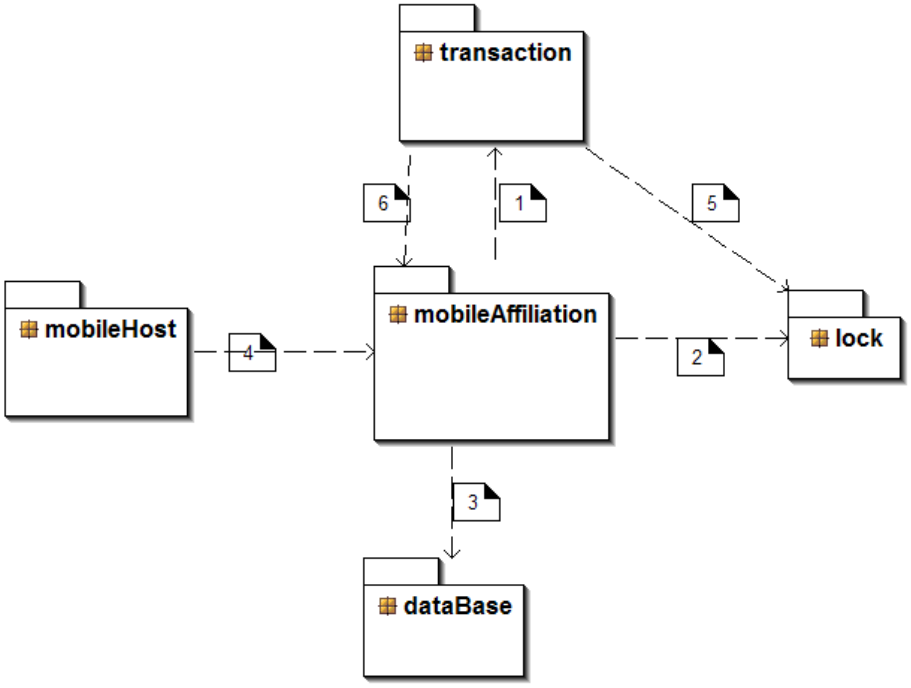
6.4.3 Avhengighetsbeskrivelse

En avhengighet mellom to klasser finnes hvis:

- en klasse sender en melding til en annen klasse.
- en klasse har en annen klasse som en del av sin informasjon.

- en klasse nevner en annen klasse som en parameter i en operasjon.

Figur 22 viser de nye avhengighetene mellom modulene som har oppstått etter utvidelsen. Hva de forskjellige avhengighetene innebærer er:



Figur 22: Nye avhengigheter mellom modulene.

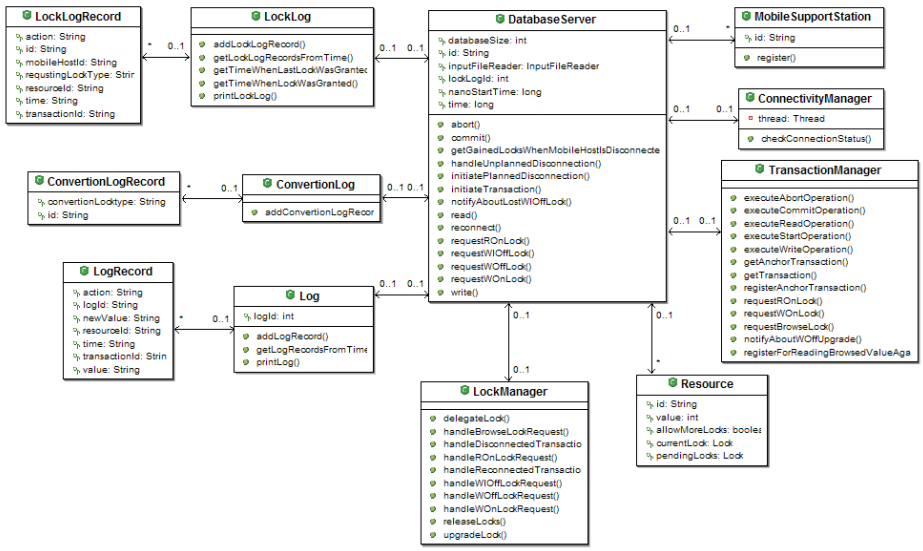
1. `MobileAffiliation` har `ExportTransaction` og `ImportTransaction` som en del av sin informasjon.
2. `SharedData` har `Lock` som en del av sin informasjon.
3. `SharedData` har `Resource` som en del av sin informasjon.
4. `LocalTransactionManager` kommuniserer med `MobileAffiliation`. Dette gjør den når den ønsker å eksportere og importere data.
5. `AnchorTransaction` holder på browselåser til transaksjoner som må lese ny modifisert verdi når de tilkobles.
6. `ExportTransaction` og `ImportTransaction` holder på hvilke mobile affiliasjoner de er tilknyttet.

6.5 Detaljert design for den adaptive låseprotokollen

I dette avsnittet beskrives klassene i hver modul.

6.5.1 Klasser i *dataBase*-modulen

Modulen *dataBase* og dens klasser er vist i UML-klassediagrammet i figur 23.



Figur 23: UML-klassediagram for modulen *dataBase*

Klasser som ikke er forandret fra online-offline blir ikke beskrevet. Disse klassene er:

- `ConnectivityManager`
- `ConvertionLog`
- `ConvertionLogRecord`
- `LockLog`
- `LockLogRecord`
- `Log`

- `LogRecord`
- `MobileSupportStation`
- `Resource`

I tillegg er klassen `MobileTask` fjernet fra modulen, da denne ikke brukes.

DatabaseServer
En utvidelse av `DatabaseServer` er at den har en `TransactionManager`, som skal utføre onlinetransaksjoner.

Databasetjeneren tar i mot browselåsforespørsler i den adaptive låseprotokollen, som håndteres i metoden *`requestBrowseLock()`*.

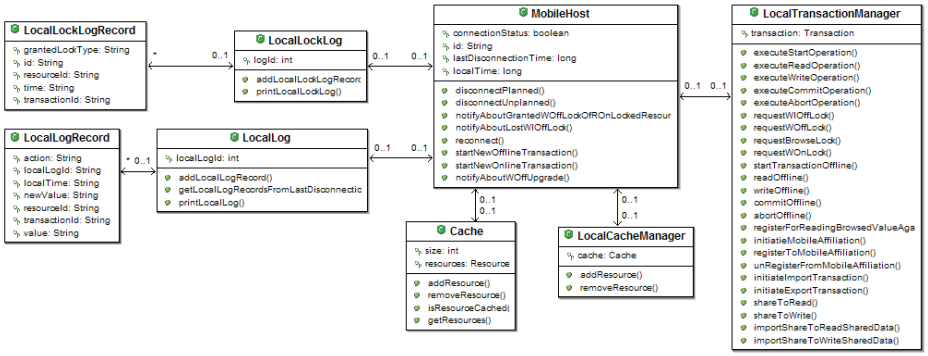
LockManager
`LockManager` bestemmer om browselåsforespørselene skal bevilges eller ikke. Dette gjøres i metoden *`handleBrowseLockRequest()`* som kalles av `DatabaseServer` når den mottar en browselåsforespørsel.

TransactionManager
`TransactionManager` utfører onlinetransaksjoner. Den holder derfor på en samling transaksjoner. `TransactionManager` holder også på en samling ankertransaksjoner, siden disse skal oppholde seg på databasetjeneren.

`TransactionManager` har metoder for å utføre typiske databaseoperasjoner. Disse metodene er: *`executeStartOperation()`*, *`executeReadOperation()`*, *`executeWriteOperation()`*, *`executeCommitOperation()`* og *`executeAbortOperation()`*. Inndatafilen forteller når onlinetransaksjonene skal utføre disse databaseoperasjonene. Onlinetransaksjoner spør etter ronlåser, wonlåser og browselåser. Metodene for dette er: *`requestROnLock()`*, *`requestWOnLock()`* og *`requestBrowseLock()`*. Transaksjoner som har browset en ressurs får beskjed når wofflåsen oppgraderes til en wonlås. Når dette skjer kaller `LockManager` metoden *`notifyAboutWOffUpgrade()`* i `TransactionManager`. Transaksjonene som holder browselås på denne ressursen skal lese den nye verdien når den er committet. `TransactionManager` registrerer disse for å lese den nye verdien i metoden *`registerForReadingBrowsedValueAgain()`*. Mobile enheter må også registrere sine ankertransaksjoner hos `TransactionManager`. Dette gjør de mobile enhetene ved å kalle metoden *`registerAnchorTransaction()`*.

6.5.2 Klasser i *mobileHost*-modulen

Modulen *mobileHost* og dens klasser er vist i UML-klassediagrammet i figur 24.



Figur 24: UML-klassediagram for modulen *mobileHost*

Klasser som ikke er forandret fra online-offline blir ikke beskrevet. Disse klassene er:

- `Cache`
- `LocalCacheManager`
- `LocalLockLog`
- `LocalLockLogRecord`

MobileHost

`MobileHost` utvides med at den holder på en ankertransaksjon. Denne ankertransaksjonen registreres hos databasetjeneren.

I tillegg får den mobile enheten beskjed når en wofflås oppgraderes til en wonlås, når en av dens offlinetransaksjoner holder en browselås på dette databaseelementet. Dette får den beskjed om gjennom metoden *`notifyAboutWOffUpgrade()`*.

LocalTransactionManager

`LocalTransactionManager` håndterer og utfører offlinetransaksjoner. Dette

gjøres på samme måte som i implementeringen av online-offline, men nå utføres alt lokalt og alle låseforespørsler går gjennom ankertransaksjonen. Dette gjelder også commit og abortering av offlinetransaksjonene.

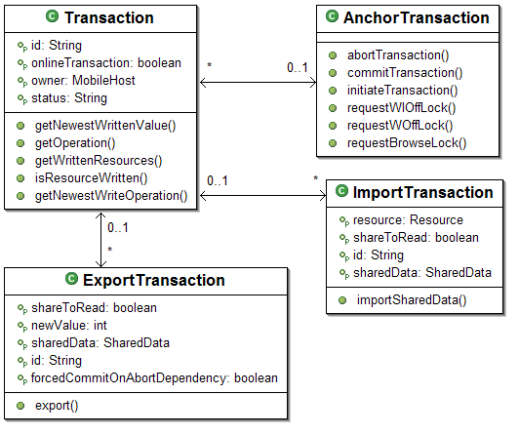
Offlinetransaksjoner forespør etter wiofflåser, wofflåser og browselåser. I tillegg spør de om oppgradering av wofflåser til wonlåser, når offlinetransaksjonene ønsker å integrere arbeidet som er utført lokalt i databasen. Metodene for å spørre etter låser er: *requestROnLock()*, *requestWOnLock()*, *requestBrowseLock()* og *requestWOnLock()*.

Offlinetransaksjon kan også initiere og registrere mobile affiliasjoner. Dette skal `LocalTransactionManager` håndtere. Inndatafilen forteller når offlinetransaksjonene skal gjøre dette. Metoden *initiateMobileAffiliation()* brukes når en offlinetransaksjon ønsker å opprette en mobil affiliasjon. Når en transaksjon ønsker å registrere seg hos en mobil affiliasjon brukes *registerToMobileAffiliation()*. For avregistrering brukes *unregisterFromMobileAffiliation()*. Offlinetransaksjoner kan dele ressurser med andre offlinetransaksjoner i disse mobile affiliasjonene. Det er to måter å dele data: dele-for-lesing og del-for-skriving. Metodene som brukes for dette er: *shareToRead()* og *shareToWrite()*. Inndatafilen forteller når og hvordan offlinetransaksjoner skal dele data. På samme måte kan offlinetransaksjoner importere delt data som er delt med dele-for-lesing eller dele-for-skriving. Metodene som brukes for å importere data er: *importSharedToReadSharedData()* og *importSharedToWriteSharedData()*. For importering og eksportering av data til mobile affiliasjoner brukes importeringstransaksjoner og eksporteringstransaksjoner. Metodene for importering og eksportering initierer disse transaksjonene gjennom metodene: *initiateImportTransaction()* og *initiateExportTransaction()*.

6.5.3 Klasser i *transaction*-modulen

Modulen *transaction* og dens klasser er vist i UML-klassediagrammet i figur 25.

Transaction
I den adaptive låseprotokollen skiller man mellom onlinetransaksjoner og offlinetransaksjoner. Klassen `Transaction` brukes for begge, men en boolsk verdi, `onlineTransaction`, forteller hvilken type transaksjon det er. Hvis det er en onlinetransaksjon holder den på hvilken `TransactionManager` den befinner seg hos. Hvis det er en offlinetransaksjon holder den på hvilken



Figur 25: UML-klassediagram for modulen *transaction*

`LocalTransactionManager` den befinner seg hos. I tillegg holder offline-transaksjoner på ankertransaksjoner den har og eventuelle importerings- og eksporteringstransaksjoner den har initiert. Offlinetransaksjoner holder også på hvilke eksporteringstransaksjoner den har en tvunget-commit-ved-abort-avhengighet til. Hvis denne offlinetransaksjonene aborterer må disse eksporteringstransaksjonene committe og ta ansvar for å integrere den eksporterte ressursen i databasen.

AnchorTransaction

`AnchorTransaction` holder på alle offlinetransaksjoner den er ansvarlig for og offlinetransaksjonenes låseoperasjoner og databaseoperasjoner.

`LocalTransactionManager` sender offlinetransaksjonenes låseforespørsler via ankertransaksjonen. Metodene som brukes er: *`requestROnLock()`*, *`requestWOnLock()`*, *`requestBrowseLock()`* og *`requestWOnLock()`*.

`LocalTransactionManager` sender offlinetransaksjonenes initierings-, commit- og abortforespørsler via ankertransaksjonen. Metodene som brukes er: *`initateTransaction()`*, *`commitTransaction()`* og *`abortTransaction()`*.

ExportTransaction

`ExportTransaction` holder på hvilken transaksjon som initierte den og hvilken mobil affiliasjon den er tilknyttet. Den har en boolsk verdi som forteller om den ble brukt under dele-for-lesing eller dele-for-skriving. I tillegg holder den på en instans av `SharedData` fra modulen *mobileAffiliation*. Dette er det

delte databaseelementet.

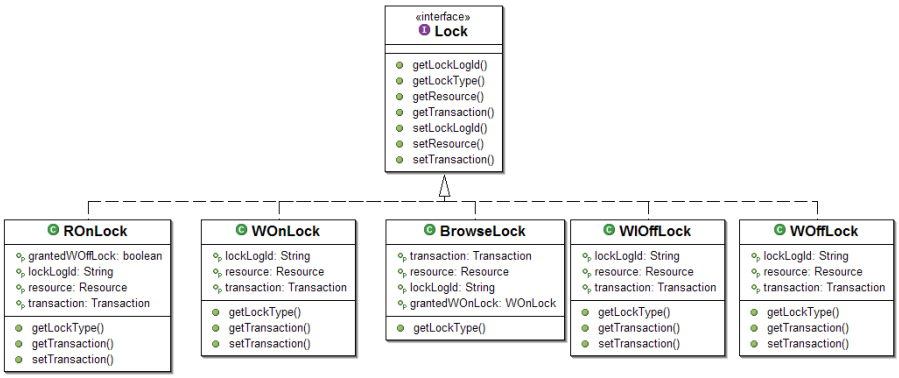
Metoden *`export()`* kalles når eksporteringstransaksjonen skal eksportere et databaseelement.

ImportTransaction
`ImportTransaction` holder på hvilken transaksjon som initierte den og hvilken mobil affiliasjon den er tilknyttet. Den har en boolsk verdi som forteller om den ble brukt under dele-for-lesing eller dele-for-skriving. I tillegg holder den på en instans av `SharedData` fra modulen *mobileAffiliation*. Dette er det delte databaseelementet.

Metoden *`importSharedData()`* kalles når importeringstransaksjonen skal importere et databaseelement.

6.5.4 Klasser i *lock*-modulen

Modulen *lock* og dens klasser er vist i UML-klassediagrammet i figur 26.



Figur 26: UML-klassediagram for modulen *lock*

Klasser som ikke er forandret fra online-offline blir ikke beskrevet. Disse klassene er:

- `Lock`
- `ROnLock`
- `WOnLock`

- `WIOffLock`
- `WOffLock`

BrowseLock
`BrowseLock` implementerer grensesnittet `Lock`. Hvis en transaksjon får bevilget en `BrowseLock` på lages det en instans av denne klassen og legges til i samlingen over låser i `Resource`. Hvis wofflåsen oppgraderes til en wonlås, settes `grantedWOnLock` til `true` på browselåsen.

6.5.5 Klasser i *operation*-modulen

Denne modulen er lik den som ble beskrevet i online-offline, bortsett fra en liten forandring i `WriteOperation`.

WriteOperation
Offlinetransaksjoner kan dele verdien som er skrevet i en skriveoperasjon og gi fra seg ansvaret for å committe denne. En boolsk attributt `commitResponsibility` forteller om transaksjonen er ansvarlig for å committe verdien som skriveoperasjonen har modifisert. Videre kan transaksjoner som har browset et databaseelement registrere seg for å få den committede verdien av en skriveoperasjon. `WriteOperation` holder derfor på en samling `ReadOperations` som ønsker å lese den committede verdien.

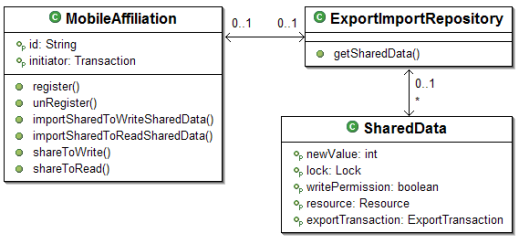
6.5.6 Klasser i *input*-modulen

Denne modulen er lik den som ble beskrevet i online-offline.

6.5.7 Klasser i *mobileAffiliation*-modulen

Modulen *mobileAffiliation* og dens klasser er vist i UML-klassediagrammet i figur 27.

MobileAffiliation
`MobileAffiliation` er grensesnittklassen for modulen *mobileAffiliation*. Alle



Figur 27: UML-klassediagram for modulen *mobileAffiliation*

kommunikasjon inn til modulen går gjennom denne klassen. `MobileAffiliation` har et `ExportImportRepository` som holder på instanser av `SharedData`. I tillegg holder `MobileAffiliation` på de registrerte transaksjonene, og i tillegg importeringstransaksjoner og eksporteringstransaksjoner.

`MobileAffiliation` har metoder som transaksjoner kan kalle for å registrere seg og avregistrere seg. Den har også metoder for å dele data med dele-for-lese og dele-for-skriving og tilsvarende metoder for å importere delt data. Disse metodene er: *`shareToRead()`*, *`shareToWrite()`*, *`importShareToReadSharedData()`* og *`importShareToWriteSharedData()`*.

ExportImportRepository
Hver mobil affiliasjon har et `ExportImportRepository`. EI holder på en samling `SharedData` og har en metode for å hente ut disse delte databaseelementene.

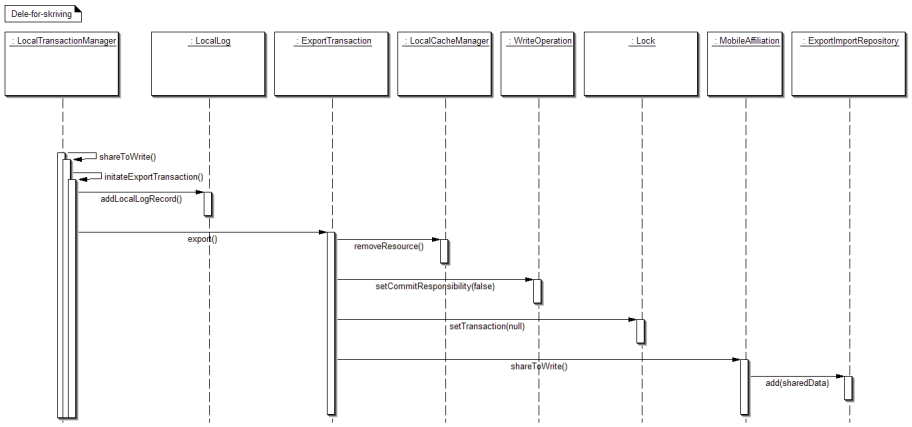
SharedData
Et delt databaseelement består av en ressurs og en ny verdi. I tillegg har det delte databaseelementet en boolsk verdi, `writePermission`, som forteller om andre transaksjoner kan få skriverettighet på ressursen. Dette kan transaksjoner få når databaseelementet er eksportert med dele-til-skrive. I dette tilfellet holder også det delte databaseelementet på wofflåsen som offlinetransaksjonen som eksporterte databaseelementet hadde. Denne låsen tar offlinetransaksjonen som importerer databaseelementet over.

6.6 Sekvensdiagram for den adaptive låseprotokollen

I dette avsnittet vil sekvensdiagrammer for sekvenser som anses som viktige for den adaptive låseprotokollen fremlegges.

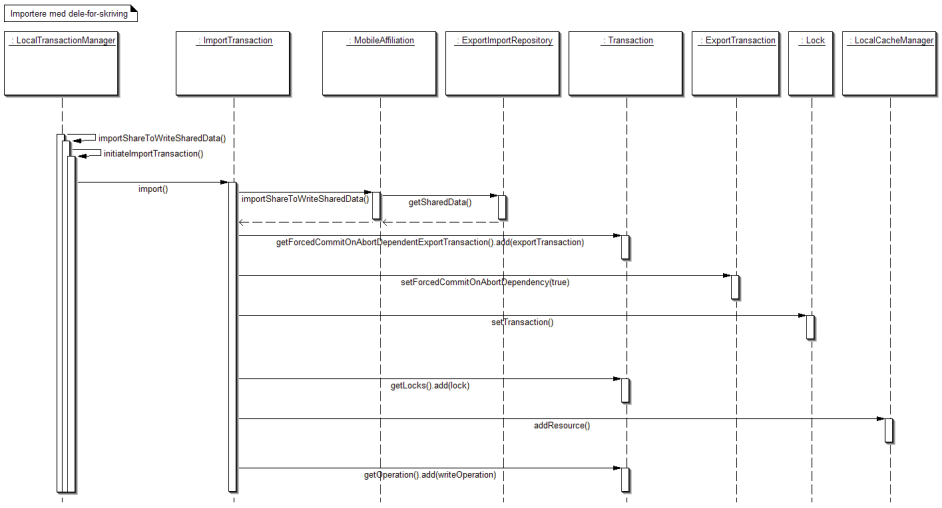
6.6.1 Eksportere data med dele-for-skriving

Figur 28 viser hvordan en offlinetransaksjon eksporterer et delt databaseelement med en ny verdi til EI. `LocalTransactionManager` får beskjed fra inn-



Figur 28: Eksportere med dele-for-skriving

datafilen når en offlinetransaksjon skal dele en ressurs med dele-for-skriving. Inndatafilen forteller hvilken offlinetransaksjon som ønsker å dele en ressurs og i hvilken mobil affiliasjon ressursen skal eksporteres. `LocalTransaction-Manager` initierer en `ExportTransaction` som skal håndtere delingen. Håndtering av delingen foregår i metoden *`export()`*. I denne metoden fjernes ressursen fra cachet til den mobile enheten, hvis offlinetransaksjonen er den siste transaksjonen til den mobile enheten som behøver denne ressursen. I tillegg gir offlinetransaksjonen fra seg wofflåsen, som den holder på ressursen. Dette gjøres ved å sette attributten `transaction` lik `null` på låsen. Denne låsen kan andre offlinetransaksjoner, som ønsker å importere ressursen, ta over. Offlinetransaksjonen er ikke lenger ansvarlig for å committe den modifiserte verdien av ressursen. For å håndtere dette settes attributten `commitResponsibility` på skriveoperasjonen som modifiserte ressursen lik false. Eksporteringstransaksjonen eller offlinetransaksjonen som importerer den delte ressursen blir nå ansvarlig for å committe denne verdien. Eksporteringstransaksjoner kaller til slutt *`shareToWrite()`* hos `MobileAffiliation`. Har lages en ny instans av `SharedData` som legges i `ExportImportRepository`, som kan importeres av andre offlinetransaksjoner.



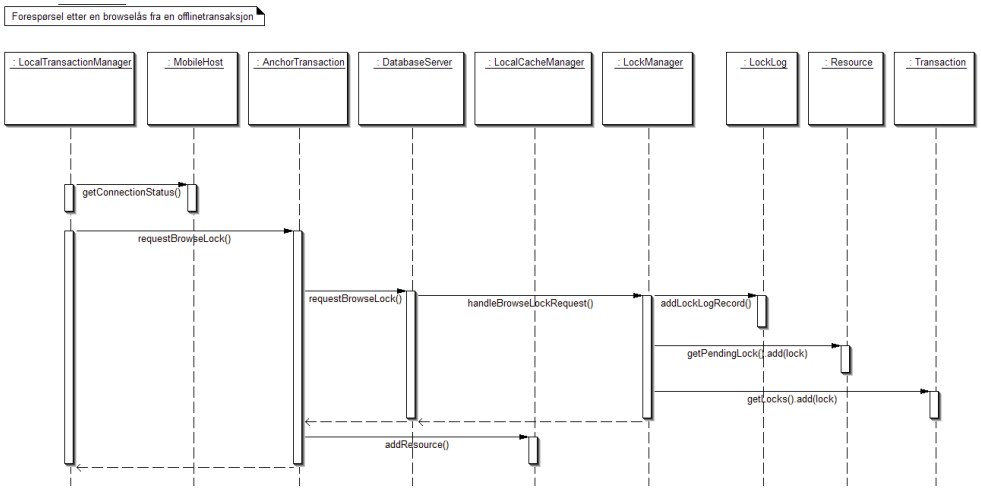
Figur 29: Importere med dele-for-skriving

6.6.2 Importere data med dele-for-skriving

Figur 29 viser hvordan en offlinetransaksjon importerer et delt databaseelement fra EI. `LocalTransactionManager` får beskjed fra inndatafilen når en offlinetransaksjon ønsker å importere en delt ressurs som er delt med dele-for-skriving. Inndatafilen forteller hvilken mobile affiliasjon offlinetransaksjonen kan finne den delte ressursen. `LocalTransactionManager` initierer en `ImportTransaction` som skal håndtere delingen. Håndtering av deling foregår i metoden *`import()`*. Import kaller *`importShareToWriteSharedData()`* og får returnert en instans av `SharedData`. `SharedData` inneholder all nødvendig informasjon. Offlinetransaksjonen som importerer ressursen danner en tvunget-commit-ved-abort-avhengighet med eksporteringstransaksjonen som eksporterte ressursen. *`import()`* setter opp dette slik at denne avhengigheten blir overholdt. Dette gjøres ved å legge eksporteringstransaksjonen i samlingen av eksporteringstransaksjoner med en slik avhengighet hos offlinetransaksjonen og sette attributten `forcedCommitOnAbortDependency` lik `true` på eksporteringstransaksjonen. Hvis offlinetransaksjonen aborteres må eksporteringstransaksjonen committe den modifiserte verdien til den importerte ressursen. Videre tar offlinetransaksjonen over wofflåsen på ressursen og legger denne i sin samling av låser. Ressursen legges også i cachet, hvis den ikke ligger der fra før. Til slutt legges en ny skriveoperasjon til på offlinetransaksjonen, slik at den skal vite at den har importert en ressursen.

6.6.3 Forespørsel etter en browselås fra en offlinetransaksjon

Figur 30 viser hvordan en offlinetransaksjon forespør etter en browselås. Den



Figur 30: Forespørsel etter browselås for en offlinetransaksjon

eneste forskjellen fra online-offline er at låseforespørselen nå går igjennom ankertransaksjonen.

7 Implementeringsdokumentasjon

Dette kapittelet beskriver hvordan låseprotokollen online-offline og den adaptive låseprotokollen ble implementert.

7.1 Implementeringsdokumentasjon for online-offline låseprotokoll

Dette avsnittet beskriver hvordan låseprotokollen online-offline ble implementert. I dette avnittet forklares det hvordan de viktige metodene for låseprotokollen ble implementert. Den fullstendige kildekoden er gitt i appendiks A.

7.1.1 Regler for LockManager

I artikkelen om online-offline ble det beskrevet frem regler for implementeringen for å kunne støtte uforutsigbare frakoblinger [3]:

1. Hvis det eksisterer en $X(i, ron)$ og den mobile enheten M_j spør etter X_{wioff}, vil $X(j, wioff)$ bli bevilget som hvilende men den nåværende låsen forblir $X(i, ron)$. Hvis M_i frakobles eller gjør seg ferdig, vil $X(i, ron)$ forandres til $X(j, wioff)$.

2. Hvis det eksisterer en $X(i, wioff)$ og den mobile enheten M_j spør etter X_{ron}, vil $X(j, ron)$ bli bevilget som nåværende lås. $X(i, wioff)$ vil bli hvilende. Hvis M_j frakobles eller gjør seg ferdig, vil $X(j, ron)$ forandres tilbake til $X(i, wioff)$.

3. Hvis det eksisterer en $X(i, wioff)$ og den mobile enheten M_j spør etter X_{woff} eller X_{won} , vil $X(j, woff)$ eller $X(j, won)$ bli bevilget som nåværende lås. Ingen andre låser vil bli bevilget etter dette.

4. Hvis det eksisterer en $X(i, ron)$ og den mobile enheten M_j spør etter X_{woff}, vil $X(j, woff)$ bli bevilget som hvilende men den nåværende låsen forblir $X(i, ron)$. Ingen andre låser vil bli bevilget etter dette. Hvis M_i frakobles eller gjør seg ferdig, vil $X(i, ron)$ forandres til $X(j, woff)$.

5. Hvis det eksisterer en $X(i, woff)$ og verdien av X ikke har blitt forandret under en frakobling, må den mobile enheten nedgradere låsen til en X_{wioff}.

7.1.2 Detaljerte metodebeskrivelser

Dette avsnittet inneholder detaljerte metodebeskrivelser som forklarer hvordan låseprotokollen ble implementert. Disse metodene er:

- *addLockLogRecord()* i LockLog
  Denne metoden legger inn låseloggposter i låseloggen til databasetjeneren.
- *upgradeLock()* i LockManager
  Denne metoden håndterer den alternerende låseoperasjonen oppgradering, som er én av to alternerende låseoperasjonene i låseprotokollen.
- *delegateLock()* i LockManager
  Denne metoden håndterer den alternerende låseoperasjonen delegasjon, som er én av to alternerende låseoperasjonene i låseprotokollen.
- *handleROnLockRequest()* i *LockManager*
  Denne metoden håndterer låseforespørsler etter ronlåser.
- *handleWOnLockRequest()* i LockManager
  Denne metoden håndterer låseforespørsler etter wonlåser.
- *handleWIOffLockRequest()* i LockManager
  Denne metoden håndterer låseforespørsler etter wiofflåser.
- *handleWOffLockRequest()* i LockManager
  Denne metoden håndterer låseforespørsler etter wofflåser.
- *releaseLocks()* i LockManager
  Denne metoden håndterer situasjoner hvor transaksjoner ønsker å frigi låsene sine.
- *handleDisconnectedTransaction()* i LockManager
  Denne metoden håndterer transaksjoner som har blitt frakoblet.
- *handleReconnectedTransaction()* i LockManager
  Denne metoden håndterer transaksjoner som har blitt frakoblet.

addLockLogRecord() **i** LockLog
Denne metoden oppretter en ny instans av LockLogRecord og legger denne til i LockLog. Parametrene metoden tar inn er id, time, mobileHostId,

`transactionId`, `resourceId`, `requestingLockType`, `currentLocks`, `pendingLocks` og `action`. Hver `LockLogRecord` får en id og tar vare på de andre parametrene.

Attributten `action` viser hvilke handlinger `LockManager` kan utføre. Disse handlingene er:

- REJECTED
  Her blir låseforespørselen avvist.
- GRANTED
  Her blir låseforespørselen bevilget og lagt i `currentLocks` hos ressursen.
- GRANTEDPENDING
  Her blir låseforespørselen bevilget og lagt i `pendingLocks` hos ressursen.
- GRANTEDUPGRADE
  Denne låseforespørselen er en oppgraderingsforespørsel, hvor transaksjonen allerede har en lås i `currentLocks` på ressursen. Oppgraderingen er mulig og låseforespørselen bevilges.
- GRANTEDUPGRADEPENDING
  Denne låseforespørselen er en oppgraderingsforespørsel, hvor transaksjonen allerede har en lås i `pendingLocks` på ressursen. Oppgraderingen er mulig og låseforespørselen bevilges.
- GRANTEDSELFUPGRADE
  Det eneste tilfellet dette skjer er når en transaksjon ønsker å oppgradere en `WOffLock` til en `WOnLock`. Låseforespørselen bevilges.
- GRANDTEDUPGRADEANDDELETE
  Denne låseforespørselen er en oppgraderingsforespørsel, hvor transaksjonen allerede har en lås på ressursen. Oppgraderingen er mulig og låseforespørselen bevilges. `WIOffLock`s på ressursen slettes.
- GRANTEDANDTAKEOVER
  Det eneste tilfellet hvor dette skjer er når det spørres etter en `ROnLock` på en ressurs som har `WIOffLock`s i `currentLocks`. Låseforespørselen bevilges og låsen lagt i `currentLocks`. Låsene som ligger i `currentLocks` legges i `pendingLocks`.

- GRANTEDTAKEOVERANDDELETE
  Låseforespørselen blir bevilget. Den nye låsen tar over låsene i `currentLocks` som slettes.

- RELEASED
  Dette er en opplåsingsforespørsel hvor låsen blir fjernet fra `currentLocks` eller `pendingLocks` hos ressursen.

- RELEASEDDELEGATEWIOFFLOCKS
  Det eneste tilfellet hvor denne handlingen utføres er når den siste `ROnLock` i `currentLocks` blir fjernet og det eksisterer `WIOffLocks` i `pendingLocks`. Når den siste `ROnLock` fjernes, flyttes `WIOffLocks` i `pendingLocks` til `currentLocks`.

- RELEASEDDELEGATEWOFFLOCKSANDDELETE
  Det eneste tilfellet hvor denne handlingen utføres er når den siste `ROnLock` i `currentLocks` blir fjernet og det eksisterer `WIOffLocks` og en `WOffLock` i `pendingLocks`. Når den siste `ROnLock` fjernes flyttes `WOffLock` i `pendingLocks` til `currentLocks`, mens `WIOffLocks` slettes.

- DISCONNECTEDDELEGATE
  Når en mobil enhet frakobles skal transaksjoner på den mobile enheten delegere `ROnLocks` til `WIOffLocks`. Ved denne handlingen blir en `ROnLock` delegert til en `WIOffLock`. Hvis det er siste `ROnLock` i `currentLocks` som blir delegert, blir den nye `WIOffLock` lagt i `currentLocks` sammen med andre `WIOffLocks` fra `pendingLocks`. Hvis det ennå er andre `ROnLocks` i `currentLocks` havner den nye `WIOffLock` i `pendingLocks`.

- DISCONNECTEDDELEGATEANDDELETE
  Denne handlingen er en delagasjon av en `ROnLock` til en `WIOffLock`, hvor `ROnLock` er den siste `ROnLock` i `currentLocks` og det eksisterer en `WOffLock` i `pendingLocks`. `WOffLock` legges i `currentLocks`, mens andre `WIOffLocks` i `pendingLocks` slettes.

- RECONNECTDELEGATE
  Hvis en transaksjon som har vært frakobla ikke har forandret verdien på ressursen når den blir tilkoblet, skal `WOffLock` gjøres om til en `WIOffLock`.

upgradeLock() **i** `LockManager`
Det er sju forskjellige situasjoner, som innebærer oppgradering av låser. Disse situasjonene er:

1. En transaksjon spør etter en `ROnLock` og transaksjonen har allerede har en `WIOffLock` i `currentLocks` til ressursen.
2. En transaksjon spør etter en `ROnLock` og transaksjonen har allerede har en `WIOffLock` i `pendingLocks` til ressursen.
3. En transaksjon spør etter en `WOffLock` og transaksjonen har allerede har en `WIOffLock` i `currentLocks` til ressursen.
4. En transaksjon spør etter en `WOffLock` og transaksjonen har allerede har en `ROnLock` i `currentLocks` til ressursen.
5. En transaksjon spør etter en `WOffLock` og transaksjonen har allerede har en `WIOffLock` i `pendingLocks` til ressursen.
6. En transaksjon spør etter en `WOnLock` og transaksjonen har allerede har en `WIOffLock` i `currentLocks` til ressursen.
7. En transaksjon spør etter en `WOnLock` og transaksjonen har allerede har en `WOffLock` i `currentLocks` til ressursen.

Parametrene som metoden tar inn er den nye låsen, den gamle låsen, `currentLocks`, `pendingLocks` og to boolske verdier som forteller om transaksjonen allerede har en lås i `currentLocks` eller `pendingLocks`.

Metoden finner ut hvilken situasjon som er aktuell og foretar handlinger ut fra det. Handlingene for hver situasjon er:

1. For situasjon 1 fjernes `ROnLock` fra `currentLocks`. Eventuelle andre `WIOffLocks` i `currentLocks` blir lagt til i `pendingLocks`. Den nye `ROnLock` blir lagt til i `currentLocks`. Videre fjernes den gamle låsen fra samlingen over låser som transaksjonen holder på og den nye låsen blir lagt til.
2. For situasjon 2 fjernes den gamle `WIOffLock` fra `pendingLocks` og den nye `ROnLock` legges til i `currentLocks`. Samlingen over låser hos transaksjonen oppdateres til slutt.

3. For situasjon 3 fjernes den gamle `WIOffLock` fra `currentLocks` og den nye `WOffLock` legges til. Eventuelle andre `WIOffLocks` slettes og transaksjonene som mister låsen får beskjed om dette gjennom metoden *`notifyAboutLockWIOffLock()`*. Samlingen over låser hos transaksjonen oppdateres til slutt.

4. For situasjon 4 sjekkes det om den gamle `ROnLock` er den eneste `ROnLock` i `currentLocks`. Hvis det er det så fjernes den gamle låsen og den nye låsen `WOffLock` legges til i `currentLocks`. Eventuelle `WIOffLocks` slettes og transaksjonene som mister låsen får beskjed om dette gjennom metoden *`notifyAboutLockWIOffLock()`*. Hvis det er andre `ROnLocks` i `currentLocks` legges den nye `WOffLock` til i `pendingLocks`. Samlingen over låser hos transaksjonen oppdateres til slutt.

5. For situasjon 5 fjernes den gamle `WIOffLock` fra `pendingLocks` og den nye `WOffLock` legges til. Samlingen over låser hos transaksjonen oppdateres til slutt.

6. For situasjon 6 sjekkes fjernes den gamle `WIOffLock` fra `currentLocks` og den nye `WOnLock` legges til. Eventuelle andre `WIOffLocks` slettes og transaksjonene som mister låsen får beskjed om dette gjennom metoden *`notifyAboutLockWIOffLock()`*. Samlingen over låser hos transaksjonen oppdateres til slutt.

7. For situasjon 7 fjernes den gamle `WOffLock` fra `currentLocks` og den nye `WOnLock` legges til. Samlingen over låser hos transaksjonen oppdateres til slutt.

`delegateLock()` **i** `LockManager`

Det er tre situasjoner som innebærer delegasjon av låser. Disse situasjonene er:

1. Hvis en transaksjon committer eller aborterer og frigir låsene sine. Betingelsene som må være oppfylt for at en delegasjon skal hende er at en `ROnLock` som frigis er siste `ROnLock` i `currentLocks` og det eksisterer `WIOffLocks` og/eller `WOffLock` i `pendingLocks`.

2. En `ROnLock` skal delegeres til en `WIOffLock` når den mobile enheten frakobles. Dette skal gjøres både for planlagte og uplanlagte frakoblinger.

3. Hvis en mobil enhet holder en `WOffLock` på en ressurs og verdien på ressursen ikke er forandret når den tilkobles, skal `WOffLock` delegeres til en `WIOffLock`.

Parametrene som denne metoden tar inn er ressursen, de nye `currentLocks` og eventuelle `pendingLocks`. Metoden setter de nye `currentLocks` på ressursen og setter de nye `pendingLocks` hvis det er noen. For situasjon 1 blir *`delegateLock()`* blir kalt fra metoden *`releaseLocks()`* hvis denne metoden finner en situasjon hvor betingelsene er oppfylt. For situasjon 2 blir *`delegateLock()`* blir kalt fra metoden *`handleDisconnectedTransaction()`*. For situasjon 3 blir *`delegateLock()`* blir kalt fra metoden *`handleReconnectedTransaction()`*.

`handleROnLockRequest()` **i** `LockManager`
Denne metoden går gjennom alle situasjoner hvor transaksjonen kan få bevilget en `ROnLock` på ressursen. Hvis den finner en situasjon hvor transaksjonen kan få bevilget en `ROnLock`, vil transaksjonen få låsen. Hvis `LockManager` ikke finner en situasjon blir låseforespørselen avvist. Metoden returnerer `true` hvis låsen bevilges og `false` hvis den avvises. Transaksjonen kan allerede ha en lås på ressursen og *`upgradeLock()`* vil da bli kalt. Situasjonene som sjekkes er:

- Før metoden kan begynne å sjekke situasjoner hvor låsen kan bevilges må den finne ut om `allowMoreLocks` på ressursen er satt til `false`. Hvis dette er tilfelle er det allerede en `WOffLock` eller `WOnLock` på ressursen og låseforespørselen må avvises. Metoden *`addLockLogRecord()`* kalles for å logge handlingen til `LockManager`.

- Første situasjon hvor en `ROnLock` kan bevilges er om ressursen ikke har noen låser. Da legges en ny instans av `ROnLock` til i `currentLocks` og i samlingen over låser hos transaksjonen. Metoden *`addLockLogRecord()`* kalles for å logge handlingen.

- Neste situasjon som låsen kan bevilges er om det eksister andre `ROnLocks` i `currentLocks`. Det sjekkes da om transaksjonen allerede har en `WIOffLock` i `pendingLocks`. Hvis dette er tilfelle kalles metoden *`upgradeLock()`*. Hvis ikke legges en ny instans av `ROnLock` i `currentLocks` og i samlingen over låser hos transaksjonen. Metoden *`addLockLogRecord()`* kalles for å logge hvilken handling som ble utført.

- Neste situasjon hvor en `ROnLock` kan bevilges er om det eksisterer `WIOffLocks` i `currentLocks`. Det sjekkes da om transaksjonen allerede har en `WIOffLock` på ressursen. Hvis den har det kalles *`upgradeLock()`*. Hvis ikke flyttes `WIOffLocks` i `currentLocks` over til `pendingLocks` og en ny `ROnLock` legges til i `currentLocks` og i samlingen av låser hos transaksjonen. Metoden *`addLockLogRecord()`* kalles for å logge hvilken handling som ble utført.

- Hvis ingen av situasjonene over er tilfelle må låsen avvises. Metoden *`addLockLogRecord()`* kalles for å logge denne handlingen.

`handleWOnLockRequest()` **i** `LockManager`
Denne metoden går gjennom alle situasjoner hvor transaksjonen kan få bevilget en `WOnLock` på ressursen. Hvis den finner en situasjon hvor transaksjonen kan få bevilget en `WOnLock`, vil transaksjonen få låsen. Hvis `LockManager` ikke finner en situasjon blir låseforespørselen avvist. Metoden returnerer `true` hvis låsen bevilges og `false` hvis den avvises. Transaksjonen kan allerede ha en lås på ressursen og *`upgradeLock()`* vil da bli kalt. Situasjonene som sjekkes er:

- Den første situasjonen hvor en `WOnLock` kan bevilges er om transaksjonen har en `WOffLock` i `currentLocks`. Hvis dette er tilfelle kalles *`upgradeLock()`*. Metoden *`addLockLogRecord()`* kalles for å logge handlingen.

- Før metoden kan gå videre å sjekke situasjoner hvor låsen kan bevilges må den finne ut om `allowMoreLocks` på ressursen er satt til `false`. Hvis dette er tilfelle har en annen transaksjon en `WOffLock` eller en `WOnLock` på ressursen og låseforespørselen må avvises. Metoden *`addLockLogRecord()`* kalles for å logge handlingen til `LockManager`.

- Neste situasjon hvor en `WOnLock` kan bevilges er om ressursen ikke har noen låser. Da legges en ny instans av `WOnLock` til i `currentLocks` og i samlingen over låser hos transaksjonen. Attributten `allowMoreLocks` på ressursen settes til `false`. Metoden *`addLockLogRecord()`* kalles for å logge handlingen.

- Neste situasjon hvor en `WOnLock` kan bevilges er om det eksisterer `WIOffLocks` i `currentLocks`. Det sjekkes da om transaksjonen allerede har en `WIOffLock` på ressursen. Hvis den har det kalles *`upgradeLock()`*. Hvis ikke slettes `WIOffLocks` fra `currentLocks` og transaksjoner som

mister disse låsene får beskjed gjennom metoden *notifyAboutLost-WIOffLock()*. En ny WOnLock legges til i currentLocks og i samlingen av låser hos transaksjonen. Attributten allowMoreLocks på ressursen settes til false for alle tilfeller. Metoden *addLockLogRecord()* kalles for å logge hvilken handling som ble utført.

- Hvis ingen av situasjonene over er tilfelle må låsen avvises. Metoden *addLockLogRecord()* kalles for å logge denne handlingen.

handleWIOffLockRequest() **i** LockManager
Denne metoden går gjennom alle situasjoner hvor transaksjonen kan få bevilget en WIOffLock på ressursen. Hvis den finner en situasjon hvor transaksjonen kan få bevilget en WIOffLock, vil transaksjonen få låsen. Hvis LockManager ikke finner en situasjon blir låseforespørselen avvist. Metoden returnerer true hvis låsen bevilges og false hvis den avvises. Situasjonene som sjekkes er:

- Før metoden kan begynne å sjekke situasjoner hvor låsen kan bevilges må den finne ut om allowMoreLocks på ressursen er satt til false. Hvis dette er tilfelle er det allerede en WOffLock eller WOnLock på ressursen og låseforespørselen må avvises. Metoden *addLockLogRecord()* kalles for å logge handlingen til LockManager.

- Første situasjon hvor en WIOffLock kan bevilges er om ressursen ikke har noen låser. Da legges en ny instans av WIOffLock til i currentLocks og i samlingen over låser hos transaksjonen. Metoden *addLockLog-Record()* kalles for å logge handlingen.

- Neste situasjon som låsen kan bevilges er om det eksister ROnLocks i currentLocks. Hvis dette er tilfelle legges en ny instans av WIOffLock i pendingLocks og i samlingen over låser hos transaksjonen. Metoden *addLockLogRecord()* kalles for å logge handlingen.

- Neste situasjon hvor en WIOffLock kan bevilges er om det eksisterer andre WIOffLocks i currentLocks. En ny instans av WIOffLock legges da til i currentLocks og i samlingen av låser hos transaksjonen. Metoden *addLockLogRecord()* kalles for å logge handlingen som ble utført.

- Hvis ingen av situasjonene over er tilfelle må låsen avvises. Metoden *addLockLogRecord()* kalles for å logge denne handlingen.

handleWOffLockRequest() **i** LockManager
Denne metoden går gjennom alle situasjoner hvor transaksjonen kan få bevilget en WOffLock på ressursen. Hvis den finner en situasjon hvor transaksjonen kan få bevilget en WOffLock, vil transaksjonen få låsen. Hvis LockManager ikke finner en situasjon blir låseforespørselen avvist. Metoden returnerer true hvis låsen bevilges og false hvis den avvises. Transaksjonen kan allerede ha en lås på ressursen og *upgradeLock()* vil da bli kalt. Situasjonene som sjekkes er:

- Før metoden kan begynne å sjekke situasjoner hvor låsen kan bevilges må den finne ut om allowMoreLocks på ressursen er satt til false. Hvis dette er tilfelle har en annen transaksjon en WOffLock eller en WOnLock på ressursen og låseforespørselen må avvises. Metoden *addLockLogRecord()* kalles for å logge handlingen til LockManager.

- Den første situasjon hvor en WOffLock kan bevilges er om ressursen ikke har noen låser. Da legges en ny instans av WOffLock til i currentLocks og i samlingen over låser hos transaksjonen. Attributten allowMoreLocks på ressursen settes til false. Metoden *addLockLogRecord()* kalles for å logge handlingen.

- Neste situasjon hvor en WOffLock kan bevilges er om det eksisterer WIOffLocks i currentLocks. Det sjekkes da om transaksjonen allerede har en WIOffLock på ressursen. Hvis den har det kalles *upgradeLock()*. Hvis ikke slettes WIOffLocks fra currentLocks og transaksjoner som mister disse låsene får beskjed gjennom metoden *notifyAboutLostWIOffLock()*. En ny WOffLock legges til i currentLocks og i samlingen av låser hos transaksjonen. Attributten allowMoreLocks på ressursen settes til false for alle tilfeller. Metoden *addLockLogRecord()* kalles for å logge hvilken handling som ble utført.

- Neste situasjon hvor en WOffLock kan bevilges er om det eksisterer ROnLocks i currentLocks. Det sjekkes da om transaksjonen allerede har en ROnLock i currentLocks. Hvis den har det kalles *upgradeLock()*. Hvis det er flere enn denne transaksjonen som har ROnLock i currentLocks må de få beskjed om at det ble bevilget en WOffLock på ressursen. Dette gjøres gjennom metoden *notifyAboutGrantedWOffLockOnROnLockedResource()*. Videre kan transaksjonen ha en WIOffLock i pendingLocks. Da kalles også *upgradeLock()*. Her må også andre transaksjoner som har en ROnLock få beskjed. Hvis transaksjonen ikke har noen låser på ressursen legges det en ny instans av WOffLock

til i `pendingLocks` og i samlingen av låser hos transaksjonen. Transaksjonene som har en `ROnLock` på ressursen får beskjed om at det ble bevilget en `WOffLock`. Attributten `allowMoreLocks` på ressursen settes til `false` for alle tilfeller. Metoden *`addLockLogRecord()`* kalles for å logge hvilken handling som ble utført.

- Hvis ingen av situasjonene over er tilfelle må låsen avvises. Metoden *`addLockLogRecord()`* kalles for å logge denne handlingen.

`releaseLocks()` **i** `LockManager`
Denne metoden tar inn en `Transaction` og frigir alle låsene den holder på. Hver gang en lås blir frigitt vil metoden *`addLockLogRecord()`* kalles for å logge handlingen. Hvis det er en `WOffLock` eller en `WOnLock` som frigis vil `allowMoreLocks` settes til `true`. Hvis det er en `ROnLock` som frigis og dette er den siste `ROnLock` i `currentLocks` på ressursen må `LockManager` delegere låser. Hvis det da eksisterer en `WOffLock` i `pendingLocks` vil *`delegateLock()`* kalles hvor `WOffLock` blir den eneste låsen i `currentLocks`. Eventuelle `WIOffLocks` i `pendingLocks` slettes og transaksjonene som holder disse får beskjed gjennom *`notifyAboutLostWIOffLock()`*. Det blir da ingen låser i `pendingLocks`. Hvis det derimot er bare `WIOffLocks` i `pendingLocks` vil *`delegateLock()`* kalles med disse som nye låser i `currentLocks`.

`handleDisconnectedTransaction()` **i** `LockManager`
Transaksjoner på mobile enheter som frakobles som har `ROnLocks` på ressurser må delegere disse til `WIOffLocks`. Denne metoden går gjennom alle transaksjoner som har blitt frakoblet enten planlagt eller uplanlagt og håndter delegasjonen. Hvis metoden finner at transaksjonen har en `ROnLock` på en ressurs som er en av mange `ROnLocks` i `currentLocks`, kalles metoden *`delegateLock()`* for å håndtere delegasjonen. Metoden *`delegateLock()`* tar inn nye `currentLocks` og de nye `pendingLocks`. De nye `currentLocks` for denne situasjonen er de samme minus `ROnLock` som skal delegeres. De nye `pendingLocks` for denne situasjonen er de samme, pluss den nye `WIOffLock`. Hvis `ROnLock` er den eneste låsen i `currentLocks` blir også *`delegateLock()`* kalt. Hvis det eksisterer en `WOffLock` i `pendingLocks` blir dette den eneste låsen i de nye `currentLocks`. Eventuelle `WIOffLocks` slettes og transaksjonene får beskjed om dette gjennom *`notifyAboutLostWIOffLock()`*. Dette gjelder også den nye `WIOffLock` som `ROnLock` delegeres til. Det er ingen låser i den nye `pendingLocks`. Hvis det er bare `WIOffLocks` i `pendingLocks` blir dette de nye `currentLocks`. Her blir det ingen låser i den nye `pendingLocks`. For

alle situasjonene blir metoden *addLockLogRecord()* kalt for å logge hvilken handling som ble utført.

handleReconnectedTransaction() **i** LockManager
Transaksjoner som har WOffLock på en ressurs og ikke forandrer verdien på ressursen må delegere låsen til en WIOffLock når den tilkobles. Denne metoden finner tilfeller hvor dette skjer og håndterer situasjonen. Hvis WOffLock er i currentLocks blir *delegateLock()* kalt, hvor den nye WIOffLock blir den eneste låsen i nye currentLocks. Hvis WOffLock er i pendingLocks blir *delegateLock()* kalt, hvor den nye WIOffLock blir lagt til i pendingLocks. For begge disse tilfellene bli allowMoreLocks på ressursen satt til true. For begge situasjonene blir metoden *addLockLogRecord()* kalt for å logge hvilken handling som ble utført.

7.2 Implementeringsdokumentasjon for utvidelse av online-offline låseprotokoll til en adaptiv låseprotokoll

Dette avsnittet beskriver hvordan den adaptive låseprotokollen ble implementert. I dette avnittet forklares det hvordan de viktige metodene for låseprotokollen ble implementert. Den fullstendige kildekoden er gitt i appendiks C.

7.2.1 Detaljerte metodebeskrivelser

Dette avsnittet inneholder detaljerte metodebeskrivelser som forklarer hvordan låseprotokollen ble implementert. Disse metodene er:

- *addLockLogRecord()* i LockLog.
- *handleBrowseLockRequest()* i LockManager.
- *releaseLocks()* i LockManager.
- *export()* i ExportTransaction.
- *importSharedData()* i ImportTransaction.
- *commit()* i DatabaseServer.
- *abort()* i DatabaseServer.

- *notifyAboutWOffUpgrade()* i `TransactionManager`.
- *registerForReadingBrowsedValueAgain()* i `TransactionManager`.

addLockLogRecord() **i** `LockLog`
For denne metoden i online-offline ble det beskrevet ulike handlinger som `LockManager` kunne utføre. For den adaptive låseprotokollen er disse handlingene utvidet med følgende handlinger:

- RELEASEDDELEGATEBROWSELOCKS
  Denne handlingen skjer, når en wofflås eller en wonlås frigis og det eksisterer browselåser i `pendingLocks`. Browselåsene til onlinetransaksjoner blir da delegert til ronlåser og browselåsene til offlinetransaksjoner blir delegert til wiofflåser.
- RECONNECTDELEGATE
  Denne handlingen har fått en ekstra mening. Hvis offlinetransaksjonen som holder en wofflås ikke har modifisert verdien, skal wofflåsen delegeres til en wiofflås. Eventuelle browselåser i `pendingLocks` må også delegeres til ronlåser og wiofflåser.

handleBrowseLockRequest() **i** `LockManager`
Denne metoden går gjennom alle situasjoner der en browselås kan bevilges. Hvis en slik situasjon eksisterer, bevilges låsen. Hvis det ikke eksisterer en slik situasjon, avvises låsen. Metoden returnerer `true` hvis låsen bevilges og `false` hvis den avvises. Situasjonene som sjekkes er:

- Hvis det eksisterer en `WOffLock` i `currentLocks`, kan browselåsen bevilges. Browselåsen legges da i `pendingLocks`. Det skrives en post i `LockLog` og låsen legges i samlingen over låser til transaksjonen.
- Hvis det eksiterer en `WOffLock` i `pendingLocks`, kan låsen bevilges. Browselåsen legges også da i `pendingLocks`. Det skrives en post i `LockLog` og låsen legges i samlingen over låser til transaksjonen.

export() **i** `ExportTransaction`
Denne metoden brukes for å eksportere ressurser fra en offlinetransaksjon til en mobil affiliasjon. Eksporteringstransaksjonen har informasjon om den skal bruke dele-for-skriving eller dele-for lesing. For deling-for-skriving kaller

metoden først *removeResource()* fra `LocalCacheManager`. Deretter finner den wofflåsen som offlinetransaksjonen har på ressursen. Hos wofflåsen settes det at ingen transaksjoner holder låsen lenger. Denne låsen kan nå andre offlinetransaksjoner overta. Deretter frigir metoden offlinetransaksjonen fra ansvaret å committe den modifiserte verdien på ressursen. Til slutt eksporterer den ressursen ved å kalle *shareToWrite()* i `MobileAffiliation`.

importSharedData() **i** `ImportTransaction`
Denne metoden brukes for å importere ressurser fra en mobil affiliasjon til en offlinetransaksjon. For deling-for-skriving henter importeringstransaksjonen først ut instansen av `SharedData` fra EI. Deretter legges eksporteringstransaksjonen, som eksporterte ressursen, i samlingen over eksporteringstransaksjoner som offlinetransaksjonen har tvunget-commit-ved-abort-avhengigheter med. Dette får også eksporteringstransaksjonen beskjed om. Offlinetransaksjonen tar så over wofflåsen på den delte ressursen og legger ressursen i cachet. Videre legges en ny skriveoperasjon, med den modifiserte verdien, til i samlingen over operasjoner til offlinetransaksjonen.

commit() **i** `DatabaseServer`
Denne metoden kalles når onlinetransaksjoner og offlinetransaksjoner skal committes. Denne metoden har i den adaptive låseprotokollen ansvaret for at tvunget-commit-ved-abort-avhengigheter overholdes sammen med *abort()*. Metoden har også ansvaret for at offlinetransaksjoner som importerer ressurser serialiseres etter offlinetransaksjonen og eksporteringstransaksjonen som eksporterte ressursen. I tillegg har metoden ansvaret for at leseoperasjoner som ønsker å lese modifiserte verdier igjen, gjør dette. Leseoperasjoner som ønsker å lese modifiserte verdier igjen har browset en ressurs tidligere og er registrert hos skriveoperasjonene som har modifisert verdien.

Det første som skjer er at metoden sjekker om offlinetransaksjoner og eksporteringstransaksjoner den skal serialiseres bak er committet. Hvis de ikke er det, legges commitoperasjonen på vent i en samling hos `DatabaseServer`. Denne samlingen holder på commitoperasjoner som venter på at andre transaksjoner skal committe. Hvis ikke fortsetter metoden å commitprosessen. Først committes eksporteringstransaksjoner som transaksjonen har tvunget-commit-ved-abort-avhengigheter til. Deretter committes transaksjonen selv. Når modifiserte verdier committes, sjekkes det om det er noen leseoperasjoner som ønsker å lese denne verdien. Hvis slike leseoperasjoner eksisterer, leser de den committede modifiserte verdien. Deretter committes transaksjonens egne eksporteringstransaksjoner som ikke har tvunget-commit-ved-

abort-avhengighet til noen transaksjoner. Disse eksporteringstransaksjonene har ansvaret for å committe den eksportere verdien. Denne verdien committes. Deretter committes alle importeringstransaksjonene til offlinetransaksjonen. Deretter sjekker metoden om det var noen commitoperasjoner som ventet på commit av denne transaksjonen. Hvis disse eksisterer startes commitprosessen av disse transaksjonene.

abort() **i** `DatabaseServer`
Denne metoden kalles når onlinetransaksjoner og offlinetransaksjoner skal aborteres. Denne metoden har i den adaptive låseprotokollen ansvaret for at tvunget-commit-ved-abort-avhengigheter overholdes sammen med *commit()*.

Det første metoden gjør er å abortere transaksjonen. Deretter sjekkes det om det eksisterer eksporteringstransaksjoner med tvunget-commit-ved-abort-avhengighet til denne transaksjonen. Hvis det eksisterer noen, sørger metoden for å committe disse. I tillegg sørger metoden for at disse eksporteringstransaksjonene committer verdien de har eksportert. Til slutt sjekkes det om noen andre transaksjoner venter på commit av denne transaksjonen. Hvis det eksisterer noen, startes commitprosessen av disse transaksjonene.

notifyAboutWOffUpgrade() **i** `TransactionManager`
Denne metoden kalles av `LockManager` når en wofflås oppgraderes til en wonlås og det eksisterer browselåser i `pendingLocks`. Innparameteren til metoden er en browselås. Først sjekker metoden om det er en onlinetransaksjon som holder browselåsen. Hvis det er det, så hentes instansen av onlinetransaksjonen hos `TransactionManager`. Deretter kalles metoden *registerForReadingBrowsedValueAgain()* med onlinetransaksjonen og browselåsen som innparametere. Hvis det er en offlinetransaksjon som holder browselåsen, sjekker metoden om den mobile enheten er tilkoblet. Hvis den er det, kaller den metoden *notifyAboutWOffUpgrade()* i `MobileHost`. Denne metoden kaller *registerForReadingBrowsedValueAgain* i `LocalTransactionManager`. Hvis den mobile enheten er frakobla, legges browselåsen i samlingen over inkonsistente browselåser hos ankertransaksjonen. Når den mobile enheten tilkobles sjekkes det om offlinetransaksjonene har noen inkonsistente browselåser i denne samlingen.

registerForReadingBrowsedValueAgain() **i** `TransactionManager`
Denne metoden finner skriveoperasjonen som modifiserte ressursen som er browset. Hos denne skriveoperasjonen registreres leseoperasjonen, som øns-

ker å lese denne verdien på nytt. Det er først når den modifiserte verdien committes at den leses på nytt. Dette håndterer metodene *`commit()`* og *`abort()`* i `DatabaseServer`.

8 Testing av implementeringene

En del av diplomoppgaven var å teste implementeringen av online-offline og den adaptive låseprotokollen. Formålet til testingen er å undersøke om implementeringen oppfører seg som spesifisert.

8.1 Testcase

For å teste låseprotokollene skrives det ulike testcase. Et testcase inneholder:

- en beskrivelse.
- forventet resultat.
- inndata for testcaset.
- resultat for testcaset.
- en diskusjon.

Inndata er dataen som implementeringene leser inn og kjører. Implementeringene skriver ut et resulatat, som tolkes for å se om implementasjonen er korrekt.

8.2 Testing av online-offline

Testcasene som er utført for å teste online-offline finner man i appendiks B. Det er skrevet og utført tolv testcase for online-offline. Disse testcasene er:

- testcase 1. I artikkelen om online-offline, [3], er det beskrevet to situasjoner som ønskes testet. Testcase 1 tester om implementeringen håndterer den første av disse to situasjonene korrekt.
- testcase 2. Dette testcaset tester om implementeringen håndterer den andre situasjonen, som er beskrevet i artikkelen om online-offline, korrekt.
- testcase 3-9. I online-offline er det alternerende låseoperasjoner. Dette kan enten være en oppgradering av en lås eller en delegering av en lås. Testcasene 3-9 tester ulike situasjoner som innebærer oppgradering av låser.

- testcase 10-12. Disse testcasene tester situasjoner som innebærer delegering av låser.

De tolv testcasene tester alt som skulle vært implementert for online-offline. Siden resultatet til alle testcase svarer til forventet resultat, kan det konkluderes med at implementeringen av online-offline er korrekt.

8.3 Testing av den adaptive låseprotokollen

Testcasene som er utført for å teste online-offline finner man i appendiks D. Det er skrevet og utført fire testcase for den adaptive låseprotokollen. Disse testcasene er:

- testcase 1. Dette testcaset tester om browselåser bevilges korrekt. Testcaset tester også om delegeringen av browselåser utføres korrekt og om transaksjoner som har browset en ressursen leser den modifiserte verdien når denne veriden committes.
- testcase 2. Dette testcaset tester om transaksjoner som har browset en ressurs, men som committer før den modifiserte verdien committes, faktisk committer med den umodifiserte verdien.
- testcase 3. Dette testcaset tester eksportering og importering av data mellom to offlinetransaksjoner i en mobil affiliasjon gjennom dele-forskriving.
- testcase 4. Dette testcaset tester om tvuget-commit-ved-abort-avhengigheter overholdes.

De fire testcasene tester alt som er implementer for den adaptive låseprotokollen. Siden resultatet til alle testcasene svarer til forventet resultat, kan det konkluderes med at implementeringen av den adaptive låseprotokollen er korrekt.

9 Diskusjon

I dette kapittelet diskuteres den adaptive låseprotokollen og evaluerer resultatene som er oppnådd i diplomoppgaven.

9.1 Hvordan den adaptive låseprotokollen oppnår sine mål

Målene for den adaptive låseprotokollen var:

- å støtte planlagte og uplanlagte frakoblinger.
- å støtte caching av data slik at mobile enheter kan arbeide i frakoblet modus.
- å støtte konfliktbevissthet for å sikre konsistens i databasen.
- å støtte kooperative transaksjoner
- å sikre høy effektivitet og tilgjengelighet.

Planlagte frakoblinger støttes ved at offlinetransaksjoner spør etter wiofflåser og wofflåser. Når disse bevilges, caches data lokalt på de mobile enhetene. De mobile enhetene har fasiliteter for å utføre databaseoperasjoner lokalt. Når den mobile enheten ønsker å koble fra, sier den fra til databasetjeneren og kobler fra. Den mobile enheten kan nå arbeide i frakoblet modus på de ressursene den har låser på og som er cachet. Uplanlagte frakoblinger støttes når reglene som er beskrevet i avsnitt 7.1.1 følges. Når en mobil enhet frakobles uplanlagt, kan offlinetransaksjonene arbeide med ressursene de allerede har fått låser på. Disse ressursene ligger i cachet. Ressurser som offlinetransaksjonene ikke har fått låser på, når den uplanlagte frakoblingen skjedde, vil ikke være tilgjengelig for for arbeid i frakoblet modus.

I den adaptive låseprotokollen er det tre forskjellige datacachingsmodus. Datacachingmodus uten konflikter, datacachingmodus med leseskrivekonflikt og datacachingmodus med skrivelesekonflikt. Data kan caches uten at det er potensielle konflikter. Det kan likevel oppstå låser med konflikt, hvis offlinetransaksjoner deler data i frakoblet modus. Data kan caches med leseskrivekonflikt. Wofflåser sørger for at dette støttes. Wofflåser bevilges, selv om det eksisterer ronlåser på ressursen. Offlinetransaksjonen med wofflås tar med seg

ressursen i frakoblet modus og arbeidet med den der. Den kan ikke committe verdien før transaksjonene som holder leselåsene er committet. Konsistensen er dermed sikret. Data kan caches med skrivelesekonflikter. Browselåser sørger for å støtte dette. Browselåser kan bevilges på ressurser som er wofflåst. Hvis offlinetransaksjonen med wofflåsen integrerer den nye verdien i databasen før transaksjonene med browselåser på ressursen har committet, leses den modifiserte verdien på nytt av disse transaksjonene. Disse transaksjonene serialiseres etter offlinetransaksjonen med wofflås. Hvis transaksjonene med browselåser committer før den med wofflås, committer de med den umodifiserte verdien og serialiseres før offlinetransaksjonen med wofflås.

Den adaptive låseprotokollen skal støtte konfliktbevissthet for å sikre konsistens i databasen. Låsehandlinger med konflikt blir lagret på begge mobile enheter og på begge ankertransaksjoner. Ved å gjøre dette er alle transaksjoner til en hver tid klar over potensielle konflikter.

Den adaptive låseprotokollen skal støtte kooperative transaksjoner. Dette er støttet ved å tillate offlinetransaksjoner å dele databaseelementer i mobile affiliasjoner. Dette kan gjøres med deling-for-lesing og deling-for-skriving.

Høy effektivitet og tilgjengelighet av data er støttet ved å tillate flere forskjellige låsetyper. Låser konkurrerer med andre låser. Wonlåser kan ta over wiofflåser. Wofflåser bevilges når det eksisterer ron- og wiofflåser. Browselåser bevilges på wofflåser. Offlinetransaksjoner kan arbeide i frakoblet modus. Ressursene blir først wonlåst når den virkelige operasjonen blir utført.

9.2 Forbedringer av den adaptive låseprotokollen

I dette avsnittet beskrives hva som kan forbedres med den adaptive låseprotokollen.

9.2.1 Implementering av deling-for-lesing

Deling-for-lesing er ikke implementert for den adaptive låseprotokollen. For deling-for-skriving tar offlinetransaksjonen, som importerer den delte dataverdien, over wofflåsen som holdes på ressursen. I deling-for-lesing beholder offlinetransaksjonen, som eksporterer den delte dataverdien, wofflåsen. Offlinetransaksjoner som importerer med deling-for-lesing skal bare ha leserettigheter på ressursen. Det betyr at de enten må skaffe seg en wiofflås eller en browselås for å kunne lese den nye verdien til ressursen. Siden offline-

transaksjonen som ønsker å importere ressursen er frakoblet, kan den ikke sende låseforespørsler til databasetjeneren. Det er antatt at offlinetransaksjonen som ønsker å importere den delte dataverdien ikke har nødvendig lås fra før. Hvordan man skal implementere deling-for-lesing er det ikke funnet en fornuftig løsning på og her er det muligheter for å forbedre den adaptive låseprotokollen.

En mulig løsning er å tillate offlinetransaksjonen som importerer den delte dataverdien å lese denne uten nødvendig lås. Men da må offlinetransaksjonen skaffe nødvendig lås på ressursen når den tilkobles. Et krav som kommer opp da er at ingen andre transaksjoner kan ha modifisert denne verdien fra offlinetransaksjonen som eksporterte verdien committet og til den nødvendige låsen bevilges. Hvis den nødvendige låsen bevilges og dette kravet overholdes, kan offlinetransaksjonen som importerte med deling-for-lesing committe uten å sette konsistensen til databasen i fare.

9.2.2 Initiering og avregistrering fra en mobil affiliasjon

Offlinetransaksjoner skal kunne forlate en mobil affiliasjon, når den selv ønsker. I implementeringen kan alle offlinetransaksjoner registrere seg og avregistrere seg når de selv ønsker, bortsett offlinetransaksjonen som initierer den mobile affiliasjonen. I implementeringen må offlinetransaksjonen som initierer den mobile affiliasjonen også være den siste som forlater den. Å tillate denne offlinetransaksjonen å forlate den mobile affiliasjonen når den selv ønsker vil være en forbedring.

9.3 Evaluering av resultatene i diplomoppgaven

Formålet satte opp disse målene for diplomoppgaven:

1. identifisere restriksjonene til mobile databaser.
2. undersøke eksisterende låseprotokoller for mobile databaser.
3. utrede hvordan eskisterende adaptive låseprotokoller løser problemer i mobile databaser.
4. beskrive en adaptiv låseprotokoll.
5. designe, implementere og teste online-offline.

6. designe, implementere og teste den adaptive låseprotokollen som er beskrevet.

Alle målene er oppnådd, bortsett fra målet om å undersøke hvordan eksisterende adaptive låseprotokoller løser problemer i mobile databaser. Grunnen til dette er at det ikke eksisterer noen kjente adaptive låseprotokoller for mobile databaser. I tillegg er målet om å implementere den adaptive låseprotokollen ikke fullstendig oppnådd, da deling-for-lesing ikke er implementert. Det er derimot beskrevet en mulig løsning på dette problemet.

9.3.1 Evaluering av prosessen

I starten ble det satt opp en tidsplan som satte opp når hver deloppgave skulle starte og være ferdig. I tillegg ble det satt opp en vekt for hvor mye tid som skulle brukes på hver deloppgave. Disse deloppgavene med vekt er:

- forstudie 10%
- en adaptiv låseprotokoll 35%
- diskusjon 10%
- implementasjon av online-offline 20%
- implementasjon av den adaptive låseprotokollen 20%
- dokumentasjon 5%

Det som det har blitt mer tid på enn vekten tilsier er dokumentasjon. Dokumentasjon inkluderer konstruksjonsdokumentasjon, implementasjonsdokumentasjon og testdokumentasjon. Det som det har blitt brukt mindre tid på enn vekten tilsier er den adaptive låseprotokollen og diskusjonen. Målet om å beskrive en adaptiv låseprotokoll er likevel oppnådd.

Det ble satt opp at det skulle brukes like mye tid på å implementere den adaptive låseprotokollen som online-offline. Mye av designet og grunnlaget for implementeringene ble gjort i perioden for implementering av online-offline. Den tiden ekstra tiden som ble brukt på dette i denne perioden regner jeg at også går på å designe og legge grunnlag for den adaptive låseprotokollen. Perioden for å implementere online-offline ble derfor lengre enn perioden for implementering av den adaptive låseprotokollen, men tidsbruk på låseprotokollspesifikk implementering er omtrent lik.

Starttidspunkt og avslutningstidspunkt for hver deloppgave har vært fulgt, som spesifisert i tidsplanen.

10 Konklusjon

I denne diplomoppgaven er det vist at målene til den adaptive låseprotokollen kan oppnås ved å utvide låseprotokollen online-offline. En ny låsetype ble innført i tillegg til de eksisterende låsetypene i online-offline. I tillegg ble det innført en ankertransaksjon. Den adaptive låseprotokollen støtter planlagte og uplanlagte frakoblinger. Mobile enheter kan arbeide i frakoblet modus og være bevisste på potensielle konflikter. Den adaptive låseprotokollen støtter også kooperative transaksjoner. Kooperative transaksjoner er offlinetransaksjoner som deler data i mobile affiliasjoner. Den adaptive låseprotokollen løser flere av problemene og restriksjonene som eksisterer i mobile databaser.

Referanser

[1] Joanne Holliday, Divyakant Agrawal, and Amr El Abbadi. Disconnection modes for mobile databases. *Wirel. Netw.*, 8(4):391–402, 2002.

[2] Hien Nam Le. *PhD Thesis*. PhD thesis, NTNU, 2006.

[3] Hien Nam Le, Mads Nygård, and Heri Ramampiaro. A locking model for mobile databases in mobile environments. In *Databases and Applications*, pages 49–55, 2004.

[4] Sanjay Kumar Madria and Bharat Bhargava. A transaction model to improve data availability in mobile computing. *Distrib. Parallel Databases*, 10(2):127–160, 2001.

[5] Sanjay Kumar Madria, Mukesh K. Mohania, Sourav S. Bhowmick, and Bharat K. Bhargava. Mobile data and transaction management. *Inf. Sci.*, 141(3-4):279–309, 2002.

[6] Mads Nygård and Hien Nam Le. Mobile transaction system for supporting mobile work. *16th International Workshop on Database and Expert Systems Applications (DEXA'05)*, pages 1090–1094, 2005.

[7] P. Krishna Reddy and Masaru Kitsuregawa. Speculative locking protocols to improve performance for distributed database systems. *IEEE Transactions on Knowledge and Data Engineering*, 16(2):154–169, 2004.

[8] Kenneth Salem, Hector Garcia-Molina, and Jeannie Shands. Altruistic locking. *ACM Trans. Database Syst.*, 19(1):117–165, 1994.

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

A Kildekode for online-offline

I appendiks A finner man kildekoden til online-offline.

A.1 Modulen *dataBase*

Klassene i modulen *dataBase* er:

- `DatabaseServer`
- `ConnectivityManager`
- `ConvertionLog`
- `ConvertionLogRecord`
- `LockLog`
- `LockLogRecord`
- `LockManger`
- `Log`
- `LogRecord`
- `MobileSupportStation`
- `MobileTask`
- `Resource`

A.1.1 `DatabaseServer`

```
package dataBase;

import input.InputFileReader;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import lock.Lock;
import mobileHost.LocalLogRecord;
```

```
import mobileHost.MobileHost;
import operation.AbortOperation;
import operation.CommitOperation;
import operation.Operation;
import operation.ReadOperation;
import operation.WriteOperation;
import transaction.Transaction;

public class DatabaseServer {

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="databaseSize"
   */
  private int databaseSize;

  /**
   * @uml.property name="time"
   */
  private long time;

  /**
   * @uml.property name="lockLogId"
   */
  private int lockLogId;

  /**
   * @uml.property name="mobileSupportStation"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="databaseServer:dataBase.MobileSupportStation"
   */

  private Collection<MobileSupportStation> mobileSupportStation;

  /**
   * @uml.property name="lockLog"
   * @uml.associationEnd inverse="databaseServer:dataBase.LockLog"
   */
  private LockLog lockLog;

  /**
   * @uml.property name="convertionLog"
   * @uml.associationEnd
    inverse="databaseServer:dataBase.ConvertionLog"
```

```
 */
private ConvertionLog convertionLog;

/**
 * @uml.property name="lockManager"
 * @uml.associationEnd inverse="databaseServer:dataBase.LockManager"
 */
private LockManager lockManager;

/**
 * @uml.property name="connectivityManager"
 * @uml.associationEnd
   inverse="databaseServer:dataBase.ConnectivityManager"
 */
private ConnectivityManager connectivityManager;

/**
 * @uml.property name="mobileTask"
 * @uml.associationEnd multiplicity="(0 -1)"
 *                     inverse="databaseServer:dataBase.MobileTask"
 */

private Collection<MobileTask> mobileTask;

/**
 * @uml.property name="resource"
 * @uml.associationEnd multiplicity="(0 -1)"
 *                     inverse="databaseServer:dataBase.Resource"
 */
private Collection<Resource> resource;

/**
 * @uml.property name="log"
 * @uml.associationEnd inverse="databaseServer:dataBase.Log"
 */
private Log log;

/**
 * @uml.property name="inputFileReader"
 */
private InputFileReader inputFileReader;

/**
 * @uml.property name="nanoStartTime"
 */
private long nanoStartTime;

/**
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
```

```
 */
private Collection<Lock> lostLocksWhenMobileHostIsDisconnected;

/**
 * @uml.property name="gainedLocksWhenMobileHostIsDisconnected"
 */
private Collection<Lock> gainedLocksWhenMobileHostIsDisconnected;

/**
 * @uml.property name="activeTransactions"
 */
private Collection<Transaction> activeTransactions;

public DatabaseServer(String id, int databaseSize, long time) {
  this.id = id;
  this.databaseSize = databaseSize;
  this.time = time;
  this.nanoStartTime = System.nanoTime();
  lockLogId = 0;
  resource = new ArrayList<Resource>();
  mobileSupportStation = new ArrayList<MobileSupportStation>();
  mobileTask = new ArrayList<MobileTask>();
  lockManager = new LockManager(this);
  lockLog = new LockLog(this);
  convertionLog = new ConvertionLog(this);
  connectivityManager = new ConnectivityManager(this);
  log = new Log(this);
  activeTransactions = new ArrayList<Transaction>();
  lostLocksWhenMobileHostIsDisconnected = new ArrayList<Lock>();
  gainedLocksWhenMobileHostIsDisconnected = new ArrayList<Lock>();
}

/**
 */
public int read(ReadOperation readOperation) {
  Transaction transaction = readOperation.getTransaction();
  Resource resource = readOperation.getResource();
  int readValue;
  if (transaction.isResourceWritten(resource)) {
    readValue = transaction.getNewestWrittenValue(resource,
        readOperation);
  } else {
    readValue = getResource(resource.getId()).getValue();
  }
  readOperation.setReadValue(readValue);
  log.addLogRecord(transaction.getId(), "READ", resource.getId(),
      new String("" + readValue), "");
  return readValue;
}
```

```
/**
 */
public void write(WriteOperation writeOperation) {
  Transaction transaction = writeOperation.getTransaction();
  Resource resource = writeOperation.getResource();
  int newValue = writeOperation.getNewValue();
  transaction.getWrittenResources().add(resource);
  log.addLogRecord(transaction.getId(), "WRITE", resource.getId(),
      new String("" + resource.getValue()), new String(""
          + newValue));
}

/**
 */
public void commit(CommitOperation commitOperation) {
  Transaction transaction = commitOperation.getTransaction();
  activeTransactions.remove(transaction);
  Collection<Operation> operations = transaction.getOperation();
  for (Iterator<Operation> i = operations.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      wo.getResource().setValue(wo.getNewValue());
    }
  }
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    transaction.getOwner().getLocalCacheManager().removeResource(
        lock.getResource(), transaction);
  }
  transaction.setStatus("COMMITED");

  getLockManager().releaseLocks(transaction);
  log.addLogRecord(transaction.getId(), "COMMIT", "", "", "");
}

/**
 */
public void abort(AbortOperation abortOperation) {
  Transaction transaction = abortOperation.getTransaction();
  activeTransactions.remove(transaction);
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    transaction.getOwner().getLocalCacheManager().removeResource(
        lock.getResource(), transaction);
  }
```

```
    transaction.setStatus("ABORTED");

    getLockManager().releaseLocks(transaction);
    log.addLogRecord(transaction.getId(), "ABORT", "", "", "");
  }

  /**
   */
  public boolean requestROnLock(Resource resource,
      Transaction transaction) {
    return lockManager.handleROnLockRequest(resource, transaction);
  }

  /**
   */
  public boolean requestWOnLock(Resource resource,
      Transaction transaction) {
    return lockManager.handleWOnLockRequest(resource, transaction);
  }

  /**
   */
  public boolean requestWIOffLock(Resource resource,
      Transaction transaction) {
    return lockManager.handleWIOffLockRequest(resource, transaction);
  }

  /**
   */
  public boolean requestWOffLock(Resource resource,
      Transaction transaction) {
    return lockManager.handleWOffLockRequest(resource, transaction);
  }

  /**
   */
  public void reconnect(MobileHost mobileHost) {
    getConnectivityManager().getOfflineMobileHosts().remove(
        mobileHost);
    getConnectivityManager().getOnlineMobileHosts().add(mobileHost);

    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1 =
      reconnectCheck1(mobileHost);
    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2 =
      reconnectCheck2(mobileHost, localLogRecordsWhichNotPassesCheck1);
    reconnectCheck3(mobileHost, localLogRecordsWhichNotPassesCheck2);

    Collection<Lock> lostLocks = new ArrayList<Lock>();
    for (Iterator<Lock> i = lostLocksWhenMobileHostIsDisconnected
```

```
    .iterator(); i.hasNext();) {
  Lock lock = i.next();
  if (lock.getTransaction().getOwner().getId().equals(
      mobileHost.getId())) {
    lostLocks.add(lock);
  }
}
for (Iterator<Lock> i = lostLocks.iterator(); i.hasNext();) {
  Lock lock = i.next();
  lock.getTransaction().getLocks().remove(lock);
  mobileHost.getLocalCacheManager().removeResource(
      lock.getResource(), lock.getTransaction());
  lostLocksWhenMobileHostIsDisconnected.remove(lock);
}
Collection<Lock> gainedLocks = new ArrayList<Lock>();
for (Iterator<Lock> i = gainedLocksWhenMobileHostIsDisconnected
    .iterator(); i.hasNext();) {
  Lock lock = i.next();
  if (lock.getTransaction().getOwner().getId().equals(
      mobileHost.getId())) {
    gainedLocks.add(lock);
  }
}
for (Iterator<Lock> i = gainedLocks.iterator(); i.hasNext();) {
  Lock lock = i.next();
  lock.getTransaction().getLocks().add(lock);
  mobileHost.getLocalCacheManager().addResource(
      lock.getResource(), lock.getTransaction());
  if (lock.getLockType().equals("WIOFFLOCK")) {
    long timeWhenLockWasGranted = getLockLog()
        .getTimeWhenLockWasGranted(lock);
    mobileHost.getLocalLockLog().addLocalLockLogRecord(
        timeWhenLockWasGranted, lock.getTransaction().getId(),
        lock.getResource().getId(), "WIOFFLOCK");
  }
  gainedLocksWhenMobileHostIsDisconnected.remove(lock);
}

reconnectCheckForUnplannedDisconnection
  WhoHasLostLockWhenDisconnected(mobileHost);

Collection<LocalLogRecord> localLogRecordsForStartTransactionCheck
    = mobileHost.
      getLocalLog().getLocalLogRecordsFromLastDisconnection(
        mobileHost.getLastDisconnectionTime());
for (Iterator<LocalLogRecord> i =
    localLogRecordsForStartTransactionCheck.iterator();
      i.hasNext();) {
  LocalLogRecord localLogRecord = i.next();
```

```
    if (localLogRecord.getAction().equals("STARTOFFLINE")) {
      initiateTransaction(mobileHost.getLocalTransactionManager()
          .getTransaction(localLogRecord.getTransactionId()));
    }
    if (localLogRecord.getAction().equals("COMMITOFFLINE")) {
      commit((CommitOperation) mobileHost
          .getLocalTransactionManager().getTransaction(
              localLogRecord.getTransactionId()).getOperation(
              "COMMITOPERATION"));
    }
    if (localLogRecord.getAction().equals("ABORTOFFLINE")) {
      abort((AbortOperation) mobileHost
          .getLocalTransactionManager().getTransaction(
              localLogRecord.getTransactionId()).getOperation(
              "ABORTOPERATION"));
    }
  }
  for (Iterator<Transaction> i = mobileHost
      .getLocalTransactionManager().getTransaction().iterator(); i
      .hasNext();) {
    Transaction transaction = i.next();
    getLockManager().handleReconnectedTransaction(transaction);
  }
  System.out.println("Mobilehost " + mobileHost.getId()
      + " reconnects at time " + getTime());

}

private Collection<LocalLogRecord> reconnectCheck1(MobileHost mh) {
  Collection<LocalLogRecord> operationsWithPossibleConflict = new
    ArrayList<LocalLogRecord>();
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1 =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LocalLogRecord> localLogRecords = mh.getLocalLog()
      .getLocalLogRecordsFromLastDisconnection(
          lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    if (localLogRecord.getAction().equals("READOFFLINE")) {
      Transaction t = mh.getLocalTransactionManager()
          .getTransaction(localLogRecord.getTransactionId());
      Collection<Lock> locks = t.getLocks();

      for (Iterator<Lock> j = locks.iterator(); j.hasNext();) {
        Lock lock = j.next();
        if (lock.getResource().getId().equals(
            localLogRecord.getResourceId())) {
```

```
          if (lock.getLockType().equals("WIOFFLOCK")
              || lock.getLockType().equals("RONLOCK")) {
            operationsWithPossibleConflict.add(localLogRecord);
            System.out
                .println("Found operation with possible conflict "
                    + localLogRecord.getTransactionId() + " "
                    + localLogRecord.getResourceId());
          }
        }
      }
    }
  }
  Collection<LockLogRecord> lockLogRecords = getLockLog()
      .getLockLogRecordsFromTime(lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = operationsWithPossibleConflict
      .iterator(); i.hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    for (Iterator<LockLogRecord> j = lockLogRecords.iterator(); j
        .hasNext();) {
      LockLogRecord lockLogRecord = j.next();
      if (localLogRecord.getResourceId().equals(
          lockLogRecord.getResourceId())
          && (lockLogRecord.getRequstingLockType().equals(
              "WOFFLOCK") || lockLogRecord.getRequstingLockType()
              .equals("WONLOCK"))
          && (new Long(localLogRecord.getLocalTime()) > new Long(
              lockLogRecord.getTime()))) {
        if (!localLogRecordsWhichNotPassesCheck1
            .contains(localLogRecord)) {
          localLogRecordsWhichNotPassesCheck1.add(localLogRecord);
          System.out
              .println("Found operation which not passes check 1: "
                  + localLogRecord.getTransactionId()
                  + " "
                  + localLogRecord.getResourceId());
        }
      }
    }
  }
  return localLogRecordsWhichNotPassesCheck1;
}

private Collection<LocalLogRecord> reconnectCheck2(MobileHost mh,
    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1) {
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2 =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LogRecord> logRecords = getLog()
      .getLogRecordsFromTime(lastDisconnectionTime);
```

```
  for (Iterator<LocalLogRecord> i = localLogRecordsWhichNotPassesCheck1
      .iterator(); i.hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    for (Iterator<LogRecord> j = logRecords.iterator(); j
        .hasNext();) {
      LogRecord logRecord = j.next();
      if (localLogRecord.getResourceId().equals(
          logRecord.getResourceId())
          && logRecord.getAction().equals("WRITE")
          && (new Long(localLogRecord.getLocalTime()) > new Long(
              logRecord.getTime()))) {
        localLogRecordsWhichNotPassesCheck2.add(localLogRecord);
        System.out
            .println("Found operation which not passes check 2: "
                + localLogRecord.getTransactionId() + " "
                + localLogRecord.getResourceId());
      }
    }
  }
  localLogRecordsWhichNotPassesCheck1
      .removeAll(localLogRecordsWhichNotPassesCheck2);
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck1);
  return localLogRecordsWhichNotPassesCheck2;
}

private void reconnectCheck3(MobileHost mh,
    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2) {
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck2);
}

private void
  reconnectCheckForUnplannedDisconnection
    WhoHasLostLockWhenDisconnected(MobileHost mh) {
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LocalLogRecord> localLogRecords = mh.getLocalLog()
      .getLocalLogRecordsFromLastDisconnection(
          lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    if (localLogRecord.getAction().equals("READOFFLINE")) {
      Transaction t = mh.getLocalTransactionManager()
          .getTransaction(localLogRecord.getTransactionId());
      if (t.getStatus().equals("ACTIVE")) {
        Collection<Lock> locks = t.getLocks();
        boolean lostLock = true;
        for (Iterator<Lock> j = locks.iterator(); j.hasNext();) {
```

```
          Lock lock = j.next();
          if (lock.getResource().getId().equals(
              localLogRecord.getResourceId())) {
            if (lock.getLockType().equals("WIOFFLOCK")) {
              lostLock = false;
            }
          }
        }
        if (lostLock) {
          localLogRecordsWhichNotPassesCheck.add(localLogRecord);
          System.out.println("Found operation with lost lock "
              + localLogRecord.getTransactionId() + " "
              + localLogRecord.getResourceId());
        }
      }

    }
  }
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck);
}

/**
 */
public void rectifyOperations(MobileHost mh,
    Collection<LocalLogRecord> localLogRecords) {
  Collection<Transaction> t = new ArrayList<Transaction>();
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    t.add(mh.getLocalTransactionManager().getTransaction(
        localLogRecord.getTransactionId()));
  }
  for (Iterator<Transaction> i = t.iterator(); i.hasNext();) {
    Transaction transaction = i.next();
    mh.getLocalTransactionManager().executeAbortOperation(
        transaction.getId());
  }
}

/**
 */
public void initiatePlannedDisconnection(MobileHost mobileHost) {
  for (Iterator<Transaction> i = mobileHost
      .getLocalTransactionManager().getTransaction().iterator(); i
      .hasNext();) {
    Transaction transaction = i.next();
    getLockManager().handleDisconnectedTransaction(transaction,
        true);
  }
```

```
  getConnectivityManager().getOnlineMobileHosts().remove(
      mobileHost);
  getConnectivityManager().getOfflineMobileHosts().add(mobileHost);
  System.out.println("Mobilehost " + mobileHost.getId()
      + " disconnects planned at time " + getTime());

}

public void handleUnplannedDisconnection(MobileHost mobileHost) {
  for (Iterator<Transaction> i = getActiveTransactions()
      .iterator(); i.hasNext();) {
    Transaction transaction = i.next();
    if (transaction.getOwner().getId().equals(mobileHost.getId())) {
      getLockManager().handleDisconnectedTransaction(transaction,
          false);
    }
  }
  System.out.println("DB found disconnected mobilehost "
      + mobileHost.getId() + " at time " + getTime());
}

/**
 */
public void initiateTransaction(Transaction transaction) {
  activeTransactions.add(transaction);
  log.addLogRecord(transaction.getId(), "START", "", "", "");
}

/**
 */
public void notifyAboutLostWIOffLock(Lock lostLock) {
  if (lostLock.getTransaction().getOwner().getConnectionStatus()) {
    lostLock.getTransaction().getOwner().notifyAboutLostWIOffLock(
        lostLock);
  } else {
    lostLocksWhenMobileHostIsDisconnected.add(lostLock);
  }
}

/**
 */
public Resource getResource(String id) {
  for (Iterator<Resource> i = resource.iterator(); i.hasNext();) {
    Resource r = i.next();
    if (r.getId().equals(id)) {
      return r;
    }
  }
  return null;
```

```
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>time</tt>
 *
 * @return Returns the time.
 * @uml.property name="time"
 */
public long getTime() {
  time = System.nanoTime() - nanoStartTime;
  return time;
}

/**
 * Setter of the property <tt>time</tt>
 *
 * @param time
 *          The time to set.
 * @uml.property name="time"
 */
public void setTime(long time) {
  this.time = time;
}

/**
 * Getter of the property <tt>databaseSize</tt>
 *
 * @return Returns the databaseSize.
```

```
 * @uml.property name="databaseSize"
 */
public int getDatabaseSize() {
  return databaseSize;
}

/**
 * Setter of the property <tt>databaseSize</tt>
 *
 * @param databaseSize
 *          The databaseSize to set.
 * @uml.property name="databaseSize"
 */
public void setDatabaseSize(int databaseSize) {
  this.databaseSize = databaseSize;
}

/**
 * Getter of the property <tt>lockLogId</tt>
 *
 * @return Returns the lockLogId.
 * @uml.property name="lockLogId"
 */
public int getLockLogId() {
  return lockLogId;
}

/**
 * Setter of the property <tt>lockLogId</tt>
 *
 * @param lockLogId
 *          The lockLogId to set.
 * @uml.property name="lockLoggId"
 */
public void setLockLogId(int lockLogId) {
  this.lockLogId = lockLogId;
}

/**
 * Getter of the property <tt>mobileSupportStation</tt>
 *
 * @return Returns the mobileSupportStation.
 * @uml.property name="mobileSupportStation"
 */
public Collection<MobileSupportStation> getMobileSupportStation() {
  return mobileSupportStation;
}

/**
```

```
 * Setter of the property <tt>mobileSupportStation</tt>
 *
 * @param mobileSupportStation
 *          The mobileSupportStation to set.
 * @uml.property name="mobileSupportStation"
 */
public void setMobileSupportStation(
    Collection<MobileSupportStation> mobileSupportStation) {
  this.mobileSupportStation = mobileSupportStation;
}

/**
 * Getter of the property <tt>lockLog</tt>
 *
 * @return Returns the lockLog.
 * @uml.property name="lockLog"
 */
public LockLog getLockLog() {
  return lockLog;
}

/**
 * Setter of the property <tt>lockLog</tt>
 *
 * @param lockLog
 *          The lockLog to set.
 * @uml.property name="lockLog"
 */
public void setLockLog(LockLog lockLog) {
  this.lockLog = lockLog;
}

/**
 * Getter of the property <tt>convertionLog</tt>
 *
 * @return Returns the convertionLog.
 * @uml.property name="convertionLog"
 */
public ConvertionLog getConvertionLog() {
  return convertionLog;
}

/**
 * Setter of the property <tt>convertionLog</tt>
 *
 * @param convertionLog
 *          The convertionLog to set.
 * @uml.property name="convertionLog"
 */
```

```
public void setConvertionLog(ConvertionLog convertionLog) {
  this.convertionLog = convertionLog;
}

/**
 * Getter of the property <tt>lockManager</tt>
 *
 * @return Returns the lockManager.
 * @uml.property name="lockManager"
 */
public LockManager getLockManager() {
  return lockManager;
}

/**
 * Setter of the property <tt>lockManager</tt>
 *
 * @param lockManager
 *          The lockManager to set.
 * @uml.property name="lockManager"
 */
public void setLockManager(LockManager lockManager) {
  this.lockManager = lockManager;
}

/**
 * Getter of the property <tt>connectivityManager</tt>
 *
 * @return Returns the connectivityManager.
 * @uml.property name="connectivityManager"
 */
public ConnectivityManager getConnectivityManager() {
  return connectivityManager;
}

/**
 * Setter of the property <tt>connectivityManager</tt>
 *
 * @param connectivityManager
 *          The connectivityManager to set.
 * @uml.property name="connectivityManager"
 */
public void setConnectivityManager(
    ConnectivityManager connectivityManager) {
  this.connectivityManager = connectivityManager;
}

/**
 * Getter of the property <tt>mobileTask</tt>
```

```
 *
 * @return Returns the mobileTask.
 * @uml.property name="mobileTask"
 */
public Collection<MobileTask> getMobileTask() {
  return mobileTask;
}

/**
 * Setter of the property <tt>mobileTask</tt>
 *
 * @param mobileTask
 *          The mobileTask to set.
 * @uml.property name="mobileTask"
 */
public void setMobileTask(Collection<MobileTask> mobileTask) {
  this.mobileTask = mobileTask;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Collection<Resource> getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Collection<Resource> resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>input</tt>
 *
 * @return Returns the input.
 * @uml.property name="inputFileReader"
 */
public InputFileReader getInputFileReader() {
  return inputFileReader;
}
```

```
/**
 * Setter of the property <tt>input</tt>
 *
 * @param input
 *          The input to set.
 * @uml.property name="inputFileReader"
 */
public void setInputFileReader(InputFileReader inputFileReader) {
  this.inputFileReader = inputFileReader;
}

/**
 * Getter of the property <tt>log</tt>
 *
 * @return Returns the log.
 * @uml.property name="log"
 */
public Log getLog() {
  return log;
}

/**
 * Setter of the property <tt>log</tt>
 *
 * @param log
 *          The log to set.
 * @uml.property name="log"
 */
public void setLog(Log log) {
  this.log = log;
}

@SuppressWarnings("unused")
private void readInputFile(String inputFile) {
  if (inputFile != null) {
    inputFileReader = new InputFileReader(inputFile);
    String line = inputFileReader.readLine();
    if (line == null || !line.startsWith("CREATE_SERVER")) {
      unexpectedInput();
    } else {
      setId(line.substring(14));
      System.out.println("A new server is created with id "
          + line.substring(14));
      line = inputFileReader.readLine();
      if (line == null || !line.startsWith("NOF_RESOURCES")) {
        unexpectedInput();
      } else {
        int numberOfResources = new Integer(line.substring(14));
```

```
      setDatabaseSize(numberOfResources);
      createResources(numberOfResources);
    }
    line = inputFileReader.readLine();
    int numberOfMobileTasks = 0;
    if (line == null || !line.startsWith("NOF_MOBILETASKS")) {
      unexpectedInput();
    } else {
      numberOfMobileTasks = new Integer(line.substring(16));
      for (int i = 0; i < numberOfMobileTasks; i++) {
        line = inputFileReader.readLine();
        if (line == null
            || !line.startsWith("CREATE_MOBILETASK")) {
          unexpectedInput();
        } else {
          MobileTask mt = new MobileTask(line.substring(18),
              this);
          getMobileTask().add(mt);
          line = inputFileReader.readLine();
          int numberOfMobileTaskResources = 0;
          if (line == null || !line.startsWith("NOF_RESOURCES")) {
            unexpectedInput();
          } else {
            numberOfMobileTaskResources = new Integer(line
                .substring(14));
            for (int j = 0; j < numberOfMobileTaskResources; j++) {
              line = inputFileReader.readLine();
              if (line == null || !line.startsWith("RESOURCE")) {
                unexpectedInput();
              } else {
                mt.getResource().add(
                    getResource(line.substring(9)));
              }
            }
          }
        }
      }
    }
  }
}
line = inputFileReader.readLine();
if (line == null
    || !line.startsWith("NOF_MOBILESUPPORTSTATIONS")) {
  unexpectedInput();
} else {
  int numberOfMSS = new Integer(line.substring(26));
  for (int i = 0; i < numberOfMSS; i++) {
    line = inputFileReader.readLine();
    if (line == null
        || !line.startsWith("CREATE_MOBILESUPPORTSTATION")) {
```

```
        unexpectedInput();
      } else {
        String subString = line.substring(28);
        String[] params = subString.split(":");
        MobileSupportStation mss = new MobileSupportStation(
            params[0], this);
        getMobileSupportStation().add(mss);
        System.out
            .println("A new mobile support station is created with
              id "+ params[0]);
      }
    }

  }
  line = inputFileReader.readLine();
  if (line == null || !line.startsWith("NOF_MOBILEHOSTS")) {
    unexpectedInput();
  } else {
    int numberOfMH = new Integer(line.substring(16));
    for (int i = 0; i < numberOfMH; i++) {
      line = inputFileReader.readLine();
      if (line == null || !line.startsWith("CREATE_MOBILEHOST")) {
        unexpectedInput();
      } else {
        String subString = line.substring(18);
        String[] params = subString.split(":");
        MobileHost mh = new MobileHost(params[0],
            getMobileSupportStation(params[1]), new Integer(
                params[2]), new Integer(params[3]));
        getMobileSupportStation(params[1]).register(mh);
        System.out
            .println("A new mobile host is created with id "
                + params[0]);
      }
    }
  }
  line = inputFileReader.readLine();
  while (!(line == null || line.startsWith("END_FILE"))) {
    if (line.startsWith("START_TRANSACTION")) {
      String subString = line.substring(18);
      String[] params = subString.split(":");

      MobileHost mh = getMobileHost(params[0]);
      mh.startNewTransaction(params[1]);
    }
    if (line.startsWith("READ")) {
      String subString = line.substring(5);
      String[] params = subString.split(":");
```

```
  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeReadOperation(
      params[1], params[2]);
}
if (line.startsWith("WRITE")) {
  String subString = line.substring(6);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeWriteOperation(
      params[1], params[2], new Integer(params[3]));
}
if (line.startsWith("COMMIT_TRANSACTION")) {
  String subString = line.substring(19);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeCommitOperation(
      params[1]);
}
if (line.startsWith("ABORT_TRANSACTION")) {
  String subString = line.substring(18);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeAbortOperation(
      params[1]);
}
if (line.startsWith("REQUEST_RONLOCK")) {
  String subString = line.substring(16);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().requestROnLock(params[1],
      params[2]);
}
if (line.startsWith("REQUEST_WONLOCK")) {
  String subString = line.substring(16);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().requestWOnLock(params[1],
      params[2]);
}
if (line.startsWith("REQUEST_WIOFFLOCK")) {
  String subString = line.substring(18);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
```

```
        mh.getLocalTransactionManager().requestWIOffLock(
            params[1], params[2]);
      }
      if (line.startsWith("REQUEST_WOFFLOCK")) {
        String subString = line.substring(17);
        String[] params = subString.split(":");

        MobileHost mh = getMobileHost(params[0]);
        mh.getLocalTransactionManager().requestWOffLock(params[1],
            params[2]);
      }
      if (line.startsWith("DISCONNECT_PLANNED")) {
        String subString = line.substring(19);
        MobileHost mh = getMobileHost(subString);
        mh.disconnectPlanned();
      }
      if (line.startsWith("DISCONNECT_UNPLANNED")) {
        String subString = line.substring(21);
        MobileHost mh = getMobileHost(subString);
        mh.disconnectUnplanned();
      }
      if (line.startsWith("RECONNECT")) {
        String subString = line.substring(10);
        MobileHost mh = getMobileHost(subString);
        mh.reconnect();
      }
      line = inputFileReader.readLine();
    }
    inputFileReader.close();
  }

}

/**
 */
private void unexpectedInput() {
  System.out
    .println("ERROR: Input file is incorrectly formatted. Program
      aborted.");
  System.exit(1);
}

/**
 */
private void addResource(Resource resource) {
  this.resource.add(resource);
}

private MobileSupportStation getMobileSupportStation(String id) {
```

```
  for (Iterator<MobileSupportStation> i = mobileSupportStation
      .iterator(); i.hasNext();) {
    MobileSupportStation mss = i.next();
    if (mss.getId().equals(id)) {
      return mss;
    }
  }
  return null;
}

private MobileHost getMobileHost(String id) {
  for (Iterator<MobileSupportStation> i = mobileSupportStation
      .iterator(); i.hasNext();) {
    MobileSupportStation mss = i.next();
    for (Iterator<MobileHost> j = mss.getMobileHost().iterator(); j
        .hasNext();) {
      MobileHost mh = j.next();
      if (mh.getId().equals(id)) {
        return mh;
      }
    }
  }
  return null;
}

private void createResources(int numberOfResources) {
  double maxValue = 10;
  for (int i = 0; i < numberOfResources; i++) {
    int resourceValue = new Double(Math.random() * (maxValue + 1))
        .intValue();
    System.out.println("A new resource is created with id " + i
        + " and value " + resourceValue);
    addResource(new Resource(this, resourceValue, "" + i));
  }
}

/**
 * @param args
 */
public static void main(String[] args) {
  DatabaseServer db = new DatabaseServer("0", 0, 0);
  db.readInputFile("../input/testcase12.txt");
  db.getLockLog().printLockLog();
  db.getLog().printLog();
  for (Iterator<MobileHost> i = db.getMobileSupportStation("0")
      .getMobileHost().iterator(); i.hasNext();) {
    MobileHost mh = i.next();
    mh.getLocalLockLog().printLocalLockLog();
    mh.getLocalLog().printLocalLog();
```

```
  }
  db.getConnectivityManager().setTerminate(true);
}

public long getNanoStartTime() {
  return nanoStartTime;
}

/**
 * Getter of the property <tt>lostLocksWhenMobileHostIsDisconnected</tt>
 *
 * @return Returns the lostLocksWhenMobileHostIsDisconnected.
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
 */
public Collection<Lock> getLostLocksWhenMobileHostIsDisconnected() {
  return lostLocksWhenMobileHostIsDisconnected;
}

/**
 * Setter of the property <tt>lostLocksWhenMobileHostIsDisconnected</tt>
 *
 * @param lostLocksWhenMobileHostIsDisconnected
 *          The lostLocksWhenMobileHostIsDisconnected to set.
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
 */
public void setLostLocksWhenMobileHostIsDisconnected(
    Collection<Lock> lostLocksWhenMobileHostIsDisconnected) {
  this.lostLocksWhenMobileHostIsDisconnected =
    lostLocksWhenMobileHostIsDisconnected;
}

/**
 * Getter of the property <tt>activeTransactions</tt>
 *
 * @return Returns the activeTransactions.
 * @uml.property name="activeTransactions"
 */
public Collection<Transaction> getActiveTransactions() {
  return activeTransactions;
}

/**
 * Setter of the property <tt>activeTransactions</tt>
 *
 * @param activeTransactions
 *          The activeTransactions to set.
 * @uml.property name="activeTransactions"
 */
public void setActiveTransactions(
```

```
      Collection<Transaction> activeTransactions) {
    this.activeTransactions = activeTransactions;
  }

  public Collection<Lock> getGainedLocksWhenMobileHostIsDisconnected() {
    return gainedLocksWhenMobileHostIsDisconnected;
  }

  public void setGainedLocksWhenMobileHostIsDisconnected(
      Collection<Lock> gainedLocksWhenMobileHostIsDisconnected) {
    this.gainedLocksWhenMobileHostIsDisconnected =
      gainedLocksWhenMobileHostIsDisconnected;
  }
}
```

A.1.2 ConnectivityManager

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import mobileHost.MobileHost;

public class ConnectivityManager implements Runnable {

  /**
   * @uml.property name="onlineMobileHosts"
   */
  private Collection<MobileHost> onlineMobileHosts;

  /**
   * @uml.property name="offlineMobileHosts"
   */
  private Collection<MobileHost> offlineMobileHosts;

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd
     inverse="connectivityManager:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  private Thread thread;

  private boolean terminate = false;
```

```
public ConnectivityManager(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
  onlineMobileHosts = new ArrayList<MobileHost>();
  offlineMobileHosts = new ArrayList<MobileHost>();
  thread = new Thread(this);
  thread.start();
}

/**
 */
public void checkConnectionStatus() {
  ArrayList<MobileHost> disconnectedMobileHosts = new
    ArrayList<MobileHost>();
  for (Iterator<MobileHost> i = onlineMobileHosts.iterator(); i
      .hasNext();) {
    MobileHost mobileHost = i.next();
    if (mobileHost.getConnectionStatus() == false) {
      disconnectedMobileHosts.add(mobileHost);
    }
  }
  for (Iterator<MobileHost> i = disconnectedMobileHosts.iterator();
    i.hasNext();) {
    MobileHost mobileHost = i.next();
    databaseServer.handleUnplannedDisconnection(mobileHost);
    onlineMobileHosts.remove(mobileHost);
    offlineMobileHosts.add(mobileHost);
  }
  try {
    Thread.sleep(new Integer(240).longValue());
  } catch (InterruptedException e) {
    e.printStackTrace();
  }
}

public void run() {
  while (!terminate) {
    checkConnectionStatus();
  }
}

/**
 * Getter of the property <tt>onlineMobileHosts</tt>
 *
 * @return Returns the onlineMobileHosts.
 * @uml.property name="onlineMobileHosts"
 */
public Collection<MobileHost> getOnlineMobileHosts() {
  return onlineMobileHosts;
```

```
}

/**
 * Setter of the property <tt>onlineMobileHosts</tt>
 *
 * @param onlineMobileHosts
 *          The onlineMobileHosts to set.
 * @uml.property name="onlineMobileHosts"
 */
public void setOnlineMobileHosts(
    Collection<MobileHost> onlineMobileHosts) {
  this.onlineMobileHosts = onlineMobileHosts;
}

/**
 * Getter of the property <tt>offlineMobileHosts</tt>
 *
 * @return Returns the offlineMobileHosts.
 * @uml.property name="offlineMobileHosts"
 */
public Collection<MobileHost> getOfflineMobileHosts() {
  return offlineMobileHosts;
}

/**
 * Setter of the property <tt>offlineMobileHosts</tt>
 *
 * @param offlineMobileHosts
 *          The offlineMobileHosts to set.
 * @uml.property name="offlineMobileHosts"
 */
public void setOfflineMobileHosts(
    Collection<MobileHost> offlineMobileHosts) {
  this.offlineMobileHosts = offlineMobileHosts;
}

public Thread getThread() {
  return thread;
}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}
```

```
  /**
   * Setter of the property <tt>databaseServer</tt>
   *
   * @param databaseServer
   *          The databaseServer to set.
   * @uml.property name="databaseServer"
   */
  public void setDatabaseServer(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  public void setTerminate(boolean terminate) {
    this.terminate = terminate;
  }
}
```

A.1.3 ConvertionLog

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;

public class ConvertionLog {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd
    inverse="convertionLog:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  /**
   * @uml.property name="convertionLogRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
     inverse="convertionLog:dataBase.ConvertionLogRecord"
   */
  private Collection<ConvertionLogRecord> convertionLogRecord;

  public ConvertionLog(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
    convertionLogRecord = new ArrayList<ConvertionLogRecord>();
  }

  /**
```

```
 */
public void addConvertionLogRecord(String id,
    String convertionLockType) {
  convertionLogRecord.add(new ConvertionLogRecord(this, id,
      convertionLockType));

}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}

/**
 * Setter of the property <tt>databaseServer</tt>
 *
 * @param databaseServer
 *          The databaseServer to set.
 * @uml.property name="databaseServer"
 */
public void setDatabaseServer(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
}

/**
 * Getter of the property <tt>convertionLogRecord</tt>
 *
 * @return Returns the convertionLogRecord.
 * @uml.property name="convertionLogRecord"
 */
public Collection<ConvertionLogRecord> getConvertionLogRecord() {
  return convertionLogRecord;
}

/**
 * Setter of the property <tt>convertionLogRecord</tt>
 *
 * @param convertionLogRecord
 *          The convertionLogRecord to set.
 * @uml.property name="convertionLogRecord"
 */
public void setConvertionLogRecord(
    Collection<ConvertionLogRecord> convertionLogRecord) {
  this.convertionLogRecord = convertionLogRecord;
```

```
  }
}
```

A.1.4 ConvertionLogRecord

```
package dataBase;

public class ConvertionLogRecord {

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="convertionLocktype"
   */
  private String convertionLocktype;

  /**
   * @uml.property name="convertionLog"
   * @uml.associationEnd
    inverse="convertionLogRecord:dataBase.ConvertionLog"
   */
  private ConvertionLog convertionLog;

  public ConvertionLogRecord(ConvertionLog convertionLog, String id,
      String convertionLockType) {
    this.convertionLog = convertionLog;
    this.id = id;
    this.convertionLocktype = convertionLockType;
  }

  /**
   * Getter of the property <tt>id</tt>
   *
   * @return Returns the id.
   * @uml.property name="id"
   */
  public String getId() {
    return id;
  }

  /**
   * Setter of the property <tt>id</tt>
   *
   * @param id
```

```
   *          The id to set.
   * @uml.property name="id"
   */
  public void setId(String id) {
    this.id = id;
  }

  /**
   * Getter of the property <tt>convertionLocktype</tt>
   *
   * @return Returns the convertionLocktype.
   * @uml.property name="convertionLocktype"
   */
  public String getConvertionLocktype() {
    return convertionLocktype;
  }

  /**
   * Setter of the property <tt>convertionLocktype</tt>
   *
   * @param convertionLocktype
   *          The convertionLocktype to set.
   * @uml.property name="convertionLocktype"
   */
  public void setConvertionLocktype(String convertionLocktype) {
    this.convertionLocktype = convertionLocktype;
  }

  /**
   * Getter of the property <tt>convertionLog</tt>
   *
   * @return Returns the convertionLog.
   * @uml.property name="convertionLog"
   */
  public ConvertionLog getConvertionLog() {
    return convertionLog;
  }

  /**
   * Setter of the property <tt>convertionLog</tt>
   *
   * @param convertionLog
   *          The convertionLog to set.
   * @uml.property name="convertionLog"
   */
  public void setConvertionLog(ConvertionLog convertionLog) {
    this.convertionLog = convertionLog;
  }
}
```

A.1.5 LockLog

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import lock.Lock;

public class LockLog {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="lockLog:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  /**
   * @uml.property name="lockLogRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
   *                     inverse="lockLog:dataBase.LockLogRecord"
   */
  private Collection<LockLogRecord> lockLogRecord;

  public LockLog(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
    lockLogRecord = new ArrayList<LockLogRecord>();
  }

  /**
   */
  public void addLockLogRecord(String id, long time,
      String mobileHostId, String transactionId, String resourceId,
      String requestingLockType, ArrayList<String> currentLocks,
      ArrayList<String> pendingLocks, String action) {

    lockLogRecord.add(new LockLogRecord(this, id, new String(""
        + time), mobileHostId, transactionId, resourceId,
        requestingLockType, currentLocks, pendingLocks, action));
    databaseServer.setLockLogId(databaseServer.getLockLogId() + 1);
    if (!action.equals("REJECTED")) {
      if (requestingLockType.equals("RONLOCK")) {
        databaseServer.getConvertionLog().addConvertionLogRecord(id,
            "WIOFF");
      } else if (requestingLockType.equals("WOFFLOCK")) {
        databaseServer.getConvertionLog().addConvertionLogRecord(id,
            "WIOFF");
```

```
    } else if (requestingLockType.equals("WIOFFLOCK")) {
    } else {
    }
  }
}

/**
 */
public Collection<LockLogRecord> getLockLogRecordsFromTime(
    long fromTime) {
  Collection<LockLogRecord> returnList = new
   ArrayList<LockLogRecord>();
  for (Iterator<LockLogRecord> i = lockLogRecord.iterator(); i
      .hasNext();) {
    LockLogRecord lockLogRec = i.next();
    int time = new Integer(lockLogRec.getTime());
    if (time > fromTime) {
      returnList.add(lockLogRec);
    }
  }
  return returnList;
}

/**
 */
public long getTimeWhenLastLockWasGranted() {
  long returnValue = 0;
  for (Iterator<LockLogRecord> i = lockLogRecord.iterator(); i
      .hasNext();) {
    LockLogRecord lockLogRecord = i.next();
    returnValue = new Long(lockLogRecord.getTime());
  }
  return returnValue;
}

/**
 */
public long getTimeWhenLockWasGranted(Lock lock) {
  long returnValue = 0;
  for (Iterator<LockLogRecord> i = lockLogRecord.iterator(); i
      .hasNext();) {
    LockLogRecord lockLogRecord = i.next();
    if (lockLogRecord.getId().equals(lock.getLockLogId())) {
      returnValue = new Long(lockLogRecord.getTime());
    }
  }
  return returnValue;
}
```

```
public void printLockLog() {
  System.out.println("");
  System.out.println("The databaseservers locklog:");
  System.out.println("");
  System.out
      .println("Time MobileHostId TransactionId ResourceId
        RequestingLock CurrentLocks PendingLocks ActionAtDB");
  System.out.println("");
  for (Iterator<LockLogRecord> i = lockLogRecord.iterator(); i
      .hasNext();) {
    LockLogRecord lockLogRecord = i.next();
    System.out.println(lockLogRecord.getTime() + " "
        + lockLogRecord.getMobileHostId() + " "
        + lockLogRecord.getTransactionId() + " "
        + lockLogRecord.getResourceId() + " "
        + lockLogRecord.getRequstingLockType() + " "
        + lockLogRecord.getCurrentLocks() + " "
        + lockLogRecord.getPendingLocks() + " "
        + lockLogRecord.getAction());
  }
}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}

/**
 * Setter of the property <tt>databaseServer</tt>
 *
 * @param databaseServer
 *          The databaseServer to set.
 * @uml.property name="databaseServer"
 */
public void setDatabaseServer(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
}

/**
 * Getter of the property <tt>lockLogRecord</tt>
 *
 * @return Returns the lockLogRecord.
 * @uml.property name="lockLogRecord"
 */
```

```
  public Collection<LockLogRecord> getLockLogRecord() {
    return lockLogRecord;
  }

  /**
   * Setter of the property <tt>lockLogRecord</tt>
   *
   * @param lockLogRecord
   *          The lockLogRecord to set.
   * @uml.property name="lockLogRecord"
   */
  public void setLockLogRecord(
      Collection<LockLogRecord> lockLogRecord) {
    this.lockLogRecord = lockLogRecord;
  }
}
```

A.1.6 LockLogRecord

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;

public class LockLogRecord {

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="time"
   */
  private String time;

  /**
   * @uml.property name="mobileHostId"
   */
  private String mobileHostId;

  /**
   * @uml.property name="transactionId"
   */
  private String transactionId;

  /**
```

```
 * @uml.property name="resourceId"
 */
private String resourceId;

/**
 * @uml.property name="requstingLockType"
 */
private String requstingLockType;

/**
 * @uml.property name="currentLocks"
 */
private Collection<String> currentLocks;

/**
 * @uml.property name="pendingLocks"
 */
private Collection<String> pendingLocks;

/**
 * @uml.property name="action"
 */
private String action;

/**
 * @uml.property name="lockLog"
 * @uml.associationEnd inverse="lockLogRecord:dataBase.LockLog"
 */
private LockLog lockLog;

public LockLogRecord(LockLog lockLog, String id, String time,
    String mobileHostId, String transactionId, String resourceId,
    String requestingLockType, ArrayList<String> currentLocks,
    ArrayList<String> pendingLocks, String action) {
  this.lockLog = lockLog;
  this.id = id;
  this.time = time;
  this.mobileHostId = mobileHostId;
  this.transactionId = transactionId;
  this.resourceId = resourceId;
  this.requstingLockType = requestingLockType;
  this.currentLocks = currentLocks;
  this.pendingLocks = pendingLocks;
  this.action = action;

}

/**
 * Getter of the property <tt>lockLog</tt>
```

```
 *
 * @return Returns the lockLog.
 * @uml.property name="lockLog"
 */
public LockLog getLockLog() {
  return lockLog;
}

/**
 * Setter of the property <tt>lockLog</tt>
 *
 * @param lockLog
 *          The lockLog to set.
 * @uml.property name="lockLog"
 */
public void setLockLog(LockLog lockLog) {
  this.lockLog = lockLog;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>time</tt>
 *
 * @return Returns the time.
 * @uml.property name="time"
 */
public String getTime() {
  return time;
}
```

```
/**
 * Setter of the property <tt>time</tt>
 *
 * @param time
 *          The time to set.
 * @uml.property name="time"
 */
public void setTime(String time) {
  this.time = time;
}

/**
 * Getter of the property <tt>mobileHostId</tt>
 *
 * @return Returns the mobileHostId.
 * @uml.property name="mobileHostId"
 */
public String getMobileHostId() {
  return mobileHostId;
}

/**
 * Setter of the property <tt>mobileHostId</tt>
 *
 * @param mobileHostId
 *          The mobileHostId to set.
 * @uml.property name="mobileHostId"
 */
public void setMobileHostId(String mobileHostId) {
  this.mobileHostId = mobileHostId;
}

/**
 * Getter of the property <tt>transactionId</tt>
 *
 * @return Returns the transactionId.
 * @uml.property name="transactionId"
 */
public String getTransactionId() {
  return transactionId;
}

/**
 * Setter of the property <tt>transactionId</tt>
 *
 * @param transactionId
 *          The transactionId to set.
 * @uml.property name="transactionId"
```

```
 */
public void setTransactionId(String transactionId) {
  this.transactionId = transactionId;
}

/**
 * Getter of the property <tt>resourceId</tt>
 *
 * @return Returns the resourceId.
 * @uml.property name="resourceId"
 */
public String getResourceId() {
  return resourceId;
}

/**
 * Setter of the property <tt>resourceId</tt>
 *
 * @param resourceId
 *          The resourceId to set.
 * @uml.property name="resourceId"
 */
public void setResourceId(String resourceId) {
  this.resourceId = resourceId;
}

/**
 * Getter of the property <tt>requstingLockType</tt>
 *
 * @return Returns the requstingLockType.
 * @uml.property name="requstingLockType"
 */
public String getRequstingLockType() {
  return requstingLockType;
}

/**
 * Setter of the property <tt>requstingLockType</tt>
 *
 * @param requstingLockType
 *          The requstingLockType to set.
 * @uml.property name="requstingLockType"
 */
public void setRequstingLockType(String requstingLockType) {
  this.requstingLockType = requstingLockType;
}

/**
 * Getter of the property <tt>currentLocks</tt>
```

```
 *
 * @return Returns the currentLocks.
 * @uml.property name="currentLocks"
 */
public Collection<String> getCurrentLocks() {
  return currentLocks;
}

/**
 * Setter of the property <tt>currentLocks</tt>
 *
 * @param currentLocks
 *          The currentLocks to set.
 * @uml.property name="currentLocks"
 */
public void setCurrentLocks(Collection<String> currentLocks) {
  this.currentLocks = currentLocks;
}

/**
 * Getter of the property <tt>pendingLocks</tt>
 *
 * @return Returns the pendingLocks.
 * @uml.property name="pendingLocks"
 */
public Collection<String> getPendingLocks() {
  return pendingLocks;
}

/**
 * Setter of the property <tt>pendingLocks</tt>
 *
 * @param pendingLocks
 *          The pendingLocks to set.
 * @uml.property name="pendingLocks"
 */
public void setPendingLocks(Collection<String> pendingLocks) {
  this.pendingLocks = pendingLocks;
}

/**
 * Getter of the property <tt>action</tt>
 *
 * @return Returns the action.
 * @uml.property name="action"
 */
public String getAction() {
  return action;
}
```

```
  /**
   * Setter of the property <tt>action</tt>
   * @param action  The action to set.
   * @uml.property  name="action"
   */
  public void setAction(String action) {
    this.action = action;
  }
}
```

A.1.7 LockManager

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import transaction.Transaction;

import lock.Lock;
import lock.ROnLock;
import lock.WIOffLock;
import lock.WOffLock;
import lock.WOnLock;
import mobileHost.LocalLogRecord;

public class LockManager {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="lockManager:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  public LockManager(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  /**
   */
  public boolean handleROnLockRequest(Resource resource,
      Transaction transaction) {
    Collection<Lock> currentLocks = resource.getCurrentLocks();
    Collection<Lock> pendingLocks = resource.getPendingLocks();
```

```
if (resource.getAllowMoreLocks() == false) {
  databaseServer.getLockLog()
      .addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RONLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "REJECTED");
  return false;
}

if (currentLocks.isEmpty()) {
  // No locks
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "RONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
  Lock newLock = new ROnLock(transaction, resource);
  Integer lockLogId = databaseServer.getLockLogId() - 1;
  newLock.setLockLogId(lockLogId.toString());
  currentLocks.add(newLock);
  transaction.getLocks().add(newLock);
  return true;
}
if (!currentLocks.isEmpty()) {
  // Other readlocks
  boolean readLocks = true;
  for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock instanceof ROnLock) {
    } else {
      readLocks = false;
    }
  }
  if (readLocks) {
    boolean alreadyGotPendingWIOffLock = false;
    Lock wIOffLockWhichTransactionHasPending = null;
    for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotPendingWIOffLock = true;
          wIOffLockWhichTransactionHasPending = lock;
        }
      } else {
      }
    }
```

```
      if (alreadyGotPendingWIOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADE");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, false,
            true, wIOffLockWhichTransactionHasPending);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTED");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        transaction.getLocks().add(newLock);
        return true;
      }
    }
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotWIOffLock = true;
          wIOffLockWhichTransactionIsAlreadyHolding = lock;
        }
      } else {
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      if (alreadyGotWIOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
```

```
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADE");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDANDTAKEOVER");
        pendingLocks.addAll(currentLocks);
        currentLocks.clear();
        ROnLock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().add(newLock);
        return true;
      }
    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "RONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWOnLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();
  if (!currentLocks.isEmpty()) {
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WOffLock
          && (lock.getTransaction().getId().equals(transaction
```

```
            .getId())))) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDSELFUPGRADE");
        Lock newLock = new WOnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, lock);
        return true;
      }
    }
  }
  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }
  if (currentLocks.isEmpty()) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WONLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new WOnLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    resource.setAllowMoreLocks(false);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
```

```
          transaction.getId())) {
        alreadyGotWIOffLock = true;
        wIOffLockWhichTransactionIsAlreadyHolding = lock;
      }
    } else {
      wioffLocks = false;
    }
  }
  if (wioffLocks) {
    if (alreadyGotWIOffLock) {
      if (currentLocks.size() == 1) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADE");
        Lock newLock = new WOnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        resource.setAllowMoreLocks(false);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADEANDDELETE");
        Lock newLock = new WOnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        resource.setAllowMoreLocks(false);
        return true;
      }
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WONLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDANDTAKEOVERANDDELETE");
      for (Iterator<Lock> i = currentLocks.iterator(); i
          .hasNext();) {
        Lock lock = i.next();
```

```
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        WOnLock newLock = new WOnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().add(newLock);
        resource.setAllowMoreLocks(false);
        return true;
      }

    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWIOffLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();
  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }
  if (currentLocks.isEmpty()) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WIOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new WIOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
```

```
    transaction.getLocks().add(newLock);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // Other WIOffLocks
    boolean wioffLocks = true;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
      } else {
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTED");
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      currentLocks.add(newLock);
      transaction.getLocks().add(newLock);
      return true;
    }
  }
  if (!currentLocks.isEmpty()) {
    // readlocks
    boolean readLocks = true;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof ROnLock) {
      } else {
        readLocks = false;
      }
    }
    if (readLocks) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDPENDING");
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      pendingLocks.add(newLock);
```

```
      transaction.getLocks().add(newLock);
      return true;
    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WIOFFLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWOffLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();
  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }
  if (currentLocks.isEmpty()) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new WOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
```

```
      if (lock.getTransaction().getId().equals(
          transaction.getId())) {
        alreadyGotWIOffLock = true;
        wIOffLockWhichTransactionIsAlreadyHolding = lock;
      }
    } else {
      wioffLocks = false;
    }
  }
  if (wioffLocks) {
    if (alreadyGotWIOffLock) {
      if (currentLocks.size() == 1) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "GRANTEDUPGRADE");
        Lock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        resource.setAllowMoreLocks(false);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "GRANTEDUPGRADEANDDELETE");
        Lock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        resource.setAllowMoreLocks(false);
        return true;
      }
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDANDTAKEOVER");
```

```
        for (Iterator<Lock> i = currentLocks.iterator(); i
            .hasNext();) {
          Lock lock = i.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        WOffLock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        transaction.getLocks().add(newLock);
        resource.setAllowMoreLocks(false);
        return true;
      }
    }
  }
  if (!currentLocks.isEmpty()) {
    // readlocks
    boolean readLocks = true;
    boolean alreadyGotROnLock = false;
    Lock rOnLockWhichTransactionIsAlreadyHolding = null;
    boolean alreadyGotPendingWIOffLock = false;
    Lock wIOffLockWhichTransactionHasPending = null;
    ArrayList<Lock> rOnLocks = new ArrayList<Lock>();
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof ROnLock) {
        rOnLocks.add(lock);
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotROnLock = true;
          rOnLockWhichTransactionIsAlreadyHolding = lock;
          rOnLocks.remove(lock);
        }
      } else {
        readLocks = false;
      }
    }
    for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotPendingWIOffLock = true;
          wIOffLockWhichTransactionHasPending = lock;
        }
      }
    }
    if (readLocks) {
```

```
if (alreadyGotROnLock) {
  if (currentLocks.size() == 1) {
    if (pendingLocks.isEmpty()) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(),
          transaction.getId(), resource.getId(), "WOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "GRANTEDUPGRADE");
      Lock newLock = new WOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      upgradeLock(newLock, currentLocks, pendingLocks, true,
          false, rOnLockWhichTransactionIsAlreadyHolding);
      resource.setAllowMoreLocks(false);
      return true;
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(),
          transaction.getId(), resource.getId(), "WOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "GRANTEDUPGRADEANDDELETE");
      Lock newLock = new WOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      upgradeLock(newLock, currentLocks, pendingLocks, true,
          false, rOnLockWhichTransactionIsAlreadyHolding);
      resource.setAllowMoreLocks(false);
      return true;
    }
  } else {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId())
            .toString(), databaseServer.getTime(),
        transaction.getOwner().getId(), transaction.getId(),
        resource.getId(), "WOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks),
        "GRANTEDUPGRADEPENDING");
    Lock newLock = new WOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    upgradeLock(newLock, currentLocks, pendingLocks, true,
        false, rOnLockWhichTransactionIsAlreadyHolding);
    resource.setAllowMoreLocks(false);
    notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
    return true;
```

```
        }
      } else if (alreadyGotPendingWIOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADEPENDING");
        Lock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, false,
            true, wIOffLockWhichTransactionHasPending);
        resource.setAllowMoreLocks(false);
        notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDPENDING");
        Lock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        pendingLocks.add(newLock);
        transaction.getLocks().add(newLock);
        resource.setAllowMoreLocks(false);
        notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
        return true;
      }
    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WOFFLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public void releaseLocks(Transaction transaction) {
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    Resource resource = lock.getResource();
```

```
Collection<Lock> currentLocks = lock.getResource()
    .getCurrentLocks();
Collection<Lock> pendingLocks = lock.getResource()
    .getPendingLocks();
if (currentLocks.contains(lock)) {
  if (lock instanceof ROnLock) {
    if (currentLocks.size() == 1 && pendingLocks.size() > 0) {
      boolean existPendingWOffLock = false;
      Lock pendingWOffLock = null;
      for (Iterator<Lock> j = pendingLocks.iterator(); j
          .hasNext();) {
        Lock lock2 = j.next();
        if (lock2 instanceof WOffLock) {
          existPendingWOffLock = true;
          pendingWOffLock = lock2;
        }
      }
      if (existPendingWOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(),
            transaction.getId(), resource.getId(),
            "RELEASE_RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks),
            "RELEASEDDELEGATEWOFFLOCKSANDDELETE");
        currentLocks.remove(lock);
        Collection<Lock> newCurrentLock = new ArrayList<Lock>();
        newCurrentLock.add(pendingWOffLock);
        pendingLocks.remove(pendingWOffLock);
        for (Iterator<Lock> j = pendingLocks.iterator(); j
            .hasNext();) {
          Lock lock2 = j.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock2);
        }
        delegateLock(resource, newCurrentLock, null);
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(),
            transaction.getId(), resource.getId(),
            "RELEASE_RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks),
            "RELEASEDDELEGATEWIOFFLOCKS");
        currentLocks.remove(lock);
        delegateLock(resource, pendingLocks, null);
      }
    } else {
```

```
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_RONLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASED");
        currentLocks.remove(lock);
      }
    } else if (lock instanceof WOnLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_WONLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      currentLocks.remove(lock);
      resource.setAllowMoreLocks(true);
    } else if (lock instanceof WIOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_WIOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      currentLocks.remove(lock);
    } else if (lock instanceof WOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_WOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      currentLocks.remove(lock);
      resource.setAllowMoreLocks(true);
    }
  } else if (pendingLocks.contains(lock)) {
    if (lock instanceof WIOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_PENDINGWIOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      pendingLocks.remove(lock);
```

```
      } else if (lock instanceof WOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_PENDINGWOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASED");
        pendingLocks.remove(lock);
        resource.setAllowMoreLocks(true);
      }
    }
  }
  transaction.getLocks().clear();
}

/**
 */
public void upgradeLock(Lock newLock,
    Collection<Lock> currentLocks, Collection<Lock> pendingLocks,
    boolean alreadyGotCurrentLock, boolean alreadyGotPendingLock,
    Lock oldLock) {
  if (alreadyGotCurrentLock) {
    if (newLock instanceof ROnLock) {
      currentLocks.remove(oldLock);
      pendingLocks.addAll(currentLocks);
      currentLocks.clear();
      currentLocks.add(newLock);
      Transaction t = newLock.getTransaction();
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);

    } else if (newLock instanceof WOffLock) {
      if (oldLock instanceof ROnLock) {
        if (currentLocks.size() == 1) {
          if (pendingLocks.size() > 0) {
            for (Iterator<Lock> i = pendingLocks.iterator(); i
                .hasNext();) {
              Lock lock = i.next();
              notifyWIOffOwnerAboutLostWIOffLock(lock);
            }
            pendingLocks.clear();
          }
          currentLocks.remove(oldLock);
          currentLocks.add(newLock);
          Transaction t = newLock.getTransaction();
          t.getLocks().remove(oldLock);
          t.getLocks().add(newLock);
        } else {
```

```
          currentLocks.remove(oldLock);
          pendingLocks.add(newLock);
          Transaction t = newLock.getTransaction();
          t.getLocks().remove(oldLock);
          t.getLocks().add(newLock);
        }
      } else {
        currentLocks.remove(oldLock);
        for (Iterator<Lock> i = currentLocks.iterator(); i
            .hasNext();) {
          Lock lock = i.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      }
    } else if (newLock instanceof WOnLock) {
      if (oldLock instanceof WIOffLock) {
        currentLocks.remove(oldLock);
        if (currentLocks.size() > 0) {
          for (Iterator<Lock> i = currentLocks.iterator(); i
              .hasNext();) {
            Lock lock = i.next();
            notifyWIOffOwnerAboutLostWIOffLock(lock);
          }
          currentLocks.clear();
        }
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      } else {
        currentLocks.remove(oldLock);
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      }
    }
  } else if (alreadyGotPendingLock) {
    if (newLock instanceof ROnLock) {
      pendingLocks.remove(oldLock);
      currentLocks.add(newLock);
      Transaction t = newLock.getTransaction();
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);
```

```
    } else if (newLock instanceof WOffLock) {
      pendingLocks.remove(oldLock);
      pendingLocks.add(newLock);
      Transaction t = newLock.getTransaction();
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);
    }
  } else {
  }
}

/**
 */
public void delegateLock(Resource resource,
    Collection<Lock> newCurrentLocks,
    Collection<Lock> newPendingLocks) {
  resource.getCurrentLocks().addAll(newCurrentLocks);
  if (newPendingLocks == null) {
    resource.getPendingLocks().clear();

  } else {
    resource.getPendingLocks().addAll(newPendingLocks);
  }
}

/**
 */
public void handleDisconnectedTransaction(Transaction transaction,
    boolean planned) {
  Collection<Lock> locks = transaction.getLocks();
  Collection<Lock> locksToAdd = new ArrayList<Lock>();
  Collection<Lock> locksToRemove = new ArrayList<Lock>();
  for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock instanceof ROnLock) {
      Resource resource = lock.getResource();
      Collection<Lock> currentLocks = resource.getCurrentLocks();
      Collection<Lock> pendingLocks = resource.getPendingLocks();
      if (currentLocks.size() == 1) {
        boolean existPendingWOffLock = false;
        Lock pendingWOffLock = null;
        for (Iterator<Lock> j = pendingLocks.iterator(); j
            .hasNext();) {
          Lock lock2 = j.next();
          if (lock2 instanceof WOffLock) {
            existPendingWOffLock = true;
            pendingWOffLock = lock2;
          }
        }
```

```
    if (existPendingWOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "DELEGATE_RONLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "DISCONNECTEDDELEGATEANDDELETE");
      currentLocks.remove(lock);
      Collection<Lock> newCurrentLock = new ArrayList<Lock>();
      newCurrentLock.add(pendingWOffLock);
      pendingLocks.remove(pendingWOffLock);
      for (Iterator<Lock> j = pendingLocks.iterator(); j
          .hasNext();) {
        Lock lock2 = j.next();
        notifyWIOffOwnerAboutLostWIOffLock(lock2);
      }
      delegateLock(resource, newCurrentLock, null);
      locksToRemove.add(lock);
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "DELEGATE_RONLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "DISCONNECTEDDELEGATE");
      currentLocks.remove(lock);
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      Collection<Lock> newCurrentLock = new ArrayList<Lock>();
      newCurrentLock.add(newLock);
      newCurrentLock.addAll(pendingLocks);
      delegateLock(resource, newCurrentLock, null);
      locksToRemove.add(lock);
      locksToAdd.add(newLock);
    }
  } else {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(),
        transaction.getOwner().getId(), transaction.getId(),
        resource.getId(), "DELEGATE_RONLOCK",
        getLocks(currentLocks), getLocks(pendingLocks),
        "DISCONNECTEDDELEGATE");
    currentLocks.remove(lock);
    Lock newLock = new WIOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
```

```
        newLock.setLockLogId(lockLogId.toString());
        Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
        Collection<Lock> newPendingLocks = new ArrayList<Lock>();
        newCurrentLocks.addAll(currentLocks);
        pendingLocks.add(newLock);
        newPendingLocks.addAll(pendingLocks);
        currentLocks.clear();
        pendingLocks.clear();
        delegateLock(resource, newCurrentLocks, newPendingLocks);
        locksToRemove.add(lock);
        locksToAdd.add(newLock);
      }
    }
  }
  if (planned) {
    locks.removeAll(locksToRemove);
    locks.addAll(locksToAdd);
  } else {
    databaseServer.getLostLocksWhenMobileHostIsDisconnected()
        .addAll(locksToRemove);
    databaseServer.getGainedLocksWhenMobileHostIsDisconnected()
        .addAll(locksToAdd);
  }
}

/**
 */
public void handleReconnectedTransaction(Transaction transaction) {
  Collection<Lock> locksToAdd = new ArrayList<Lock>();
  Collection<Lock> locksToRemove = new ArrayList<Lock>();
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    if (lock.getLockType().equals("WOFFLOCK")) {
      Resource resource = lock.getResource();
      Collection<Lock> currentLocks = resource.getCurrentLocks();
      Collection<Lock> pendingLocks = resource.getPendingLocks();
      boolean valueChanged = false;
      for (Iterator<LocalLogRecord> j = transaction.getOwner()
          .getLocalLog().getLocalLogRecordsFromLastDisconnection(
              transaction.getOwner().getLastDisconnectionTime())
          .iterator(); j.hasNext();) {
        LocalLogRecord localLogRecord = j.next();
        if (localLogRecord.getTransactionId().equals(
            transaction.getId())
            && localLogRecord.getResourceId().equals(
                lock.getResource().getId())
            && localLogRecord.getAction().equals("WRITEOFFLINE")) {
          valueChanged = true;
```

```
        }
      }
      if (!valueChanged) {
        if (lock.getResource().getCurrentLocks().contains(lock)) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "DELEGATE_WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "RECONNECTDELEGATE");
          currentLocks.remove(lock);
          Lock newLock = new WIOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
          newCurrentLocks.add(newLock);
          locksToAdd.add(newLock);
          locksToRemove.add(lock);
          resource.setAllowMoreLocks(true);
          delegateLock(resource, newCurrentLocks, null);
        } else {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "DELEGATE_WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "RECONNECTDELEGATE");
          pendingLocks.remove(lock);
          Lock newLock = new WIOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
          Collection<Lock> newPendingLocks = new ArrayList<Lock>();
          newPendingLocks.add(newLock);
          locksToAdd.add(newLock);
          locksToRemove.add(lock);
          resource.setAllowMoreLocks(true);
          delegateLock(resource, newCurrentLocks, newPendingLocks);
        }
      }
    }
  }
  transaction.getLocks().removeAll(locksToRemove);
  transaction.getLocks().addAll(locksToAdd);
}

/**
```

```
   */
  private void notifyROnLockOwnersAboutGrantedWOffLock(
      Collection<Lock> rOnLocks) {
    for (Iterator<Lock> i = rOnLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      lock.getTransaction().getOwner()
          .notifyAboutGrantedWOffLockOfROnLockedResource(
              (ROnLock) lock);
    }
  }

  private void notifyWIOffOwnerAboutLostWIOffLock(Lock lostLock) {
    databaseServer.notifyAboutLostWIOffLock(lostLock);
  }

  /**
   * Getter of the property <tt>databaseServer</tt>
   *
   * @return Returns the databaseServer.
   * @uml.property name="databaseServer"
   */
  public DatabaseServer getDatabaseServer() {
    return databaseServer;
  }

  /**
   * Setter of the property <tt>databaseServer</tt>
   *
   * @param databaseServer
   *          The databaseServer to set.
   * @uml.property name="databaseServer"
   */
  public void setDatabaseServer(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  private ArrayList<String> getLocks(Collection<Lock> locks) {
    ArrayList<String> returnArrayList = new ArrayList<String>();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      String lockString = "";
      Lock lock = i.next();
      lockString = lockString + lock.getResource().getId();
      lockString = lockString + "(" + lock.getTransaction().getId()
          + "," + lock.getLockType() + ")";
      returnArrayList.add(lockString);
    }
    return returnArrayList;
  }
}
```

A.1.8 Log

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class Log {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="log:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  /**
   * @uml.property name="logRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
    inverse="log:dataBase.LogRecord"
   */
  private Collection<LogRecord> logRecord;

  /**
   * @uml.property name="logId"
   */
  private int logId;

  public Log(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
    logRecord = new ArrayList<LogRecord>();
    logId = 0;
  }

  /**
   */
  public void addLogRecord(String transactionId, String action,
      String resourceId, String value, String newValue) {
    logRecord.add(new LogRecord(this, new String("" + logId),
        new String("" + databaseServer.getTime()), transactionId,
        action, resourceId, value, newValue));
    logId = logId + 1;
  }

  /**
   */
  public Collection<LogRecord> getLogRecordsFromTime(long fromTime) {
    Collection<LogRecord> returnList = new ArrayList<LogRecord>();
```

```
  for (Iterator<LogRecord> i = logRecord.iterator(); i.hasNext();) {
    LogRecord logRec = i.next();
    int time = new Integer(logRec.getTime());
    if (time > fromTime) {
      returnList.add(logRec);
    }
  }
  return returnList;
}

/**
 */
public void printLog() {
  System.out.println("");
  System.out.println("The databaseservers log:");
  System.out.println("");
  System.out
      .println("Time TransactionId action ResourceId  value newValue");
  System.out.println("");
  for (Iterator<LogRecord> i = logRecord.iterator(); i.hasNext();) {
    LogRecord logRecord = i.next();
    System.out.println(logRecord.getTime() + " "
        + logRecord.getTransactionId() + " "
        + logRecord.getAction() + " " + logRecord.getResourceId()
        + " " + logRecord.getValue() + " "
        + logRecord.getNewValue());
  }
}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}

/**
 * Setter of the property <tt>databaseServer</tt>
 *
 * @param databaseServer
 *          The databaseServer to set.
 * @uml.property name="databaseServer"
 */
public void setDatabaseServer(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
}
```

```
  /**
   * Getter of the property <tt>logId</tt>
   *
   * @return Returns the logId.
   * @uml.property name="logId"
   */
  public int getLogId() {
    return logId;
  }

  /**
   * Setter of the property <tt>logId</tt>
   *
   * @param logId
   *          The logId to set.
   * @uml.property name="logId"
   */
  public void setLogId(int logId) {
    this.logId = logId;
  }

  /**
   * Getter of the property <tt>logRecord</tt>
   *
   * @return Returns the logRecord.
   * @uml.property name="logRecord"
   */
  public Collection<LogRecord> getLogRecord() {
    return logRecord;
  }

  /**
   * Setter of the property <tt>logRecord</tt>
   *
   * @param logRecord
   *          The logRecord to set.
   * @uml.property name="logRecord"
   */
  public void setLogRecord(Collection<LogRecord> logRecord) {
    this.logRecord = logRecord;
  }
}
```

A.1.9 LogRecord

```
package dataBase;
```

```
public class LogRecord {

  /**
   * @uml.property name="log"
   * @uml.associationEnd inverse="logRecord:dataBase.Log"
   */
  private Log log;

  /**
   * @uml.property name="logId"
   */
  private String logId;

  /**
   * @uml.property name="time"
   */
  private String time;

  /**
   * @uml.property name="transactionId"
   */
  private String transactionId;

  /**
   * @uml.property name="action"
   */
  private String action = "";

  /**
   * @uml.property name="resourceId"
   */
  private String resourceId = "";

  /**
   * @uml.property name="value"
   */
  private String value = "";

  /**
   * @uml.property name="newValue"
   */
  private String newValue = "";

  public LogRecord(Log log, String logId, String time,
      String transactionId, String action, String resourceId,
      String value, String newValue) {
    this.log = log;
    this.logId = logId;
```

```
  this.time = time;
  this.transactionId = transactionId;
  this.action = action;
  this.resourceId = resourceId;
  this.value = value;
  this.newValue = newValue;
}

/**
 * Getter of the property <tt>transactionId</tt>
 *
 * @return Returns the transactionId.
 * @uml.property name="transactionId"
 */
public String getTransactionId() {
  return transactionId;
}

/**
 * Setter of the property <tt>transactionId</tt>
 *
 * @param transactionId
 *          The transactionId to set.
 * @uml.property name="transactionId"
 */
public void setTransactionId(String transactionId) {
  this.transactionId = transactionId;
}

/**
 * Getter of the property <tt>resourceId</tt>
 *
 * @return Returns the resourceId.
 * @uml.property name="resourceId"
 */
public String getResourceId() {
  return resourceId;
}

/**
 * Setter of the property <tt>resourceId</tt>
 *
 * @param resourceId
 *          The resourceId to set.
 * @uml.property name="resourceId"
 */
public void setResourceId(String resourceId) {
  this.resourceId = resourceId;
}
```

```
/**
 * Getter of the property <tt>logId</tt>
 *
 * @return Returns the logId.
 * @uml.property name="logId"
 */
public String getLogId() {
  return logId;
}

/**
 * Setter of the property <tt>logId</tt>
 *
 * @param logId
 *          The logId to set.
 * @uml.property name="logId"
 */
public void setLogId(String logId) {
  this.logId = logId;
}

/**
 * Getter of the property <tt>time</tt>
 *
 * @return Returns the time.
 * @uml.property name="time"
 */
public String getTime() {
  return time;
}

/**
 * Setter of the property <tt>time</tt>
 *
 * @param time
 *          The time to set.
 * @uml.property name="time"
 */
public void setTime(String time) {
  this.time = time;
}

/**
 * Getter of the property <tt>action</tt>
 *
 * @return Returns the action.
 * @uml.property name="action"
 */
```

```
public String getAction() {
  return action;
}

/**
 * Setter of the property <tt>action</tt>
 *
 * @param action
 *          The action to set.
 * @uml.property name="action"
 */
public void setAction(String action) {
  this.action = action;
}

/**
 * Getter of the property <tt>value</tt>
 *
 * @return Returns the value.
 * @uml.property name="value"
 */
public String getValue() {
  return value;
}

/**
 * Setter of the property <tt>value</tt>
 *
 * @param value
 *          The value to set.
 * @uml.property name="value"
 */
public void setValue(String value) {
  this.value = value;
}

/**
 * Getter of the property <tt>newValue</tt>
 *
 * @return Returns the newValue.
 * @uml.property name="newValue"
 */
public String getNewValue() {
  return newValue;
}

/**
 * Setter of the property <tt>newValue</tt>
 *
```

```
   * @param newValue
   *          The newValue to set.
   * @uml.property name="newValue"
   */
  public void setNewValue(String newValue) {
    this.newValue = newValue;
  }

  /**
   * Getter of the property <tt>log</tt>
   *
   * @return Returns the log.
   * @uml.property name="log"
   */
  public Log getLog() {
    return log;
  }

  /**
   * Setter of the property <tt>log</tt>
   * @param log  The log to set.
   * @uml.property  name="log"
   */
  public void setLog(Log log) {
    this.log = log;
  }
}
```

A.1.10 MobileSupportStation

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;

import mobileHost.MobileHost;

public class MobileSupportStation {

  /**
   * @uml.property name="id"
   */
  private String id = "";

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd multiplicity="(0 -1)"
```

```
 *                     inverse="mobileHost:mobileHost.MobileHost"
 */
private Collection<MobileHost> mobileHosts;

/**
 * @uml.property name="databaseServer"
 * @uml.associationEnd
  inverse="mobileSupportStation:dataBase.DatabaseServer"
 */
private DatabaseServer databaseServer;

public MobileSupportStation(String id,
    DatabaseServer databaseServer) {
  this.id = id;
  this.databaseServer = databaseServer;
  mobileHosts = new ArrayList<MobileHost>();
}

/**
 */
public void register(MobileHost mobileHost) {
  mobileHosts.add(mobileHost);
  databaseServer.getConnectivityManager().getOnlineMobileHosts()
      .add(mobileHost);
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>mobileHost</tt>
```

```
   *
   * @return Returns the mobileHost.
   * @uml.property name="mobileHost"
   */
  public Collection<MobileHost> getMobileHost() {
    return mobileHosts;
  }

  /**
   * Setter of the property <tt>mobileHost</tt>
   *
   * @param mobileHost
   *          The mobileHost to set.
   * @uml.property name="mobileHost"
   */
  public void setMobileHost(Collection<MobileHost> mobileHosts) {
    this.mobileHosts = mobileHosts;
  }

  /**
   * Getter of the property <tt>databaseServer</tt>
   *
   * @return Returns the databaseServer.
   * @uml.property name="databaseServer"
   */
  public DatabaseServer getDatabaseServer() {
    return databaseServer;
  }

  /**
   * Setter of the property <tt>databaseServer</tt>
   *
   * @param databaseServer
   *          The databaseServer to set.
   * @uml.property name="databaseServer"
   */
  public void setDatabaseServer(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }
}
```

A.1.11 MobileTask

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
```

```
public class MobileTask {

  /**
   * @uml.property name="id"
   */
  private String id = "";

  /**
   * @uml.property name="resource"
   * @uml.associationEnd multiplicity="(0 -1)"
   *                     inverse="mobileTask:dataBase.Resource"
   */
  private Collection<Resource> resource;

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="mobileTask:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  public MobileTask(String id, DatabaseServer databaseServer) {
    this.id = id;
    this.databaseServer = databaseServer;
    resource = new ArrayList<Resource>();
  }

  /**
   * Getter of the property <tt>id</tt>
   *
   * @return Returns the id.
   * @uml.property name="id"
   */
  public String getId() {
    return id;
  }

  /**
   * Setter of the property <tt>id</tt>
   *
   * @param id
   *          The id to set.
   * @uml.property name="id"
   */
  public void setId(String id) {
    this.id = id;
  }

  /**
```

```
   * Getter of the property <tt>databaseServer</tt>
   *
   * @return Returns the databaseServer.
   * @uml.property name="databaseServer"
   */
  public DatabaseServer getDatabaseServer() {
    return databaseServer;
  }

  /**
   * Setter of the property <tt>databaseServer</tt>
   *
   * @param databaseServer
   *          The databaseServer to set.
   * @uml.property name="databaseServer"
   */
  public void setDatabaseServer(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  /**
   * Getter of the property <tt>resource</tt>
   *
   * @return Returns the resource.
   * @uml.property name="resource"
   */
  public Collection<Resource> getResource() {
    return resource;
  }

  /**
   * Setter of the property <tt>resource</tt>
   *
   * @param resource
   *          The resource to set.
   * @uml.property name="resource"
   */
  public void setResource(Collection<Resource> resource) {
    this.resource = resource;
  }
}
```

A.1.12 Resource

```
package dataBase;

import java.util.ArrayList;
```

```
import java.util.Collection;

import lock.Lock;

public class Resource {

  /**
   * @uml.property name="id"
   */
  private String id = "";

  /**
   * @uml.property name="value"
   */
  private int value;

  /**
   * @uml.property name="allowMoreLocks"
   */
  private boolean allowMoreLocks;

  /**
   * @uml.property name="currentLock"
   */
  private Collection<Lock> currentLocks;

  /**
   * @uml.property name="pendingLocks"
   */
  private Collection<Lock> pendingLocks;

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="resource:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  /**
   * @uml.property name="mobileTask"
   * @uml.associationEnd multiplicity="(0 -1)"
   *                     inverse="resource:dataBase.MobileTask"
   */
  private Collection mobileTask;

  public Resource(DatabaseServer databaseServer, int value, String id) {
    this.databaseServer = databaseServer;
    this.value = value;
    this.id = id;
    allowMoreLocks = true;
```

```
    currentLocks = new ArrayList<Lock>();
    pendingLocks = new ArrayList<Lock>();
  }

  /**
   * Getter of the property <tt>id</tt>
   *
   * @return Returns the id.
   * @uml.property name="id"
   */
  public String getId() {
    return id;
  }

  /**
   * Setter of the property <tt>id</tt>
   *
   * @param id
   *          The id to set.
   * @uml.property name="id"
   */
  public void setId(String id) {
    this.id = id;
  }

  /**
   * Getter of the property <tt>value</tt>
   *
   * @return Returns the value.
   * @uml.property name="value"
   */
  public int getValue() {
    return value;
  }

  /**
   * Setter of the property <tt>value</tt>
   *
   * @param value
   *          The value to set.
   * @uml.property name="value"
   */
  public void setValue(int value) {
    this.value = value;
  }

  /**
   * Getter of the property <tt>allowMoreLocks</tt>
   *
```

```
 * @return Returns the allowMoreLocks.
 * @uml.property name="allowMoreLocks"
 */
public boolean getAllowMoreLocks() {
  return allowMoreLocks;
}

/**
 * Setter of the property <tt>allowMoreLocks</tt>
 *
 * @param allowMoreLocks
 *          The allowMoreLocks to set.
 * @uml.property name="allowMoreLocks"
 */
public void setAllowMoreLocks(boolean allowMoreLocks) {
  this.allowMoreLocks = allowMoreLocks;
}

/**
 * Getter of the property <tt>currentLock</tt>
 *
 * @return Returns the currentLock.
 * @uml.property name="currentLock"
 */
public Collection<Lock> getCurrentLocks() {
  return currentLocks;
}

/**
 * Setter of the property <tt>currentLock</tt>
 *
 * @param currentLock
 *          The currentLock to set.
 * @uml.property name="currentLock"
 */
public void setCurrentLocks(Collection<Lock> currentLocks) {
  this.currentLocks = currentLocks;
}

/**
 * Getter of the property <tt>pendingLocks</tt>
 *
 * @return Returns the pendingLocks.
 * @uml.property name="pendingLocks"
 */
public Collection<Lock> getPendingLocks() {
  return pendingLocks;
}
```

```
/**
 * Setter of the property <tt>pendingLocks</tt>
 *
 * @param pendingLocks
 *          The pendingLocks to set.
 * @uml.property name="pendingLocks"
 */
public void setPendingLocks(Collection<Lock> pendingLocks) {
  this.pendingLocks = pendingLocks;
}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}

/**
 * Setter of the property <tt>databaseServer</tt>
 *
 * @param databaseServer
 *          The databaseServer to set.
 * @uml.property name="databaseServer"
 */
public void setDatabaseServer(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
}

/**
 * Getter of the property <tt>mobileTask</tt>
 *
 * @return Returns the mobileTask.
 * @uml.property name="mobileTask"
 */
public Collection getMobileTask() {
  return mobileTask;
}

/**
 * Setter of the property <tt>mobileTask</tt>
 *
 * @param mobileTask
 *          The mobileTask to set.
 * @uml.property name="mobileTask"
 */
```

```
  public void setMobileTask(Collection mobileTask) {
    this.mobileTask = mobileTask;
  }
}
```

A.2 Modulen *mobileHost*

Klassene i modulen *mobileHost* er:

- `MobileHost`
- `Cache`
- `LocalCacheManager`
- `LocalLockLog`
- `LocalLockLogRecord`
- `LocalLog`
- `LocalLogRecord`
- `LocalTransactionManger`

A.2.1 `MobileHost`

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;

import dataBase.MobileSupportStation;
import dataBase.MobileTask;

import lock.Lock;
import lock.ROnLock;

public class MobileHost {

  /**
   * @uml.property name="id"
   */
  private String id = "";

  /**
   * @uml.property name="localTime"
   */
  private long localTime;

  /**
```

```
 * @uml.property name="lastDisconnectionTime"
 */
private long lastDisconnectionTime;

/**
 * @uml.property name="mobileTasks"
 */
private Collection<MobileTask> mobileTasks;

/**
 * @uml.property name="connectionStatus"
 */
private boolean connectionStatus;

/**
 * @uml.property name="mobileSupportStation"
 * @uml.associationEnd
  inverse="mobileHost:dataBase.MobileSupportStation"
 */
private MobileSupportStation mobileSupportStation;

/**
 * @uml.property name="localTransactionManager"
 * @uml.associationEnd
  inverse="mobileHost:mobileHost.LocalTransactionManager"
 */
private LocalTransactionManager localTransactionManager;

/**
 * @uml.property name="localCacheManager"
 * @uml.associationEnd
  inverse="mobileHost:mobileHost.LocalCacheManager"
 */
private LocalCacheManager localCacheManager;

/**
 * @uml.property name="cache"
 * @uml.associationEnd inverse="mobileHost:mobileHost.Cache"
 */
private Cache cache;

/**
 * @uml.property name="localLockLog"
 * @uml.associationEnd inverse="mobileHost:mobileHost.LocalLockLog"
 */
private LocalLockLog localLockLog;

/**
 * @uml.property name="localLog"
```

```
 * @uml.associationEnd inverse="mobileHost:mobileHost.LocalLog"
 */
private LocalLog localLog;

private long nanoStartTime;

public MobileHost(String id, MobileSupportStation mss,
    int cacheSize, long localTime) {
  this.id = id;
  mobileSupportStation = mss;
  this.localTime = localTime;
  nanoStartTime = mobileSupportStation.getDatabaseServer()
      .getNanoStartTime();
  lastDisconnectionTime = 0;
  connectionStatus = true;
  mobileTasks = new ArrayList<MobileTask>();
  localTransactionManager = new LocalTransactionManager(this);
  cache = new Cache(this, cacheSize);
  localCacheManager = new LocalCacheManager(this, cache);
  localLockLog = new LocalLockLog(this);
  localLog = new LocalLog(this);

}

/**
 */
public void reconnect() {
  connectionStatus = true;
  mobileSupportStation.getDatabaseServer().reconnect(this);
}

/**
 */
public void disconnectPlanned() {
  localTime = mobileSupportStation.getDatabaseServer().getTime();
  lastDisconnectionTime = localTime;
  mobileSupportStation.getDatabaseServer()
      .initiatePlannedDisconnection(this);
  connectionStatus = false;
}

/**
 */
public void disconnectUnplanned() {
  localTime = mobileSupportStation.getDatabaseServer().getTime();
  lastDisconnectionTime = localTime;
  connectionStatus = false;
  try {
    Thread.sleep(new Integer(500).longValue());
```

```
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
  }

  /**
   */
  public void startNewTransaction(String transactionId) {
    localTransactionManager.executeStartOperation(transactionId);
  }

  /**
   */
  public void notifyAboutGrantedWOffLockOfROnLockedResource(
      ROnLock lock) {
    lock.setGrantedWOffLock(true);
  }

  /**
   */
  public void notifyAboutLostWIOffLock(Lock lostLock) {
    lostLock.getTransaction().getLocks().remove(lostLock);
    localCacheManager.removeResource(lostLock.getResource(),
        lostLock.getTransaction());
  }

  /**
   * Getter of the property <tt>id</tt>
   *
   * @return Returns the id.
   * @uml.property name="id"
   */
  public String getId() {
    return id;
  }

  /**
   * Setter of the property <tt>id</tt>
   *
   * @param id
   *          The id to set.
   * @uml.property name="id"
   */
  public void setId(String id) {
    this.id = id;
  }

  /**
   * Getter of the property <tt>localTime</tt>
```

```
 *
 * @return Returns the localTime.
 * @uml.property name="localTime"
 */
public long getLocalTime() {
  return System.nanoTime() - nanoStartTime;
}

/**
 * Setter of the property <tt>localTime</tt>
 *
 * @param localTime
 *          The localTime to set.
 * @uml.property name="localTime"
 */
public void setLocalTime(long localTime) {
  this.localTime = localTime;
}

/**
 * Getter of the property <tt>mobileTasks</tt>
 *
 * @return Returns the mobileTasks.
 * @uml.property name="mobileTasks"
 */
public Collection<MobileTask> getMobileTasks() {
  return mobileTasks;
}

/**
 * Setter of the property <tt>mobileTasks</tt>
 *
 * @param mobileTasks
 *          The mobileTasks to set.
 * @uml.property name="mobileTasks"
 */
public void setMobileTasks(Collection<MobileTask> mobileTasks) {
  this.mobileTasks = mobileTasks;
}

/**
 * Getter of the property <tt>connectionStatus</tt>
 *
 * @return Returns the connectionStatus.
 * @uml.property name="connectionStatus"
 */
public boolean getConnectionStatus() {
  return connectionStatus;
}
```

```
/**
 * Setter of the property <tt>connectionStatus</tt>
 *
 * @param connectionStatus
 *          The connectionStatus to set.
 * @uml.property name="connectionStatus"
 */
public void setConnectionStatus(boolean connectionStatus) {
  this.connectionStatus = connectionStatus;
}

/**
 * Getter of the property <tt>mobileSupportStation</tt>
 *
 * @return Returns the mobileSupportStation.
 * @uml.property name="mobileSupportStation"
 */
public MobileSupportStation getMobileSupportStation() {
  return mobileSupportStation;
}

/**
 * Setter of the property <tt>mobileSupportStation</tt>
 *
 * @param mobileSupportStation
 *          The mobileSupportStation to set.
 * @uml.property name="mobileSupportStation"
 */
public void setMobileSupportStation(
    MobileSupportStation mobileSupportStation) {
  this.mobileSupportStation = mobileSupportStation;
}

/**
 * Getter of the property <tt>localTransactionManager</tt>
 *
 * @return Returns the localTransactionManager.
 * @uml.property name="localTransactionManager"
 */
public LocalTransactionManager getLocalTransactionManager() {
  return localTransactionManager;
}

/**
 * Setter of the property <tt>localTransactionManager</tt>
 *
 * @param localTransactionManager
 *          The localTransactionManager to set.
```

```
 * @uml.property name="localTransactionManager"
 */
public void setLocalTransactionManager(
    LocalTransactionManager localTransactionManager) {
  this.localTransactionManager = localTransactionManager;
}

/**
 * Getter of the property <tt>localCacheManager</tt>
 *
 * @return Returns the localCacheManager.
 * @uml.property name="localCacheManager"
 */
public LocalCacheManager getLocalCacheManager() {
  return localCacheManager;
}

/**
 * Setter of the property <tt>localCacheManager</tt>
 *
 * @param localCacheManager
 *          The localCacheManager to set.
 * @uml.property name="localCacheManager"
 */
public void setLocalCacheManager(
    LocalCacheManager localCacheManager) {
  this.localCacheManager = localCacheManager;
}

/**
 * Getter of the property <tt>cache</tt>
 *
 * @return Returns the cache.
 * @uml.property name="cache"
 */
public Cache getCache() {
  return cache;
}

/**
 * Setter of the property <tt>cache</tt>
 *
 * @param cache
 *          The cache to set.
 * @uml.property name="cache"
 */
public void setCache(Cache cache) {
  this.cache = cache;
}
```

```
/**
 * Getter of the property <tt>localLockLog</tt>
 *
 * @return Returns the localLockLog.
 * @uml.property name="localLockLog"
 */
public LocalLockLog getLocalLockLog() {
  return localLockLog;
}

/**
 * Setter of the property <tt>localLockLog</tt>
 *
 * @param localLockLog
 *          The localLockLog to set.
 * @uml.property name="localLockLog"
 */
public void setLocalLockLog(LocalLockLog localLockLog) {
  this.localLockLog = localLockLog;
}

/**
 * Getter of the property <tt>localLog</tt>
 *
 * @return Returns the localLog.
 * @uml.property name="localLog"
 */
public LocalLog getLocalLog() {
  return localLog;
}

/**
 * Setter of the property <tt>localLog</tt>
 *
 * @param localLog
 *          The localLog to set.
 * @uml.property name="localLog"
 */
public void setLocalLog(LocalLog localLog) {
  this.localLog = localLog;
}

/**
 * Getter of the property <tt>lastDisconnectionTime</tt>
 *
 * @return Returns the lastDisconnectionTime.
 * @uml.property name="lastDisconnectionTime"
 */
```

```
  public long getLastDisconnectionTime() {
    return lastDisconnectionTime;
  }

  /**
   * Setter of the property <tt>lastDisconnectionTime</tt>
   * @param lastDisconnectionTime  The lastDisconnectionTime to set.
   * @uml.property  name="lastDisconnectionTime"
   */
  public void setLastDisconnectionTime(long lastDisconnectionTime) {
    this.lastDisconnectionTime = lastDisconnectionTime;
  }
}
```

A.2.2 Cache

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import dataBase.Resource;

public class Cache {

  /**
   * @uml.property name="size"
   */
  private int size;

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd inverse="cache:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="resources"
   */
  private Collection<Resource> resources;

  public Cache(MobileHost mobileHost, int size) {
    this.mobileHost = mobileHost;
    this.size = size;
    resources = new ArrayList<Resource>();
  }
```

```
/**
 */
public boolean addResource(Resource resource) {
  if (resources.size() < size) {
    resources.add(resource);
    return true;
  }
  return false;
}

/**
 */
public void removeResource(Resource resource) {
  resources.remove(resource);
}

public boolean isResourceCached(Resource resource) {
  for (Iterator<Resource> i = resources.iterator(); i.hasNext();) {
    Resource r = i.next();
    if (r.getId().equals(resource.getId())) {
      return true;
    }
  }
  return false;
}

/**
 * Getter of the property <tt>size</tt>
 *
 * @return Returns the size.
 * @uml.property name="size"
 */
public int getSize() {
  return size;
}

/**
 * Setter of the property <tt>size</tt>
 *
 * @param size
 *          The size to set.
 * @uml.property name="size"
 */
public void setSize(int size) {
  this.size = size;
}

/**
```

```
   * Getter of the property <tt>mobileHost</tt>
   *
   * @return Returns the mobileHost.
   * @uml.property name="mobileHost"
   */
  public MobileHost getMobileHost() {
    return mobileHost;
  }

  /**
   * Setter of the property <tt>mobileHost</tt>
   *
   * @param mobileHost
   *          The mobileHost to set.
   * @uml.property name="mobileHost"
   */
  public void setMobileHost(MobileHost mobileHost) {
    this.mobileHost = mobileHost;
  }

  public Collection<Resource> getResources() {
    return resources;
  }
}
```

A.2.3 LocalCacheManager

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import transaction.Transaction;

import dataBase.Resource;

public class LocalCacheManager {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd
    inverse="localCacheManager:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
```

```
 * @uml.property name="cache"
 */
private Cache cache;

/**
 * @uml.property name="cachedResources"
 */
private Collection<cachedResource> cachedResources;

public LocalCacheManager(MobileHost mobileHost, Cache cache) {
  this.mobileHost = mobileHost;
  this.cache = cache;
  cachedResources = new ArrayList<cachedResource>();
}

public class cachedResource {
  private Resource resource;

  private Collection<Transaction> transactions;

  public cachedResource(Resource resource) {
    this.resource = resource;
    transactions = new ArrayList<Transaction>();
  }

  public Resource getResource() {
    return resource;
  }

  public Collection<Transaction> getTransactions() {
    return transactions;
  }
}

/**
 */
public void addResource(Resource resource, Transaction transaction) {
  boolean resourceAlreadyCached = false;
  for (Iterator<cachedResource> i = cachedResources.iterator(); i
      .hasNext();) {
    cachedResource cr = i.next();
    if (cr.getResource().getId().equals(resource.getId())) {
      resourceAlreadyCached = true;
      boolean transactionAlreadyNeedsResource = false;
      for (Iterator<Transaction> j = cr.getTransactions()
          .iterator(); j.hasNext();) {
        Transaction t = j.next();
        if (t.getId().equals(transaction.getId())) {
          transactionAlreadyNeedsResource = true;
```

```
        }
      }
      if (!transactionAlreadyNeedsResource) {
        cr.getTransactions().add(transaction);
      }
    }
  }
  if (!resourceAlreadyCached) {
    if (cachedResources.size() < cache.getSize()) {
      cachedResource cachedR = new cachedResource(resource);
      cachedResources.add(cachedR);
      cachedR.getTransactions().add(transaction);
      cache.addResource(resource);
    } else {
    }
  }
}

/**
 */
public void removeResource(Resource resource,
    Transaction transaction) {
  cachedResource cachedResourceToRemove = null;
  for (Iterator<cachedResource> i = cachedResources.iterator(); i
      .hasNext();) {
    cachedResource cr = i.next();
    if (cr.getResource().getId().equals(resource.getId())) {
      if (cr.getTransactions().size() == 1) {
        if (cr.getTransactions().contains(transaction)) {
          cachedResourceToRemove = cr;
          cache.removeResource(resource);
        }
      } else {
        cr.getTransactions().remove(transaction);
      }
    }
  }
  if (cachedResourceToRemove != null) {
    cachedResources.remove(cachedResourceToRemove);
  }
}

/**
 * Getter of the property <tt>mobileHost</tt>
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
```

```
  return mobileHost;
}

/**
 * Setter of the property <tt>mobileHost</tt>
 *
 * @param mobileHost
 *          The mobileHost to set.
 * @uml.property name="mobileHost"
 */
public void setMobileHost(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
}

/**
 * Getter of the property <tt>cache</tt>
 *
 * @return Returns the cache.
 * @uml.property name="cache"
 */
public Cache getCache() {
  return cache;
}

/**
 * Setter of the property <tt>cache</tt>
 *
 * @param cache
 *          The cache to set.
 * @uml.property name="cache"
 */
public void setCache(Cache cache) {
  this.cache = cache;
}

/**
 * Getter of the property <tt>cachedResources</tt>
 *
 * @return Returns the cachedResources.
 * @uml.property name="cachedResources"
 */
public Collection<cachedResource> getCachedResources() {
  return cachedResources;
}

/**
 * Setter of the property <tt>cachedResources</tt>
 *
 * @param cachedResources
```

```
   *          The cachedResources to set.
   * @uml.property name="cachedResources"
   */
  public void setCachedResources(
      Collection<cachedResource> cachedResources) {
    this.cachedResources = cachedResources;
  }
}
```

A.2.4 LocalLockLog

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class LocalLockLog {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd inverse="localLockLog:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="localLockLogRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="localLockLog:mobileHost.LocalLockLogRecord"
   */
  private Collection<LocalLockLogRecord> localLockLogRecord;

  /**
   * @uml.property name="logId"
   */
  private int logId;

  public LocalLockLog(MobileHost mobileHost) {
    this.mobileHost = mobileHost;
    localLockLogRecord = new ArrayList<LocalLockLogRecord>();
    logId = 0;

  }

  /**
   */
```

```
public void addLocalLockLogRecord(long time, String transactionId,
    String resourceId, String grantedLockType) {
  localLockLogRecord.add(new LocalLockLogRecord(this, new String(
      "" + logId), new String("" + time), transactionId,
      resourceId, grantedLockType));
  logId = logId + 1;
}

public void printLocalLockLog() {
  System.out.println("");
  System.out.println("Mobilehost " + mobileHost.getId()
      + "s locklog:");
  System.out.println("");
  System.out.println("Time TransactionId ResourceId grantedLock");
  System.out.println("");
  for (Iterator<LocalLockLogRecord> i = localLockLogRecord
      .iterator(); i.hasNext();) {
    LocalLockLogRecord localLockLogRecord = i.next();
    System.out.println(localLockLogRecord.getTime() + " "
        + localLockLogRecord.getTransactionId() + " "
        + localLockLogRecord.getResourceId() + " "
        + localLockLogRecord.getGrantedLockType());
  }
}

/**
 * Getter of the property <tt>mobileHost</tt>
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
  return mobileHost;
}

/**
 * Setter of the property <tt>mobileHost</tt>
 *
 * @param mobileHost
 *          The mobileHost to set.
 * @uml.property name="mobileHost"
 */
public void setMobileHost(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
}

/**
 * Getter of the property <tt>localLockLogRecord</tt>
```

```
   *
   * @return Returns the localLockLogRecord.
   * @uml.property name="localLockLogRecord"
   */
  public Collection<LocalLockLogRecord> getLocalLockLogRecord() {
    return localLockLogRecord;
  }

  /**
   * Setter of the property <tt>localLockLogRecord</tt>
   *
   * @param localLockLogRecord
   *          The localLockLogRecord to set.
   * @uml.property name="localLockLogRecord"
   */
  public void setLocalLockLogRecord(
      Collection<LocalLockLogRecord> localLockLogRecord) {
    this.localLockLogRecord = localLockLogRecord;
  }

  /**
   * Getter of the property <tt>logId</tt>
   *
   * @return Returns the logId.
   * @uml.property name="logId"
   */
  public int getLogId() {
    return logId;
  }

  /**
   * Setter of the property <tt>logId</tt>
   * @param logId  The logId to set.
   * @uml.property  name="logId"
   */
  public void setLogId(int logId) {
    this.logId = logId;
  }
}
```

A.2.5 LocalLockLogRecord

```
package mobileHost;

public class LocalLockLogRecord {

  /**
```

```
 * @uml.property name="id"
 */
private String id;

/**
 * @uml.property name="time"
 */
private String time;

/**
 * @uml.property name="transactionId"
 */
private String transactionId;

/**
 * @uml.property name="resourceId"
 */
private String resourceId;

/**
 * @uml.property name="grantedLockType"
 */
private String grantedLockType;

/**
 * @uml.property name="localLockLog"
 * @uml.associationEnd
  inverse="localLockLogRecord:mobileHost.LocalLockLog"
 */
private LocalLockLog localLockLog;

public LocalLockLogRecord(LocalLockLog localLockLog, String id,
    String time, String transactionId, String resourceId,
    String grantedLockType) {
  this.localLockLog = localLockLog;
  this.id = id;
  this.time = time;
  this.transactionId = transactionId;
  this.resourceId = resourceId;
  this.grantedLockType = grantedLockType;
}

/**
 * Getter of the property <tt>localLockLog</tt>
 *
 * @return Returns the localLockLog.
 * @uml.property name="localLockLog"
 */
public LocalLockLog getLocalLockLog() {
```

```
  return localLockLog;
}

/**
 * Setter of the property <tt>localLockLog</tt>
 *
 * @param localLockLog
 *          The localLockLog to set.
 * @uml.property name="localLockLog"
 */
public void setLocalLockLog(LocalLockLog localLockLog) {
  this.localLockLog = localLockLog;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>time</tt>
 *
 * @return Returns the time.
 * @uml.property name="time"
 */
public String getTime() {
  return time;
}

/**
 * Setter of the property <tt>time</tt>
 *
 * @param time
```

```
     *          The time to set.
     * @uml.property name="time"
     */
    public void setTime(String time) {
      this.time = time;
    }

    /**
     * Getter of the property <tt>transactionId</tt>
     *
     * @return Returns the transactionId.
     * @uml.property name="transactionId"
     */
    public String getTransactionId() {
      return transactionId;
    }

    /**
     * Setter of the property <tt>transactionId</tt>
     *
     * @param transactionId
     *          The transactionId to set.
     * @uml.property name="transactionId"
     */
    public void setTransactionId(String transactionId) {
      this.transactionId = transactionId;
    }

    /**
     * Getter of the property <tt>resourceId</tt>
     *
     * @return Returns the resourceId.
     * @uml.property name="resourceId"
     */
    public String getResourceId() {
      return resourceId;
    }

    /**
     * Setter of the property <tt>resourceId</tt>
     *
     * @param resourceId
     *          The resourceId to set.
     * @uml.property name="resourceId"
     */
    public void setResourceId(String resourceId) {
      this.resourceId = resourceId;
    }
```

```
  /**
   * Getter of the property <tt>grantedLockType</tt>
   *
   * @return Returns the grantedLockType.
   * @uml.property name="grantedLockType"
   */
  public String getGrantedLockType() {
    return grantedLockType;
  }

  /**
   * Setter of the property <tt>grantedLockType</tt>
   *
   * @param requstingLockType
   *          The grantedLockType to set.
   * @uml.property name="grantedLockType"
   */
  public void setGrantedLockType(String grantedLockType) {
    this.grantedLockType = grantedLockType;
  }
}
```

A.2.6 LocalLog

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class LocalLog {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd inverse="localLog:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="localLogRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
   *                     inverse="localLog:mobileHost.LocalLogRecord"
   */
  private Collection<LocalLogRecord> localLogRecord;

  /**
   * @uml.property name="localLogId"
```

```
 */
private int localLogId;

public LocalLog(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
  localLogRecord = new ArrayList<LocalLogRecord>();
  localLogId = 0;
}

/**
 */
public void addLocalLogRecord(String transactionId, String action,
    String resourceId, String value, String newValue) {
  localLogRecord.add(new LocalLogRecord(this, new String(""
      + localLogId), new String("" + mobileHost.getLocalTime()),
      transactionId, action, resourceId, value, newValue));
  localLogId = localLogId + 1;
}

/**
 */
public Collection<LocalLogRecord>
  getLocalLogRecordsFromLastDisconnection(
    long lastDisconnectionTime) {
  Collection<LocalLogRecord> returnList = new
    ArrayList<LocalLogRecord>();
  for (Iterator<LocalLogRecord> i = localLogRecord.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRec = i.next();
    int time = new Integer(localLogRec.getLocalTime());
    if (time > lastDisconnectionTime) {
      returnList.add(localLogRec);
    }
  }
  return returnList;
}

/**
 */
public void printLocalLog() {
  System.out.println("");
  System.out.println("Mobilehost " + mobileHost.getId()
      + "s local log:");
  System.out.println("");
  System.out
      .println("Localtime TransactionId action ResourceId  value
        newValue");
  System.out.println("");
  for (Iterator<LocalLogRecord> i = localLogRecord.iterator(); i
```

```
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    System.out.println(localLogRecord.getLocalTime() + " "
        + localLogRecord.getTransactionId() + " "
        + localLogRecord.getAction() + " "
        + localLogRecord.getResourceId() + " "
        + localLogRecord.getValue() + " "
        + localLogRecord.getNewValue());
  }
}

/**
 * Getter of the property <tt>mobileHost</tt>
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
  return mobileHost;
}

/**
 * Setter of the property <tt>mobileHost</tt>
 *
 * @param mobileHost
 *          The mobileHost to set.
 * @uml.property name="mobileHost"
 */
public void setMobileHost(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
}

/**
 * Getter of the property <tt>localLogRecord</tt>
 *
 * @return Returns the localLogRecord.
 * @uml.property name="localLogRecord"
 */
public Collection<LocalLogRecord> getLocalLogRecord() {
  return localLogRecord;
}

/**
 * Setter of the property <tt>localLogRecord</tt>
 *
 * @param localLogRecord
 *          The localLogRecord to set.
 * @uml.property name="localLogRecord"
 */
```

```
  public void setLocalLogRecord(
      Collection<LocalLogRecord> localLogRecord) {
    this.localLogRecord = localLogRecord;
  }

  /**
   * Getter of the property <tt>logId</tt>
   *
   * @return Returns the logId.
   * @uml.property name="localLogId"
   */
  public int getLocalLogId() {
    return localLogId;
  }

  /**
   * Setter of the property <tt>logId</tt>
   * @param logId  The logId to set.
   * @uml.property  name="localLogId"
   */
  public void setLocalLogId(int localLogId) {
    this.localLogId = localLogId;
  }
}
```

A.2.7 LocalLogRecord

```
package mobileHost;

public class LocalLogRecord {

  /**
   * @uml.property name="localLog"
   * @uml.associationEnd inverse="localLogRecord:mobileHost.LocalLog"
   */
  private LocalLog localLog;

  /**
   * @uml.property name="localLogId"
   */
  private String localLogId;

  /**
   * @uml.property name="localTime"
   */
  private String localTime;
```

```
/**
 * @uml.property name="transactionId"
 */
private String transactionId;

/**
 * @uml.property name="action"
 */
private String action;

/**
 * @uml.property name="resourceId"
 */
private String resourceId;

/**
 * @uml.property name="value"
 */
private String value;

/**
 * @uml.property name="newValue"
 */
private String newValue;

public LocalLogRecord(LocalLog localLog, String localLogId,
    String localTime, String transactionId, String action,
    String resourceId, String value, String newValue) {
  this.localLog = localLog;
  this.localLogId = localLogId;
  this.localTime = localTime;
  this.transactionId = transactionId;
  this.action = action;
  this.resourceId = resourceId;
  this.value = value;
  this.newValue = newValue;
}

/**
 * Getter of the property <tt>localLog</tt>
 *
 * @return Returns the localLog.
 * @uml.property name="localLog"
 */
public LocalLog getLocalLog() {
  return localLog;
}

/**
```

```
 * Setter of the property <tt>localLog</tt>
 *
 * @param localLog
 *          The localLog to set.
 * @uml.property name="localLog"
 */
public void setLocalLog(LocalLog localLog) {
  this.localLog = localLog;
}

/**
 * Getter of the property <tt>localLogId</tt>
 *
 * @return Returns the localLogId.
 * @uml.property name="localLogId"
 */
public String getLocalLogId() {
  return localLogId;
}

/**
 * Setter of the property <tt>localLogId</tt>
 *
 * @param localLogId
 *          The localLogId to set.
 * @uml.property name="localLogId"
 */
public void setLocalLogId(String localLogId) {
  this.localLogId = localLogId;
}

/**
 * Getter of the property <tt>localTime</tt>
 *
 * @return Returns the localTime.
 * @uml.property name="localTime"
 */
public String getLocalTime() {
  return localTime;
}

/**
 * Setter of the property <tt>localTime</tt>
 *
 * @param localTime
 *          The localTime to set.
 * @uml.property name="localTime"
 */
public void setLocalTime(String localTime) {
```

```
  this.localTime = localTime;
}

/**
 * Getter of the property <tt>transactionId</tt>
 *
 * @return Returns the transactionId.
 * @uml.property name="transactionId"
 */
public String getTransactionId() {
  return transactionId;
}

/**
 * Setter of the property <tt>transactionId</tt>
 *
 * @param transactionId
 *          The transactionId to set.
 * @uml.property name="transactionId"
 */
public void setTransactionId(String transactionId) {
  this.transactionId = transactionId;
}

/**
 * Getter of the property <tt>action</tt>
 *
 * @return Returns the action.
 * @uml.property name="action"
 */
public String getAction() {
  return action;
}

/**
 * Setter of the property <tt>action</tt>
 *
 * @param action
 *          The action to set.
 * @uml.property name="action"
 */
public void setAction(String action) {
  this.action = action;
}

/**
 * Getter of the property <tt>resourceId</tt>
 *
 * @return Returns the resourceId.
```

```
 * @uml.property name="resourceId"
 */
public String getResourceId() {
  return resourceId;
}

/**
 * Setter of the property <tt>resourceId</tt>
 *
 * @param resourceId
 *          The resourceId to set.
 * @uml.property name="resourceId"
 */
public void setResourceId(String resourceId) {
  this.resourceId = resourceId;
}

/**
 * Getter of the property <tt>value</tt>
 *
 * @return Returns the value.
 * @uml.property name="value"
 */
public String getValue() {
  return value;
}

/**
 * Setter of the property <tt>value</tt>
 *
 * @param value
 *          The value to set.
 * @uml.property name="value"
 */
public void setValue(String value) {
  this.value = value;
}

/**
 * Getter of the property <tt>newValue</tt>
 *
 * @return Returns the newValue.
 * @uml.property name="newValue"
 */
public String getNewValue() {
  return newValue;
}

/**
```

```
   * Setter of the property <tt>newValue</tt>
   *
   * @param newValue
   *          The newValue to set.
   * @uml.property name="newValue"
   */
  public void setNewValue(String newValue) {
    this.newValue = newValue;
  }
}
```

A.2.8 LocalTransactionManager

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import dataBase.Resource;

import lock.Lock;

import operation.AbortOperation;
import operation.CommitOperation;
import operation.ReadOperation;
import operation.StartOperation;
import operation.WriteOperation;
import transaction.Transaction;

public class LocalTransactionManager {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd
    inverse="localTransactionManager:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="transaction"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="localTransactionManager:transaction.Transaction"
   */
  private Collection<Transaction> transaction;
```

```
public LocalTransactionManager(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
  transaction = new ArrayList<Transaction>();
}

/**
 */
public Transaction getTransaction(String transactionId) {
  for (Iterator<Transaction> i = transaction.iterator(); i
      .hasNext();) {
    Transaction t = i.next();
    if (t.getId().equals(transactionId)) {
      return t;
    }
  }
  return null;
}

/**
 */
public void executeStartOperation(String transactionId) {
  Transaction t = new Transaction(transactionId, getMobileHost(),
      "ACTIVE");
  StartOperation startOperation = new StartOperation(t);
  t.getOperation().add(startOperation);
  transaction.add(t);
  if (mobileHost.getConnectionStatus() == true) {
    mobileHost.getMobileSupportStation().getDatabaseServer()
        .initiateTransaction(t);
  } else {
    startTransactionOffline(t);
  }
}

/**
 */
public void executeReadOperation(String transactionId,
    String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    ReadOperation readOperation = new ReadOperation(t, r);
    t.getOperation().add(readOperation);
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
```

```
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("RONLOCK")
        && mobileHost.getConnectionStatus() == true) {
      mobileHost.getMobileSupportStation().getDatabaseServer()
          .read(readOperation);
    } else if (lockType.equals("WONLOCK")
        && mobileHost.getConnectionStatus() == true) {
      mobileHost.getMobileSupportStation().getDatabaseServer()
          .read(readOperation);
    } else if (lockType.equals("WOFFLOCK")
        && mobileHost.getConnectionStatus() == false) {
      if (mobileHost.getCache().isResourceCached(r)) {
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else if (lockType.equals("WIOFFLOCK")
        && mobileHost.getConnectionStatus() == false) {
      if (mobileHost.getCache().isResourceCached(r)) {
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else if (lockType.equals("RONLOCK")
        && mobileHost.getConnectionStatus() == false) {
      if (mobileHost.getCache().isResourceCached(r)) {
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void executeWriteOperation(String transactionId,
    String resourceId, int newValue) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    WriteOperation writeOperation = new WriteOperation(t, r,
        newValue);
    t.getOperation().add(writeOperation);
```

```
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("WONLOCK")
        && mobileHost.getConnectionStatus() == true) {
      mobileHost.getMobileSupportStation().getDatabaseServer()
          .write(writeOperation);
    } else if (lockType.equals("WOFFLOCK")
        && mobileHost.getConnectionStatus() == false) {
      if (mobileHost.getCache().isResourceCached(r)) {
        writeOffline(writeOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void executeCommitOperation(String transactionId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    CommitOperation commitOperation = new CommitOperation(t);
    t.getOperation().add(commitOperation);
    if (mobileHost.getConnectionStatus() == true) {
      mobileHost.getMobileSupportStation().getDatabaseServer()
          .commit(commitOperation);
    } else {
      commitOffline(commitOperation);
    }
  }
}

/**
 */
public void executeAbortOperation(String transactionId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    AbortOperation abortOperation = new AbortOperation(t);
    t.getOperation().add(abortOperation);
    if (mobileHost.getConnectionStatus() == true) {
```

```
      mobileHost.getMobileSupportStation().getDatabaseServer()
          .abort(abortOperation);
    } else {
      abortOffline(abortOperation);
    }
  }
}

/**
 */
public void requestROnLock(String transactionId, String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    if (mobileHost.getConnectionStatus() == true) {
      if (mobileHost.getMobileSupportStation().getDatabaseServer()
          .requestROnLock(r, t)) {
        mobileHost.getLocalCacheManager().addResource(r, t);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void requestWOnLock(String transactionId, String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    if (mobileHost.getConnectionStatus() == true) {
      if (mobileHost.getMobileSupportStation().getDatabaseServer()
          .requestWOnLock(r, t)) {
        mobileHost.getLocalCacheManager().addResource(r, t);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
```

```
 */
public void requestWIOffLock(String transactionId,
    String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    if (mobileHost.getConnectionStatus() == true) {
      if (mobileHost.getMobileSupportStation().getDatabaseServer()
          .requestWIOffLock(r, t)) {
        long timeWhenRequestWasGranted = mobileHost
            .getMobileSupportStation().getDatabaseServer()
            .getLockLog().getTimeWhenLastLockWasGranted();
        mobileHost.getLocalLockLog().addLocalLockLogRecord(
            timeWhenRequestWasGranted, transactionId, resourceId,
            "WIOFFLOCK");
        mobileHost.getLocalCacheManager().addResource(r, t);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void requestWOffLock(String transactionId, String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    if (mobileHost.getConnectionStatus() == true) {
      if (mobileHost.getMobileSupportStation().getDatabaseServer()
          .requestWOffLock(r, t)) {
        mobileHost.getLocalCacheManager().addResource(r, t);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void startTransactionOffline(Transaction transaction) {
```

```
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "STARTOFFLINE", "", "", "");
}

/**
 */
public int readOffline(ReadOperation readOperation) {
  Transaction transaction = readOperation.getTransaction();
  Resource resource = readOperation.getResource();
  int readValue;
  if (transaction.isResourceWritten(resource)) {
    readValue = transaction.getNewestWrittenValue(resource,
        readOperation);
  } else {
    readValue = resource.getValue();
  }
  readOperation.setReadValue(readValue);
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "READOFFLINE", resource.getId(), new String("" + readValue),
      "");
  return readValue;
}

/**
 */
public void writeOffline(WriteOperation writeOperation) {
  Transaction transaction = writeOperation.getTransaction();
  Resource resource = writeOperation.getResource();
  int newValue = writeOperation.getNewValue();
  transaction.getWrittenResources().add(resource);
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "WRITEOFFLINE", resource.getId(),
      new String("" + resource.getValue()),
      new String("" + newValue));
}

/**
 */
public void commitOffline(CommitOperation commitOperation) {
  Transaction t = commitOperation.getTransaction();
  t.setStatus("COMMITEDOFFLINE");
  mobileHost.getLocalLog().addLocalLogRecord(t.getId(),
      "COMMITOFFLINE", "", "", "");
}

/**
 */
public void abortOffline(AbortOperation abortOperation) {
  Transaction t = abortOperation.getTransaction();
```

```
    t.setStatus("ABORTEDOFFLINE");
    mobileHost.getLocalLog().addLocalLogRecord(t.getId(),
        "ABORTOFFLINE", "", "", "");
  }

  /**
   * Getter of the property <tt>mobileHost</tt>
   *
   * @return Returns the mobileHost.
   * @uml.property name="mobileHost"
   */
  public MobileHost getMobileHost() {
    return mobileHost;
  }

  /**
   * Setter of the property <tt>mobileHost</tt>
   *
   * @param mobileHost
   *          The mobileHost to set.
   * @uml.property name="mobileHost"
   */
  public void setMobileHost(MobileHost mobileHost) {
    this.mobileHost = mobileHost;
  }

  /**
   * Getter of the property <tt>transaction</tt>
   *
   * @return Returns the transaction.
   * @uml.property name="transaction"
   */
  public Collection<Transaction> getTransaction() {
    return transaction;
  }

  /**
   * Setter of the property <tt>transaction</tt>
   *
   * @param transaction
   *          The transaction to set.
   * @uml.property name="transaction"
   */
  public void setTransaction(Collection<Transaction> transaction) {
    this.transaction = transaction;
  }
}
```

A.3 Modulen *transaction*

Klassene i modulen *transaction* er:

- `Transaction`

A.3.1 `Transaction`

```
package transaction;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import dataBase.*;
import operation.*;
import lock.*;
import mobileHost.*;

public class Transaction {

  /**
   * @uml.property name="owner"
   */
  private MobileHost owner;

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="status"
   */
  private String status = "";

  /**
   * @uml.property name="locks"
   */
  private Collection<Lock> locks;

  /**
   * @uml.property name="localTransactionManager"
   * @uml.associationEnd
     inverse="transaction:mobileHost.LocalTransactionManager"
   */
```

```
private LocalTransactionManager localTransactionManager;

/**
 * @uml.property name="operation"
 * @uml.associationEnd multiplicity="(0 -1)"
 *                     inverse="transaction:operation.Operation"
 */
private Collection<Operation> operation;

private Collection<Resource> writtenResources;

public Transaction(String id, MobileHost owner, String status) {
  this.id = id;
  this.owner = owner;
  this.status = status;
  locks = new ArrayList<Lock>();
  operation = new ArrayList<Operation>();
  writtenResources = new ArrayList<Resource>();
}

public boolean isResourceWritten(Resource resource) {
  for (Iterator<Resource> i = writtenResources.iterator(); i
      .hasNext();) {
    Resource r = i.next();
    if (r.getId().equals(resource.getId())) {
      return true;
    }
  }
  return false;
}

public int getNewestWrittenValue(Resource resource,
    ReadOperation ro) {
  int returnValue = 0;
  for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      if (wo.getResource().getId().equals(resource.getId())) {
        returnValue = wo.getNewValue();
      }
    }
    if (op.equals(ro)) {
      break;
    }
  }
  return returnValue;
}
```

```
/**
 */
public Operation getOperation(String type) {
  if (type.equals("COMMITOPERATION")) {
    for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
      Operation op = i.next();
      if (op.getOperationType().equals("COMMITOPERATION")) {
        return op;
      }
    }
  }
  if (type.equals("ABORTOPERATION")) {
    for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
      Operation op = i.next();
      if (op.getOperationType().equals("ABORTOPERATION")) {
        return op;
      }
    }
  }
  return null;
}

/**
 * Getter of the property <tt>owner</tt>
 *
 * @return Returns the owner.
 * @uml.property name="owner"
 */
public MobileHost getOwner() {
  return owner;
}

/**
 * Setter of the property <tt>owner</tt>
 *
 * @param owner
 *          The owner to set.
 * @uml.property name="owner"
 */
public void setOwner(MobileHost owner) {
  this.owner = owner;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
```

```
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>status</tt>
 *
 * @return Returns the status.
 * @uml.property name="status"
 */
public String getStatus() {
  return status;
}

/**
 * Setter of the property <tt>status</tt>
 *
 * @param status
 *          The status to set.
 * @uml.property name="status"
 */
public void setStatus(String status) {
  this.status = status;
}

/**
 * Getter of the property <tt>locks</tt>
 *
 * @return Returns the locks.
 * @uml.property name="locks"
 */
public Collection<Lock> getLocks() {
  return locks;
}

/**
 * Setter of the property <tt>locks</tt>
 *
```

```
 * @param locks
 *          The locks to set.
 * @uml.property name="locks"
 */
public void setLocks(Collection<Lock> locks) {
  this.locks = locks;
}

/**
 * Getter of the property <tt>localTransactionManager</tt>
 *
 * @return Returns the localTransactionManager.
 * @uml.property name="localTransactionManager"
 */
public LocalTransactionManager getLocalTransactionManager() {
  return localTransactionManager;
}

/**
 * Setter of the property <tt>localTransactionManager</tt>
 *
 * @param localTransactionManager
 *          The localTransactionManager to set.
 * @uml.property name="localTransactionManager"
 */
public void setLocalTransactionManager(
    LocalTransactionManager localTransactionManager) {
  this.localTransactionManager = localTransactionManager;
}

/**
 * Getter of the property <tt>operation</tt>
 *
 * @return Returns the operation.
 * @uml.property name="operation"
 */
public Collection<Operation> getOperation() {
  return operation;
}

/**
 * Setter of the property <tt>operation</tt>
 *
 * @param operation
 *          The operation to set.
 * @uml.property name="operation"
 */
public void setOperation(Collection<Operation> operation) {
  this.operation = operation;
```

```
  }

  public Collection<Resource> getWrittenResources() {
    return writtenResources;
  }
}
```

A.4 Modulen *lock*

Klassene i modulen *lock* er:

- `Lock`
- `ROnLock`
- `WIOffLock`
- `WOffLock`
- `WOnLock`

A.4.1 `Lock`

```
package lock;

import transaction.Transaction;
import dataBase.Resource;

public interface Lock {

  public String getLockType();

  public Transaction getTransaction();

  public void setTransaction(Transaction transaction);

  public Resource getResource();

  public void setResource(Resource resource);

  public String getLockLogId();

  public void setLockLogId(String lockLogId);

}
```

A.4.2 `ROnLock`

```
package lock;

import transaction.Transaction;
```

```
import dataBase.Resource;

public class ROnLock implements Lock {

  /**
   * @uml.property name="transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="grantedWOffLock"
   */
  private boolean grantedWOffLock;

  /**
   * @uml.property name="lockLogId"
   */
  private String lockLogId;

  public ROnLock(Transaction transaction, Resource resource) {
    this.resource = resource;
    this.transaction = transaction;
    grantedWOffLock = false;
    lockLogId = "";
  }

  public String getLockType() {
    return "RONLOCK";
  }

  public Transaction getTransaction() {
    return transaction;
  }

  public void setTransaction(Transaction transaction) {
    this.transaction = transaction;
  }

  /**
   * Getter of the property <tt>resource</tt>
   *
   * @return Returns the resource.
   * @uml.property name="resource"
   */
```

```
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>grantedWOffLock</tt>
 *
 * @return Returns the grantedWOffLock.
 * @uml.property name="grantedWOffLock"
 */
public boolean getGrantedWOffLock() {
  return grantedWOffLock;
}

/**
 * Setter of the property <tt>grantedWOffLock</tt>
 *
 * @param grantedWOffLock
 *          The grantedWOffLock to set.
 * @uml.property name="grantedWOffLock"
 */
public void setGrantedWOffLock(boolean grantedWOffLock) {
  this.grantedWOffLock = grantedWOffLock;
}

/**
 * Getter of the property <tt>lockLogId</tt>
 *
 * @return Returns the lockLogId.
 * @uml.property name="lockLogId"
 */
public String getLockLogId() {
  return lockLogId;
}

/**
 * Setter of the property <tt>lockLogId</tt>
 *
```

```
   * @param lockLogId
   *          The lockLogId to set.
   * @uml.property name="lockLogId"
   */
  public void setLockLogId(String lockLogId) {
    this.lockLogId = lockLogId;
  }
}
```

A.4.3 WIOffLock

```
package lock;

import transaction.Transaction;
import dataBase.Resource;

public class WIOffLock implements Lock {

  /**
   * @uml.property name="transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="lockLogId"
   */
  private String lockLogId;

  public WIOffLock(Transaction transaction, Resource resource) {
    this.transaction = transaction;
    this.resource = resource;
    lockLogId = "";
  }

  public String getLockType() {
    return "WIOFFLOCK";
  }

  public Transaction getTransaction() {
    return transaction;
  }
```

```
  public void setTransaction(Transaction transaction) {
    this.transaction = transaction;
  }

  /**
   * Getter of the property <tt>resource</tt>
   *
   * @return Returns the resource.
   * @uml.property name="resource"
   */
  public Resource getResource() {
    return resource;
  }

  /**
   * Setter of the property <tt>resource</tt>
   *
   * @param resource
   *          The resource to set.
   * @uml.property name="resource"
   */
  public void setResource(Resource resource) {
    this.resource = resource;
  }

  /**
   * Getter of the property <tt>lockLogId</tt>
   *
   * @return Returns the lockLogId.
   * @uml.property name="lockLogId"
   */
  public String getLockLogId() {
    return lockLogId;
  }

  /**
   * Setter of the property <tt>lockLogId</tt>
   *
   * @param lockLogId
   *          The lockLogId to set.
   * @uml.property name="lockLogId"
   */
  public void setLockLogId(String lockLogId) {
    this.lockLogId = lockLogId;
  }

}
```

A.4.4 WOffLock

```
package lock;

import transaction.Transaction;
import dataBase.Resource;

public class WOffLock implements Lock {

  /**
   * @uml.property name="transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="lockLogId"
   */
  private String lockLogId;

  public WOffLock(Transaction transaction, Resource resource) {
    this.transaction = transaction;
    this.resource = resource;
    lockLogId = "";
  }

  public String getLockType() {
    return "WOFFLOCK";
  }

  public Transaction getTransaction() {
    return transaction;
  }

  public void setTransaction(Transaction transaction) {
    this.transaction = transaction;
  }

  /**
   * Getter of the property <tt>resource</tt>
   *
   * @return Returns the resource.
   * @uml.property name="resource"
   */
```

```
  public Resource getResource() {
    return resource;
  }

  /**
   * Setter of the property <tt>resource</tt>
   *
   * @param resource
   *          The resource to set.
   * @uml.property name="resource"
   */
  public void setResource(Resource resource) {
    this.resource = resource;
  }

  /**
   * Getter of the property <tt>lockLogId</tt>
   *
   * @return Returns the lockLogId.
   * @uml.property name="lockLogId"
   */
  public String getLockLogId() {
    return lockLogId;
  }

  /**
   * Setter of the property <tt>lockLogId</tt>
   *
   * @param lockLogId
   *          The lockLogId to set.
   * @uml.property name="lockLogId"
   */
  public void setLockLogId(String lockLogId) {
    this.lockLogId = lockLogId;
  }
}
```

A.4.5 WOnLock

```
package lock;

import transaction.Transaction;
import dataBase.Resource;

public class WOnLock implements Lock {

  /**
```

```
 * @uml.property name="transaction"
 */
private Transaction transaction;

/**
 * @uml.property name="resource"
 */
private Resource resource;

/**
 * @uml.property name="lockLogId"
 */
private String lockLogId;

public WOnLock(Transaction transaction, Resource resource) {
  this.transaction = transaction;
  this.resource = resource;
  lockLogId = "";
}

public String getLockType() {
  return "WONLOCK";
}

public Transaction getTransaction() {
  return transaction;
}

public void setTransaction(Transaction transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
```

```
  public void setResource(Resource resource) {
    this.resource = resource;
  }

  /**
   * Getter of the property <tt>lockLogId</tt>
   *
   * @return Returns the lockLogId.
   * @uml.property name="lockLogId"
   */
  public String getLockLogId() {
    return lockLogId;
  }

  /**
   * Setter of the property <tt>lockLogId</tt>
   *
   * @param lockLogId
   *          The lockLogId to set.
   * @uml.property name="lockLogId"
   */
  public void setLockLogId(String lockLogId) {
    this.lockLogId = lockLogId;
  }

}
```

A.5 Modulen *operation*

Klassene i modulen *operation* er:

- `Operation`

A.5.1 `Operation`

```
package operation;

public interface Operation {

  /**
   */
  public abstract String getOperationType();

}
```

A.5.2 `AbortOperation`

```
package operation;

import transaction.Transaction;

public class AbortOperation implements Operation {
  private Transaction transaction;

  public AbortOperation(Transaction transaction) {
    this.transaction = transaction;
  }

  public String getOperationType() {
    return "ABORTOPERATION";
  }

  public Transaction getTransaction() {
    return transaction;
  }
}
```

A.5.3 `CommitOperation`

```
package operation;
```

```
import transaction.Transaction;

public class CommitOperation implements Operation {

  private Transaction transaction;

  public CommitOperation(Transaction transaction) {
    this.transaction = transaction;
  }

  public String getOperationType() {
    return "COMMITOPERATION";
  }

  public Transaction getTransaction() {
    return transaction;
  }

}
```

A.5.4 ReadOperation

```
package operation;

import transaction.Transaction;
import dataBase.Resource;

public class ReadOperation implements Operation {

  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="readValue"
   */
  private Integer readValue;

  /**
   * Getter of the property <tt>readValue</tt>
   *
   * @return Returns the readValue.
   * @uml.property name="readValue"
```

```
 */

public ReadOperation(Transaction transaction, Resource resource) {
  this.resource = resource;
  this.transaction = transaction;
  readValue = null;
}

public String getOperationType() {
  return "READOPERATION";
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

public Transaction getTransaction() {
  return transaction;
}

public int getReadValue() {
  return readValue.intValue();
}

/**
 * Setter of the property <tt>readValue</tt>
 *
 * @param readValue
 *          The readValue to set.
 * @uml.property name="readValue"
 */
public void setReadValue(int readValue) {
```

```
    this.readValue = new Integer(readValue);
  }
}
```

A.5.5 StartOperation

```
package operation;

import transaction.Transaction;

public class StartOperation implements Operation {
  private Transaction transaction;

  public StartOperation(Transaction transaction) {
    this.transaction = transaction;
  }

  public String getOperationType() {
    return "STARTOPERATION";
  }

  public Transaction getTransaction() {
    return transaction;
  }
}
```

A.5.6 WriteOperation

```
package operation;

import transaction.Transaction;
import dataBase.Resource;

public class WriteOperation implements Operation {

  /**
   * @uml.property name="transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
```

```
 * @uml.property name="newValue"
 */
private int newValue;

public WriteOperation(Transaction transaction, Resource resource,
    int newValue) {
  this.transaction = transaction;
  this.resource = resource;
  this.newValue = newValue;
}

public String getOperationType() {
  return "WRITEOPERATION";
}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Transaction getTransaction() {
  return transaction;
}

/**
 * Setter of the property <tt>transaction</tt>
 *
 * @param transaction
 *          The transaction to set.
 * @uml.property name="transaction"
 */
public void setTransaction(Transaction transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
```

```
   * @param resource
   *          The resource to set.
   * @uml.property name="resource"
   */
  public void setResource(Resource resource) {
    this.resource = resource;
  }

  /**
   * Getter of the property <tt>newValue</tt>
   *
   * @return Returns the newValue.
   * @uml.property name="newValue"
   */
  public int getNewValue() {
    return newValue;
  }

  /**
   * Setter of the property <tt>newValue</tt>
   *
   * @param newValue
   *          The newValue to set.
   * @uml.property name="newValue"
   */
  public void setNewValue(int newValue) {
    this.newValue = newValue;
  }
}
```

A.6 Modulen *input*

Klassene i modulen *input* er:

- `InputFileReader`

A.6.1 `InputFileReader`

```
package input;

import java.io.BufferedReader;
import java.io.FileReader;
import java.io.IOException;

public class InputFileReader {

  /**
   * @uml.property name="reader"
   */
  private BufferedReader reader;

  public InputFileReader(String fileName) {
    try {
      reader = new BufferedReader(new FileReader(fileName));
    } catch (IOException ioe) {
      System.err.println("Input file not found.");
      System.exit(1);
    }
  }

  /**
   */
  public String readLine() {
    try {
      String line = reader.readLine();
      if (line != null) {
        line = line.trim();
        if (line.startsWith("#") || line.length() == 0)
          line = readLine();
      }
      return line;
    } catch (IOException ioe) {
      ioe.printStackTrace();
      return null;
    }
```

```
  }

  /**
   */
  public void close() {
    try {
      reader.close();
    } catch (IOException ioe) {
    }
  }

  /**
   * Getter of the property <tt>reader</tt>
   *
   * @return Returns the reader.
   * @uml.property name="reader"
   */
  public BufferedReader getReader() {
    return reader;
  }

  /**
   * Setter of the property <tt>reader</tt>
   *
   * @param reader
   *          The reader to set.
   * @uml.property name="reader"
   */
  public void setReader(BufferedReader reader) {
    this.reader = reader;
  }
}
```

B Testdokumentasjon for online-offline

For å teste låseprotokollen skrives ulike testcase for de situasjonene som ønskes testet. For å utføre disse testcasene skrives det inndatafiler som svarer til den situasjonen som ønskes testet. Disse inndatafilene kjøres og resultatene skrives ut.

B.1 Mal for inndatafiler for online-offline

Inndatafiler skrives etter en mal. Linjer som starter med # er kommentarer og vil ignoreres av metoden som leser inn inndatafilene. Malen for inndatafilene ser slik ut:

```
#Create server (id)
CREATE_SERVER 0

#Number of resources at server
NOF_RESOURCES 2

#Number of mobile tasks at server
NOF_MOBILETASKS 1

#Create MT (mt_id)
CREATE_MOBILETASK 1
NOF_RESOURCES 2
RESOURCE 0
RESOURCE 1

#Number of MSS
NOF_MOBILESUPPORTSTATIONS 1

#Create MSS (mss_id, db_id)
CREATE_MOBILESUPPORTSTATION 0:0

#Number of MH
NOF_MOBILEHOSTS 2

#Create MH (mh_id, mss_id, casheSize, localTime)
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

#START TRANSACTION (mh_id, t_id)
#READ (mh_id,t_id,r_id)
#WRITE (mh_id,t_id,r_id, new_value)
#COMMIT_TRANSACTION (mh_id,t_id)
```

```
#ABORT_TRANSACTION (mh_id,t_id)

#REQUEST_RONLOCK (mh_id,t_id,r_id)
#REQUEST_WONLOCK (mh_id,t_id,r_id)
#REQUEST_WIOFFLOCK (mh_id,t_id,r_id)
#REQUEST_WOFFLOCK (mh_id,t_id,r_id)

#DISCONNECT_PLANNED (mh_id)
#DISCONNECT_UNPLANNED (mh_id)
#RECONNECT (mh_id)

START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_WIOFFLOCK 1:1:0
REQUEST_WOFFLOCK 2:2:0
DISCONNECT_PLANNED 2
WRITE 2:2:0:5
RECONNECT 2
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2

END_FILE
```

B.2 Resultater etter kjøring av inndatafil

For hvert testcase skrives det ut et resultat. Resultatet inneholder `DatabaseServers` `LockLog` og `Log`. I tillegg inneholder resultatet de mobile enhetenes `LocalLockLog` og `LocalLog`. Resultatet vil se slik ut:

```
A new server is created with id 0
A new resource is created with id 0 and value 3
A new resource is created with id 1 and value 9
A new mobile support station is created with id 0
A new mobile host is created with id 1
A new mobile host is created with id 2
Mobilehost 2 disconnects planned at time 65539894
Mobilehost 2 reconnects at time 67524504

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

59963487 1 1 0 WIOFFLOCK [] [] GRANTED
63321456 2 2 0 WOFFLOCK [0(1,WIOFFLOCK)] [] GRANTEDANDTAKEOVER
68900936 2 2 0 RELEASE_WOFFLOCK [0(2,WOFFLOCK)] [] RELEASED
```

```
The databaseservers log:

Time TransactionId action ResourceId  value newValue

57211182 1 START
59405036 2 START
68262866 1 COMMIT
68944517 2 COMMIT

Mobilehost 1s locklog:

Time TransactionId ResourceId grantedLock

59963487 1 0 WIOFFLOCK

Mobilehost 1s local log:

Localtime TransactionId action ResourceId  value newValue


Mobilehost 2s locklog:

Time TransactionId ResourceId grantedLock


Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

67257710 2 WRITEOFFLINE 0 3 5
```

I `currentLocks` i `DatabaseServers` `LockLog` står *X(Y,RONLOCK)* for at transaksjon Y holder en `ROnLock` på ressurs X. Resultatet inneholder mye overflødig informasjon. Informasjon som anses som overflødig vil redigeres vekxk.

B.3 Testcase 1

Testcase 1 ønker å teste situasjonen som er beskrevet i tabell 2 i artikkelen om online-offline [3].

Beskrivelse av testcase 1

Situasjonen er en situasjon hvor tre mobile enheter konkurerer for låser på én ressurs.

Forventet resultat for testcase 1

Resultatet forventes lik det som er presentert i artikkelen, bortsett fra at `LockLog` i denne implementasjonen viser `pendingLocks` før `actionAtDB` er utført, mens den i artikklen vises `pendingLocks` etter at `actionAtDB` er utført.

Inndata for testcase 1

Inndata for testcase 1:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_WIOFFLOCK 1:1:0
DISCONNECT_PLANNED 1
REQUEST_RONLOCK 3:3:0
REQUEST_WIOFFLOCK 2:2:0
DISCONNECT_PLANNED 2
RECONNECT 2
REQUEST_WOFFLOCK 2:2:0
DISCONNECT_PLANNED 2
COMMIT_TRANSACTION 3:3
WRITE 2:2:0:7
RECONNECT 2
RECONNECT 1
REQUEST_WOFFLOCK 1:1:0
REQUEST_WONLOCK 2:2:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2

END_FILE
```

Resultat for testcase 1

Resultat for testcase 1:

```
Mobilehost 1 disconnects planned at time 42802367
Mobilehost 2 disconnects planned at time 44034926
Mobilehost 2 reconnects at time 44295574
Mobilehost 2 disconnects planned at time 45285364
Mobilehost 2 reconnects at time 82892658
Mobilehost 1 reconnects at time 83083185

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

37550024 1 1 0 WIOFFLOCK [] [] GRANTED
42990939 3 3 0 RONLOCK [0(1,WIOFFLOCK)] [] GRANTEDANDTAKEOVER
43918990 2 2 0 WIOFFLOCK [0(3,RONLOCK)] [0(1,WIOFFLOCK)]
  GRANTEDPENDING
44425758 2 2 0 WOFFLOCK [0(3,RONLOCK)] [0(1,WIOFFLOCK),
  0(2,WIOFFLOCK)] GRANTEDUPGRADEPENDING
46033504 3 3 0 RELEASE_RONLOCK [0(3,RONLOCK)]
  [0(1,WIOFFLOCK), 0(2,WOFFLOCK)] RELEASEDDELEGATEWOFFLOCKSANDDELETE
83524023 1 1 0 WOFFLOCK [0(2,WOFFLOCK)] [] REJECTED
84576950 2 2 0 WONLOCK [0(2,WOFFLOCK)] [] GRANTEDSELFUPGRADE
85371185 2 2 0 RELEASE_WONLOCK [0(2,WONLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue

35990608 1 START
37329884 2 START
37396652 3 START
47597111 3 COMMIT
84485598 1 ABORT
85408620 2 COMMIT

Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

82673915 2 WRITEOFFLINE 0 0 7
```

Diskusjon for testcase 1

Resultater er likt det forventede resultatet. Man kan se at transaksjon 2 utfører en skriveoperasjon mens den er frakoblet. Denne er utført for at ikke

wofflåsen skal delegeres til en wiofflås når den mobile enheten tilkobles. I denne implementasjonen er det valgt å abortere transaksjoner som ikke får bevilget en forespurt lås. Av denne grunn aborterer transaksjon 1.

B.4 Testcase 2

Testcase 2 ønker å teste situasjonen som er beskrevet i tabell 3 i artikkelen om online-offline.

Beskrivelse av testcase 2
Situasjonen er lik testcase 1, bortsett fra at:

- transaksjon 2 ikke utfører en skriveoperasjon mens den er frakoblet.
- transaksjon 3 ikke committer før helt til slutt.

Forventet resultat for testcase 2
Resultatet forventes lik det som er presentert i artikkelen.

Inndata for testcase 2
Inndata for testcase 2:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_WIOFFLOCK 1:1:0
DISCONNECT_PLANNED 1
```

```
REQUEST_RONLOCK 3:3:0
REQUEST_WIOFFLOCK 2:2:0
DISCONNECT_PLANNED 2
RECONNECT 2
REQUEST_WOFFLOCK 2:2:0
DISCONNECT_PLANNED 2
RECONNECT 2
RECONNECT 1
REQUEST_WOFFLOCK 1:1:0
COMMIT_TRANSACTION 3:3
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2

END_FILE
```

Resultat for testcase 2

Resultat for testcase 2:

```
Mobilehost 1 disconnects planned at time 44242774
Mobilehost 2 disconnects planned at time 45484831
Mobilehost 2 reconnects at time 45709441
Mobilehost 2 disconnects planned at time 46624641
Mobilehost 2 reconnects at time 46867409
Mobilehost 1 reconnects at time 47016870

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

38931205 1 1 0 WIOFFLOCK [] [] GRANTED
44415142 3 3 0 RONLOCK [0(1,WIOFFLOCK)] [] GRANTEDANDTAKEOVER
45339003 2 2 0 WIOFFLOCK [0(3,RONLOCK)] [0(1,WIOFFLOCK)]
  GRANTEDPENDING
45839904 2 2 0 WOFFLOCK [0(3,RONLOCK)]
  [0(1,WIOFFLOCK), 0(2,WIOFFLOCK)] GRANTEDUPGRADEPENDING
46788349 2 2 0 DELEGATE_WOFFLOCK [0(3,RONLOCK)]
  [0(1,WIOFFLOCK), 0(2,WOFFLOCK)] RECONNECTDELEGATE
55608464 1 1 0 WOFFLOCK [0(3,RONLOCK)]
  [0(1,WIOFFLOCK), 0(2,WIOFFLOCK)] GRANTEDUPGRADEPENDING
71121051 3 3 0 RELEASE_RONLOCK [0(3,RONLOCK)]
  [0(2,WIOFFLOCK), 0(1,WOFFLOCK)] RELEASEDDELEGATEWOFFLOCKSANDDELETE
76575096 1 1 0 RELEASE_WOFFLOCK [0(1,WOFFLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue
```

```
37420957 1 START
38713300 2 START
38779789 3 START
71236708 3 COMMIT
76640467 1 COMMIT
76854181 2 COMMIT
```

Diskusjon for testcase 2
Resultatatet er likt forventet resultat. Implementasjonen er korrekt for denne situasjonen.

B.5 Testcase 3

Testcase 3 ønsker å teste situasjon 1 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 3
I denne situasjonen har en transaksjon allerede en wiofflås i `currentLocks` og ønsker å oppgradere denne til en ronlås. Andre transaksjoner har også wiofflåser i `currentLocks` på denne ressursen.

Forventet resultat for testcase 3
Det forventede resultatet er at wiofflåsen oppgraderes til en ronlås, mens de andre låsene vil flyttes over til `pendingLocks` til ressursen.

Inndata for testcase 3
Inndata for testcase 3:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
```

```
START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_WIOFFLOCK 1:1:0
REQUEST_WIOFFLOCK 2:2:0
REQUEST_RONLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
END_FILE
```

Resultat for testcase 3
Resultat for testcase 3:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

58216897 1 1 0 WIOFFLOCK [] [] GRANTED
115025513 2 2 0 WIOFFLOCK [0(1,WIOFFLOCK)] [] GRANTED
115770860 1 1 0 RONLOCK [0(1,WIOFFLOCK), 0(2,WIOFFLOCK)] []
  GRANTEDUPGRADE
118055228 1 1 0 RELEASE_RONLOCK [0(1,RONLOCK)] [0(2,WIOFFLOCK)]
  RELEASEDDELEGATEWIOFFLOCKS
```

Diskusjon for testcase 3
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.6 Testcase 4

Testcase 4 ønsker å teste situasjon 2 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 4
I denne situasjonen har en transaksjon allerede en wiofflås i `pendingLocks` og ønsker å oppgradere denne til en ronlås. Andre transaksjoner kan også ha wiofflåser i `pendingLocks` på denne ressursen. For at transaksjonen skal ha en wiofflås i `pendingLocks`, må en annen transaksjon allerede ha en ronlås på denne ressursen.

Forventet resultat for testcase 4

Det forventede resultatet er at wiofflåsen i `pendingLocks` blir slettet og en ny oppgradert ronlås havner i `currentLocks`.

Inndata for testcase 4

Inndata for testcase 4:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_WIOFFLOCK 3:3:0
REQUEST_RONLOCK 2:2:0
REQUEST_WIOFFLOCK 1:1:0
REQUEST_RONLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
COMMIT_TRANSACTION 3:3
END_FILE
```

Resultat for testcase 4

Resultat for testcase 4:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

74025330 3 3 0 WIOFFLOCK [] [] GRANTED
86024341 2 2 0 RONLOCK [0(3,WIOFFLOCK)] [] GRANTEDANDTAKEOVER
107701956 1 1 0 WIOFFLOCK [0(2,RONLOCK)] [0(3,WIOFFLOCK)]
  GRANTEDPENDING
107862312 1 1 0 RONLOCK [0(2,RONLOCK)]
```

```
  [0(3,WIOFFLOCK), 0(1,WIOFFLOCK)] GRANTEDUPGRADE
108880598 1 1 0 RELEASE_RONLOCK [0(2,RONLOCK), 0(1,RONLOCK)]
  [0(3,WIOFFLOCK)] RELEASED
128261832 2 2 0 RELEASE_RONLOCK [0(2,RONLOCK)]
  [0(3,WIOFFLOCK)] RELEASEDDELEGATEWIOFFLOCKS
130095584 3 3 0 RELEASE_WIOFFLOCK [0(3,WIOFFLOCK)] [] RELEASED
```

Diskusjon for testcase 4
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.7 Testcase 5

Testcase 5 ønsker å teste situasjon 3 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 5
I denne situasjonen har en transaksjon allerede en wiofflås i `currentLocks` til en ressurs og ønsker å oppgradere denne til en wofflås. Andre transaksjoner kan også ha wiofflåser i `currentLocks` på denne ressursen.

Forventet resultat for testcase 5
Det forventede resultatet er at wiofflåsen som skal oppgraderes blir til en wofflås og andre wiofflåser blir slettet.

Inndata for testcase 5
Inndata for testcase 5:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
```

```
START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_WIOFFLOCK 2:2:0
REQUEST_WIOFFLOCK 1:1:0
REQUEST_WOFFLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
END_FILE
```

Resultat for testcase 5
Resultat for testcase 5:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

32221414 2 2 0 WIOFFLOCK [] [] GRANTED
36744614 1 1 0 WIOFFLOCK [0(2,WIOFFLOCK)] [] GRANTED
36912513 1 1 0 WOFFLOCK [0(2,WIOFFLOCK), 0(1,WIOFFLOCK)] []
  GRANTEDUPGRADEANDDELETE
40379154 1 1 0 RELEASE_WOFFLOCK [0(1,WOFFLOCK)] [] RELEASED
```

Diskusjon for testcase 5
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.8 Testcase 6

Testcase 6 ønsker å teste situasjon 4 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 6
I denne situasjonen har en transaksjon allerede en ronlås i `currentLocks` og ønsker å oppgradere denne til en wofflås. Andre transaksjoner kan også ha ronlåser i `currentLocks` på denne ressursen og wiofflåser i `pendingLocks`.

Forventet resultat for testcase 6
Det forventede resultatet er at når det eksisterer andre ronlåser i `curr-`

entLocks, vil ronlåsen som skal oppgraderes bli til en wofflås som havner i pendingLocks. Hvis ronlåsen er den eneste ronlåsen i currentLocks, vil eventuelle wioffläser i pendingLocks slettes.

Inndata for testcase 6

Inndata for testcase 6:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 4
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0
CREATE_MOBILEHOST 4:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_RONLOCK 2:2:0
REQUEST_RONLOCK 1:1:0
REQUEST_WOFFLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2

START_TRANSACTION 3:3
START_TRANSACTION 4:4
REQUEST_WIOFFLOCK 4:4:1
REQUEST_RONLOCK 3:3:1
REQUEST_WOFFLOCK 3:3:1
COMMIT_TRANSACTION 3:3
COMMIT_TRANSACTION 4:4
END_FILE
```

Resultat for testcase 6

Resultat for testcase 6:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB
```

```
62646230 2 2 0 RONLOCK [] [] GRANTED
71689279 1 1 0 RONLOCK [0(2,RONLOCK)] [] GRANTED
73105381 1 1 0 WOFFLOCK [0(2,RONLOCK), 0(1,RONLOCK)] []
  GRANTEDUPGRADEPENDING
75709902 1 1 0 RELEASE_PENDINGWOFFLOCK [0(2,RONLOCK)] [0(1,WOFFLOCK)]
  RELEASED
75974181 2 2 0 RELEASE_RONLOCK [0(2,RONLOCK)] [] RELEASED
76453292 4 4 1 WIOFFLOCK [] [] GRANTED
78506626 3 3 1 RONLOCK [1(4,WIOFFLOCK)] [] GRANTEDANDTAKEOVER
78749673 3 3 1 WOFFLOCK [1(3,RONLOCK)] [1(4,WIOFFLOCK)]
  GRANTEDUPGRADEANDDELETE
80165217 3 3 1 RELEASE_WOFFLOCK [1(3,WOFFLOCK)] [] RELEASED
```

Diskusjon for testcase 6
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.9 Testcase 7

Testcase 7 ønsker å teste situasjon 5 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 7
I denne situasjonen har en transaksjon allerede en wiofflås i `pendingLocks` og ønsker å oppgradere denne til en wofflås. For at det skal eksistere en wiofflås i `pendingLocks`, må det eksistere minst én ronlås i `currentLocks`. Andre transaksjoner kan også ha wiofflåser i `pendningLocks` på denne ressursen.

Forventet resultat for testcase 7
Det forventede resultatet er at wiofflåsen i `pendingLocks` vil oppgraderes til en wofflås som vil forbli i `pendingLocks`.

Inndata for testcase 7
Inndata for testcase 7:

```
CREATE_SERVER 0

NOF_RESOURCES 1
```

```
NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_RONLOCK 2:2:0
REQUEST_WIOFFLOCK 1:1:0
REQUEST_WOFFLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
END_FILE
```

Resultat for testcase 7
Resultat for testcase 7:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

53383880 2 2 0 RONLOCK [] [] GRANTED
58264109 1 1 0 WIOFFLOCK [0(2,RONLOCK)] [] GRANTEDPENDING
59391068 1 1 0 WOFFLOCK [0(2,RONLOCK)] [0(1,WIOFFLOCK)]
  GRANTEDUPGRADEPENDING
60948249 1 1 0 RELEASE_PENDINGWOFFLOCK [0(2,RONLOCK)] [0(1,WOFFLOCK)]
  RELEASED
61087931 2 2 0 RELEASE_RONLOCK [0(2,RONLOCK)] [] RELEASED
```

Diskusjon for testcase 7
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.10 Testcase 8

Testcase 8 ønsker å teste situasjon 6 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 8

I denne situasjonen har en transaksjon allerede en wiofflås i `currentLocks` og ønsker å oppgradere denne til en wonlås. Andre transaksjoner kan også ha wiofflåser i `currentLocks` på denne ressursen.

Forventet resultat for testcase 8

Det forventede resultatet er at wiofflåsen vil oppgraderes til wonlås og andre wiofflåser vil bli slettet.

Inndata for testcase 8

Inndata for testcase 8:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
REQUEST_WIOFFLOCK 2:2:0
REQUEST_WIOFFLOCK 1:1:0
REQUEST_WONLOCK 1:1:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
END_FILE
```

Resultat for testcase 8

Resultat for testcase 8:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

33003077 2 2 0 WIOFFLOCK [] [] GRANTED
38068246 1 1 0 WIOFFLOCK [0(2,WIOFFLOCK)] [] GRANTED
```

```
38789008 1 1 0 WONLOCK [0(2,WIOFFLOCK), 0(1,WIOFFLOCK)] []
  GRANTEDUPGRADEANDDELETE
40781999 1 1 0 RELEASE_WONLOCK [0(1,WONLOCK)] [] RELEASED
```

Diskusjon for testcase 8
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.11 Testcase 9

Testcase 9 ønsker å teste situasjon 7 i *upgradeLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 9
I denne situasjonen har en transaksjon allerede en wofflås i `currentLocks` og ønsker å oppgradere denne til en wonlås.

Forventet resultat for testcase 9
Det forventede resultatet er at wofflåsen oppgraderes til wonlås.

Inndata for testcase 9
Inndata for testcase 9:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 1
CREATE_MOBILEHOST 1:0:10:0

START_TRANSACTION 1:1
REQUEST_WOFFLOCK 1:1:0
REQUEST_WONLOCK 1:1:0
COMMIT_TRANSACTION 1:1
END_FILE
```

Resultat for testcase 9
Resultat for testcase 9:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

32690747 1 1 0 WOFFLOCK [] [] GRANTED
37230710 1 1 0 WONLOCK [0(1,WOFFLOCK)] [] GRANTEDSELFUPGRADE
39860374 1 1 0 RELEASE_WONLOCK [0(1,WONLOCK)] [] RELEASED
```

Diskusjon for testcase 9
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.12 Testcase 10

Testcase 10 ønsker å teste situasjon 1 i *`delegateLock()`* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 10
I denne situasjonen committer en transaksjon som har den siste ronlåsen i `currentLocks` til en ressurs. Det eksisterer en wiofflås og en wofflås fra to andre transaksjoner i `pendingLocks`.

Forventet resultat for testcase 10
Det forventede resultatet er at wofflåsen vil delegeres til `currentLocks` og at wiofflåsen vil slettes.

Inndata for testcase 10
Inndata for testcase 10:

```
CREATE_SERVER 0

NOF_RESOURCES 1

NOF_MOBILETASKS 0
```

```
NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_RONLOCK 1:1:0
REQUEST_WIOFFLOCK 2:2:0
REQUEST_WOFFLOCK 3:3:0
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
COMMIT_TRANSACTION 3:3
END_FILE
```

Resultat for testcase 10
Resultat for testcase 10:

```
The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

32001832 1 1 0 RONLOCK [] [] GRANTED
38058748 2 2 0 WIOFFLOCK [0(1,RONLOCK)] [] GRANTEDPENDING
39582125 3 3 0 WOFFLOCK [0(1,RONLOCK)] [0(2,WIOFFLOCK)] GRANTEDPENDING
41093211 1 1 0 RELEASE_RONLOCK [0(1,RONLOCK)]
  [0(2,WIOFFLOCK),0(3,WOFFLOCK)] RELEASEDDELEGATEWOFFLOCKSANDDELETE
41922926 3 3 0 RELEASE_WOFFLOCK [0(3,WOFFLOCK)] [] RELEASED
```

Diskusjon for testcase 10
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.13 Testcase 11

Testcase 11 ønsker å teste situasjon 2 i *delegateLock()* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 11
I denne situasjonen vil en ronlås delegeres til en wiofflås når en transaksjon frakobles. Dette skal gjøres både for planlagte og uplanlagte frakoblinger. Testcaset tar for seg begge.

Forventet resultat for testcase 11
Det forventede resultatet er at ronlåser delegeres til wiofflåser.

Inndata for testcase 11
Inndata for testcase 11:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_RONLOCK 1:1:0
REQUEST_RONLOCK 2:2:1
REQUEST_RONLOCK 3:3:1
DISCONNECT_PLANNED 1
DISCONNECT_UNPLANNED 2
RECONNECT 1
RECONNECT 2
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
COMMIT_TRANSACTION 3:3
END_FILE
```

Resultat for testcase 11
Resultat for testcase 11:

```
Mobilehost 1 disconnects planned at time 75918588
```

```
DB found disconnected mobilehost 2 at time 270727704
Mobilehost 1 reconnects at time 576582778
Mobilehost 2 reconnects at time 578635274

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

62532529 1 1 0 RONLOCK [] [] GRANTED
74051311 2 2 1 RONLOCK [] [] GRANTED
74237927 3 3 1 RONLOCK [1(2,RONLOCK)] [] GRANTED
74456670 1 1 0 DELEGATE_RONLOCK [0(1,RONLOCK)] [] DISCONNECTEDDELEGATE
270633000 2 2 1 DELEGATE_RONLOCK [1(2,RONLOCK), 1(3,RONLOCK)] []
  DISCONNECTEDDELEGATE
580090486 1 1 0 RELEASE_WIOFFLOCK [0(1,WIOFFLOCK)] [] RELEASED
580217877 2 2 1 RELEASE_PENDINGWIOFFLOCK [1(3,RONLOCK)]
  [1(2,WIOFFLOCK)] RELEASED
580340239 3 3 1 RELEASE_RONLOCK [1(3,RONLOCK)] [] RELEASED
```

Diskusjon for testcase 11
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

B.14 Testcase 12

Testcase 12 ønsker å teste situasjon 3 i *`delegateLock()`* som er beskrevet i avsnitt 7.1.2.

Beskrivelse av testcase 12
I denne situasjonen vil en wofflås delegeres til en wiofflås når en transaksjon tilkobles uten å ha modifisert verdien på ressursen. Testcaset tester både når wofflåsen er i `currentLocks` og i `penidngLocks`.

Forventet resultat for testcase 12
Det forventede resultatet er at wofflåsen delegeres til en wiofflås.

Inndata for testcase 12
Inndata for testcase 12:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILETASKS 0

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 3
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0
CREATE_MOBILEHOST 3:0:10:0

START_TRANSACTION 1:1
START_TRANSACTION 2:2
START_TRANSACTION 3:3
REQUEST_RONLOCK 3:3:0
REQUEST_WOFFLOCK 1:1:0
REQUEST_WOFFLOCK 2:2:1
DISCONNECT_PLANNED 1
DISCONNECT_UNPLANNED 2
RECONNECT 1
RECONNECT 2
COMMIT_TRANSACTION 1:1
COMMIT_TRANSACTION 2:2
COMMIT_TRANSACTION 3:3
END_FILE
```

Resultat for testcase 12

Resultat for testcase 12:

```
Mobilehost 1 disconnects planned at time 71105685
DB found disconnected mobilehost 2 at time 274706981
Mobilehost 1 reconnects at time 572035552
Mobilehost 2 reconnects at time 572278599

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

58556604 3 3 0 RONLOCK [] [] GRANTED
69382002 1 1 0 WOFFLOCK [0(3,RONLOCK)] [] GRANTEDPENDING
70880517 2 2 1 WOFFLOCK [] [] GRANTED
571797253 1 1 0 DELEGATE_WOFFLOCK [0(3,RONLOCK)] [0(1,WOFFLOCK)]
  RECONNECTDELEGATE
572226637 2 2 1 DELEGATE_WOFFLOCK [1(2,WOFFLOCK)] [] RECONNECTDELEGATE
```

```
573925177 1 1 0 RELEASE_PENDINGWIOFFLOCK [0(3,RONLOCK)]
  [0(1,WIOFFLOCK)] RELEASED
574676669 2 2 1 RELEASE_WIOFFLOCK [1(2,WIOFFLOCK)] [] RELEASED
574799869 3 3 0 RELEASE_RONLOCK [0(3,RONLOCK)] [] RELEASED
```

Diskusjon for testcase 12
Resultatet svarer til forventet resultat. Implementasjonen av låseprotokollen er korrekt for denne situasjonen.

C Kildekode for den adaptive låseprotokollen

I appendiks B finner man kildekoden til den adaptive låseprotokollen. Bare klasser som er utvidet eller modifisert er tatt med. I tillegg har det kommet noen nye klasser.

C.1 Modulen *dataBase*

De utvidede klassene i modulen *dataBase* er:

- `DatabaseServer`
- `LockManger`

Den nye klassen i modulen *dataBase* er:

- `TransactionManager`

I tillegg er klassen `MobileTask` fjernet fra modulen.

C.1.1 `DatabaseServer`

```
package dataBase;

import input.InputFileReader;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import operation.AbortOperation;
import operation.CommitOperation;
import operation.Operation;
import operation.ReadOperation;
import operation.WriteOperation;
import transaction.ExportTransaction;
import transaction.ImportTransaction;
import transaction.Transaction;

import lock.BrowseLock;
import lock.Lock;
```

```
import mobileHost.LocalLogRecord;
import mobileHost.MobileHost;

public class DatabaseServer {

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="databaseSize"
   */
  private int databaseSize;

  /**
   * @uml.property name="time"
   */
  private long time;

  /**
   * @uml.property name="lockLogId"
   */
  private int lockLogId;

  /**
   * @uml.property name="mobileSupportStation"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="databaseServer:dataBase.MobileSupportStation"
   */

  private Collection<MobileSupportStation> mobileSupportStation;

  /**
   * @uml.property name="lockLog"
   * @uml.associationEnd inverse="databaseServer:dataBase.LockLog"
   */
  private LockLog lockLog;

  /**
   * @uml.property name="convertionLog"
   * @uml.associationEnd
    inverse="databaseServer:dataBase.ConvertionLog"
   */
  private ConvertionLog convertionLog;

  /**
   * @uml.property name="lockManager"
```

```
 * @uml.associationEnd inverse="databaseServer:dataBase.LockManager"
 */
private LockManager lockManager;

/**
 * @uml.property name="connectivityManager"
 * @uml.associationEnd
  inverse="databaseServer:dataBase.ConnectivityManager"
 */
private ConnectivityManager connectivityManager;

/**
 * @uml.property name="resource"
 * @uml.associationEnd multiplicity="(0 -1)"
 *                     inverse="databaseServer:dataBase.Resource"
 */
private Collection<Resource> resource;

/**
 * @uml.property name="log"
 * @uml.associationEnd inverse="databaseServer:dataBase.Log"
 */
private Log log;

/**
 * @uml.property name="inputFileReader"
 */
private InputFileReader inputFileReader;

/**
 * @uml.property name="nanoStartTime"
 */
private long nanoStartTime;

/**
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
 */
private Collection<Lock> lostLocksWhenMobileHostIsDisconnected;

/**
 * @uml.property name="gainedLocksWhenMobileHostIsDisconnected"
 */
private Collection<Lock> gainedLocksWhenMobileHostIsDisconnected;

/**
 * @uml.property name="activeTransactions"
 */
private Collection<Transaction> activeTransactions;
```

```
/**
 * @uml.property name="transactionManager"
 * @uml.associationEnd
  inverse="databaseServer:dataBase.TransactionManager"
 */
private TransactionManager transactionManager;

/**
 * @uml.property name="dependentCommitOperations"
 */
private Collection<CommitOperation> dependentCommitOperations;

public DatabaseServer(String id, int databaseSize, long time) {
  this.id = id;
  this.databaseSize = databaseSize;
  this.time = time;
  this.nanoStartTime = System.nanoTime();
  lockLogId = 0;
  resource = new ArrayList<Resource>();
  mobileSupportStation = new ArrayList<MobileSupportStation>();
  lockManager = new LockManager(this);
  lockLog = new LockLog(this);
  convertionLog = new ConvertionLog(this);
  connectivityManager = new ConnectivityManager(this);
  log = new Log(this);
  activeTransactions = new ArrayList<Transaction>();
  lostLocksWhenMobileHostIsDisconnected = new ArrayList<Lock>();
  gainedLocksWhenMobileHostIsDisconnected = new ArrayList<Lock>();
  transactionManager = new TransactionManager(this);
  dependentCommitOperations = new ArrayList<CommitOperation>();
}

/**
 */
public int read(ReadOperation readOperation) {
  Transaction transaction = readOperation.getTransaction();
  Resource resource = readOperation.getResource();
  int readValue;
  if (transaction.isResourceWritten(resource)) {
    readValue = transaction.getNewestWrittenValue(resource,
        readOperation);
  } else {
    readValue = getResource(resource.getId()).getValue();
  }
  readOperation.setReadValue(readValue);
  log.addLogRecord(transaction.getId(), "READ", resource.getId(),
      new String("" + readValue), "");
  return readValue;
}
```

```
/**
 */
public void write(WriteOperation writeOperation) {
  Transaction transaction = writeOperation.getTransaction();
  Resource resource = writeOperation.getResource();
  int newValue = writeOperation.getNewValue();
  transaction.getWrittenResources().add(resource);
  log.addLogRecord(transaction.getId(), "WRITE", resource.getId(),
      new String("" + resource.getValue()), new String(""
          + newValue));
}

/**
 */
public void commit(CommitOperation commitOperation) {
  Transaction transaction = commitOperation.getTransaction();

  boolean dependencyTransactionNotCommitted = false;
  for (Iterator<ImportTransaction> i = transaction
      .getImportTransaction().iterator(); i.hasNext();) {
    ImportTransaction imT = i.next();
    if (!imT.getShareToRead()) {
      Transaction dependentTransaction = imT.getSharedData()
          .getExportTransaction().getTransaction();
      for (Iterator<Transaction> j = activeTransactions.iterator(); j
          .hasNext();) {
        Transaction dT = j.next();
        if (dependentTransaction.equals(dT)) {
          dependencyTransactionNotCommitted = true;
        }
      }
    }
  }

  if (dependencyTransactionNotCommitted) {
    dependentCommitOperations.add(commitOperation);
  } else {
    activeTransactions.remove(transaction);
    Collection<Operation> operations = transaction.getOperation();
    for (Iterator<Operation> i = operations.iterator(); i
        .hasNext();) {
      Operation op = i.next();
      if (op.getOperationType().equals("WRITEOPERATION")) {
        WriteOperation wo = (WriteOperation) op;
        if (wo.getCommitResponsibility()) {
          wo.getResource().setValue(wo.getNewValue());
        }
        for (Iterator<ReadOperation> j = wo
```

```
        .getReadOperationsForReadBrowsedValueAgain()
        .iterator(); j.hasNext();) {
      ReadOperation ro = j.next();
      ro.setReadValue(wo.getNewValue());
    }
  }
}

for (Iterator<Lock> i = transaction.getLocks().iterator(); i
    .hasNext();) {
  Lock lock = i.next();
  transaction.getOwner().getLocalCacheManager()
      .removeResource(lock.getResource(), transaction);
}
transaction.setStatus("COMMITED");

for (Iterator<ExportTransaction> i = transaction
    .getExportTransaction().iterator(); i.hasNext();) {
  ExportTransaction exT = i.next();
  if (!exT.getForcedCommitOnAbortDependency()) {
    exT.getSharedData().getLock().setTransaction(transaction);
  }
}

getLockManager().releaseLocks(transaction);

for (Iterator<ExportTransaction> i = transaction
    .getForcedCommitOnAbortDependentExportTransaction()
    .iterator(); i.hasNext();) {
  ExportTransaction exT = i.next();
  exT.getMobileAffiliation().getExportImportRepository()
      .getSharedData().remove(exT.getSharedData());
  log.addLogRecord(exT.getId(), "EXPORTTRANSACTIONCOMMIT", "",
      "", "");
}

log.addLogRecord(transaction.getId(), "COMMIT", "", "", "");

for (Iterator<ExportTransaction> i = transaction
    .getExportTransaction().iterator(); i.hasNext();) {
  ExportTransaction exT = i.next();
  if (!exT.getForcedCommitOnAbortDependency()) {
    exT.getMobileAffiliation().getExportImportRepository()
        .getSharedData().remove(exT.getSharedData());
    exT.getSharedData().getResource().setValue(
        exT.getSharedData().getNewValue());
    System.out.println("Exporttransaction " + exT.getId()
        + " commits resource "
        + exT.getSharedData().getResource().getId());
```

```
        log.addLogRecord(exT.getId(), "EXPORTTRANSACTIONCOMMIT",
            "", "", "");
      }
    }

    for (Iterator<ImportTransaction> i = transaction
        .getImportTransaction().iterator(); i.hasNext();) {
      ImportTransaction imT = i.next();
      log.addLogRecord(imT.getId(), "IMPORTTRANSACTIONCOMMIT", "",
          "", "");
    }

    Collection<CommitOperation>
      commitOperationDependentOnThisTransaction = new
        ArrayList<CommitOperation>();
    for (Iterator<CommitOperation> i = dependentCommitOperations
        .iterator(); i.hasNext();) {
      CommitOperation co = i.next();
      boolean dependencyTransactionNotCommitted2 = false;
      for (Iterator<ImportTransaction> j = co.getTransaction()
          .getImportTransaction().iterator(); j.hasNext();) {
        ImportTransaction imT = j.next();
        if (!imT.getShareToRead()) {
          Transaction dependentTransaction = imT.getSharedData()
              .getExportTransaction().getTransaction();
          for (Iterator<Transaction> k = activeTransactions
              .iterator(); k.hasNext();) {
            Transaction dT = k.next();
            if (dependentTransaction.equals(dT)) {
              dependencyTransactionNotCommitted2 = true;
            }
          }
        }
      }
      if (!dependencyTransactionNotCommitted2) {
        commitOperationDependentOnThisTransaction.add(co);
      }
    }
    for (Iterator<CommitOperation> i =
      commitOperationDependentOnThisTransaction
        .iterator(); i.hasNext();) {
      CommitOperation co = i.next();
      dependentCommitOperations.remove(co);
      commit(co);
    }
  }
}

/**
```

```
 */
public void abort(AbortOperation abortOperation) {
  Transaction transaction = abortOperation.getTransaction();
  activeTransactions.remove(transaction);
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    transaction.getOwner().getLocalCacheManager().removeResource(
        lock.getResource(), transaction);
  }
  transaction.setStatus("ABORTED");

  getLockManager().releaseLocks(transaction);
  log.addLogRecord(transaction.getId(), "ABORT", "", "", "");

  for (Iterator<ExportTransaction> i = transaction
      .getForcedCommitOnAbortDependentExportTransaction()
      .iterator(); i.hasNext();) {
    ExportTransaction exT = i.next();
    if (exT.getMobileAffiliation() != null) {
      exT.getMobileAffiliation().getExportImportRepository()
          .getSharedData().remove(exT.getSharedData());
    }
    exT.getSharedData().getResource().setValue(
        exT.getSharedData().getNewValue());
    System.out.println("Exporttransaction " + exT.getId()
        + " commits resource "
        + exT.getSharedData().getResource().getId());
    log.addLogRecord(exT.getId(), "EXPORTTRANSACTIONCOMMIT", "",
        "", "");
  }

  Collection<CommitOperation>
    commitOperationDependentOnThisTransaction = new
      ArrayList<CommitOperation>();
  for (Iterator<CommitOperation> i = dependentCommitOperations
      .iterator(); i.hasNext();) {
    CommitOperation co = i.next();
    boolean dependencyTransactionNotCommitted2 = false;
    for (Iterator<ImportTransaction> j = co.getTransaction()
        .getImportTransaction().iterator(); j.hasNext();) {
      ImportTransaction imT = j.next();
      if (!imT.getShareToRead()) {
        Transaction dependentTransaction = imT.getSharedData()
            .getExportTransaction().getTransaction();
        for (Iterator<Transaction> k = activeTransactions
            .iterator(); k.hasNext();) {
          Transaction dT = k.next();
          if (dependentTransaction.equals(dT)) {
```

```
            dependencyTransactionNotCommitted2 = true;
          }
        }
      }
    }
    if (!dependencyTransactionNotCommitted2) {
      commitOperationDependentOnThisTransaction.add(co);
    }
  }
  for (Iterator<CommitOperation> i =
    commitOperationDependentOnThisTransaction
      .iterator(); i.hasNext();) {
    CommitOperation co = i.next();
    dependentCommitOperations.remove(co);
    commit(co);
  }
}

/**
 */
public boolean requestROnLock(Resource resource,
    Transaction transaction) {
  return lockManager.handleROnLockRequest(resource, transaction);
}

/**
 */
public boolean requestWOnLock(Resource resource,
    Transaction transaction) {
  return lockManager.handleWOnLockRequest(resource, transaction);
}

/**
 */
public boolean requestWIOffLock(Resource resource,
    Transaction transaction) {
  return lockManager.handleWIOffLockRequest(resource, transaction);
}

/**
 */
public boolean requestWOffLock(Resource resource,
    Transaction transaction) {
  return lockManager.handleWOffLockRequest(resource, transaction);
}

/**
 */
public boolean requestBrowseLock(Resource resource,
```

```
    Transaction transaction) {
  return lockManager
      .handleBrowseLockRequest(resource, transaction);
}

/**
 */
public void reconnect(MobileHost mobileHost) {
  getConnectivityManager().getOfflineMobileHosts().remove(
      mobileHost);
  getConnectivityManager().getOnlineMobileHosts().add(mobileHost);

  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1 =
    reconnectCheck1(mobileHost);
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2 =
    reconnectCheck2(mobileHost,
      localLogRecordsWhichNotPassesCheck1);
  reconnectCheck3(mobileHost, localLogRecordsWhichNotPassesCheck2);

  Collection<Lock> lostLocks = new ArrayList<Lock>();
  for (Iterator<Lock> i = lostLocksWhenMobileHostIsDisconnected
      .iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock.getTransaction().getOwner().getId().equals(
        mobileHost.getId())) {
      lostLocks.add(lock);
    }
  }
  for (Iterator<Lock> i = lostLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    lock.getTransaction().getLocks().remove(lock);
    mobileHost.getLocalCacheManager().removeResource(
        lock.getResource(), lock.getTransaction());
    lostLocksWhenMobileHostIsDisconnected.remove(lock);
  }
  Collection<Lock> gainedLocks = new ArrayList<Lock>();
  for (Iterator<Lock> i = gainedLocksWhenMobileHostIsDisconnected
      .iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock.getTransaction().getOwner().getId().equals(
        mobileHost.getId())) {
      gainedLocks.add(lock);
    }
  }
  for (Iterator<Lock> i = gainedLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    lock.getTransaction().getLocks().add(lock);
    mobileHost.getLocalCacheManager().addResource(
        lock.getResource(), lock.getTransaction());
```

```
  if (lock.getLockType().equals("WIOFFLOCK")) {
    long timeWhenLockWasGranted = getLockLog()
        .getTimeWhenLockWasGranted(lock);
    mobileHost.getLocalLockLog().addLocalLockLogRecord(
        timeWhenLockWasGranted, lock.getTransaction().getId(),
        lock.getResource().getId(), "WIOFFLOCK");
  }
  gainedLocksWhenMobileHostIsDisconnected.remove(lock);
}

reconnectCheckForUnplannedDisconnection
  WhoHasLostLockWhenDisconnected(mobileHost);
for (Iterator<BrowseLock> i = mobileHost.getAnchorTransaction()
    .getInconsistentBrowseLocks().iterator(); i.hasNext();) {
  BrowseLock browseLock = i.next();
  WriteOperation writeOperationWithNewValue = null;
  Transaction wOnTransaction = browseLock.getGrantedWOnLock()
      .getTransaction();
  if (wOnTransaction.getStatus().equals("COMMITED")) {
    for (Iterator<Operation> j = wOnTransaction.getOperation()
        .iterator(); j.hasNext();) {
      Operation operation = j.next();
      if (operation instanceof WriteOperation) {
        WriteOperation writeOperation = (WriteOperation) operation;
        if (writeOperation.getResource().equals(
            browseLock.getGrantedWOnLock().getResource())) {
          writeOperationWithNewValue = writeOperation;
        }
      }
    }
    for (Iterator<Operation> j = browseLock.getTransaction()
        .getOperation().iterator(); j.hasNext();) {
      Operation operation = j.next();
      if (operation instanceof ReadOperation) {
        ReadOperation readOperation = (ReadOperation) operation;
        if (readOperation.getTransaction().equals(
            browseLock.getTransaction())
            && readOperation.getResource().equals(
                browseLock.getResource())) {
          readOperation.setReadValue(writeOperationWithNewValue
              .getNewValue());
        }
      }
    }
  } else {
    mobileHost.getLocalTransactionManager()
        .registerForReadingBrowsedValueAgain(browseLock);
  }
}
```

```
  mobileHost.getAnchorTransaction().getInconsistentBrowseLocks()
      .clear();

  Collection<LocalLogRecord> localLogRecordsForStartTransactionCheck
      = mobileHost
        .getLocalLog().getLocalLogRecordsFromLastDisconnection(
          mobileHost.getLastDisconnectionTime());
  for (Iterator<LocalLogRecord> i =
      localLogRecordsForStartTransactionCheck
        .iterator(); i.hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    if (localLogRecord.getAction().equals("STARTOFFLINE")) {
      initiateTransaction(mobileHost.getLocalTransactionManager()
          .getTransaction(localLogRecord.getTransactionId()));
    }
    if (localLogRecord.getAction().equals("COMMITOFFLINE")) {
      commit((CommitOperation) mobileHost
          .getLocalTransactionManager().getTransaction(
              localLogRecord.getTransactionId()).getOperation(
              "COMMITOPERATION"));
    }
    if (localLogRecord.getAction().equals("ABORTOFFLINE")) {
      abort((AbortOperation) mobileHost
          .getLocalTransactionManager().getTransaction(
              localLogRecord.getTransactionId()).getOperation(
              "ABORTOPERATION"));
    }
  }
  for (Iterator<Transaction> i = mobileHost
      .getLocalTransactionManager().getTransaction().iterator(); i
      .hasNext();) {
    Transaction transaction = i.next();
    getLockManager().handleReconnectedTransaction(transaction);
  }
  System.out.println("Mobilehost " + mobileHost.getId()
      + " reconnects at time " + getTime());

}

private Collection<LocalLogRecord> reconnectCheck1(MobileHost mh) {
  Collection<LocalLogRecord> operationsWithPossibleConflict = new
    ArrayList<LocalLogRecord>();
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1 =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LocalLogRecord> localLogRecords = mh.getLocalLog()
      .getLocalLogRecordsFromLastDisconnection(
          lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
```

```
    .hasNext();) {
  LocalLogRecord localLogRecord = i.next();
  if (localLogRecord.getAction().equals("READOFFLINE")) {
    Transaction t = mh.getLocalTransactionManager()
        .getTransaction(localLogRecord.getTransactionId());
    Collection<Lock> locks = t.getLocks();

    for (Iterator<Lock> j = locks.iterator(); j.hasNext();) {
      Lock lock = j.next();
      if (lock.getResource().getId().equals(
          localLogRecord.getResourceId())) {
        if (lock.getLockType().equals("WIOFFLOCK")
            || lock.getLockType().equals("RONLOCK")) {
          operationsWithPossibleConflict.add(localLogRecord);
          System.out
              .println("Found operation with possible conflict "
                  + localLogRecord.getTransactionId() + " "
                  + localLogRecord.getResourceId());
        }
      }
    }
  }
}
Collection<LockLogRecord> lockLogRecords = getLockLog()
    .getLockLogRecordsFromTime(lastDisconnectionTime);
for (Iterator<LocalLogRecord> i = operationsWithPossibleConflict
    .iterator(); i.hasNext();) {
  LocalLogRecord localLogRecord = i.next();
  for (Iterator<LockLogRecord> j = lockLogRecords.iterator(); j
      .hasNext();) {
    LockLogRecord lockLogRecord = j.next();
    if (localLogRecord.getResourceId().equals(
        lockLogRecord.getResourceId())
        && (lockLogRecord.getRequstingLockType().equals(
            "WOFFLOCK") || lockLogRecord.getRequstingLockType()
            .equals("WONLOCK"))
        && (new Long(localLogRecord.getLocalTime()) > new Long(
            lockLogRecord.getTime()))) {
      if (!localLogRecordsWhichNotPassesCheck1
          .contains(localLogRecord)) {
        localLogRecordsWhichNotPassesCheck1.add(localLogRecord);
        System.out
            .println("Found operation which not passes check 1: "
                + localLogRecord.getTransactionId()
                + " "
                + localLogRecord.getResourceId());
      }
    }
  }
```

```
  }
  return localLogRecordsWhichNotPassesCheck1;
}

private Collection<LocalLogRecord> reconnectCheck2(MobileHost mh,
    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck1) {
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2 =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LogRecord> logRecords = getLog()
      .getLogRecordsFromTime(lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = localLogRecordsWhichNotPassesCheck1
      .iterator(); i.hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    for (Iterator<LogRecord> j = logRecords.iterator(); j
        .hasNext();) {
      LogRecord logRecord = j.next();
      if (localLogRecord.getResourceId().equals(
          logRecord.getResourceId())
          && logRecord.getAction().equals("WRITE")
          && (new Long(localLogRecord.getLocalTime()) > new Long(
              logRecord.getTime()))) {
        localLogRecordsWhichNotPassesCheck2.add(localLogRecord);
        System.out
            .println("Found operation which not passes check 2: "
                + localLogRecord.getTransactionId() + " "
                + localLogRecord.getResourceId());
      }
    }
  }
  localLogRecordsWhichNotPassesCheck1
      .removeAll(localLogRecordsWhichNotPassesCheck2);
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck1);
  return localLogRecordsWhichNotPassesCheck2;
}

private void reconnectCheck3(MobileHost mh,
    Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck2) {
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck2);
}

private void
  reconnectCheckForUnplannedDisconnectionWhoHasLostLockWhenDisconnected(
    MobileHost mh) {
  Collection<LocalLogRecord> localLogRecordsWhichNotPassesCheck =
    new ArrayList<LocalLogRecord>();
  long lastDisconnectionTime = mh.getLastDisconnectionTime();
  Collection<LocalLogRecord> localLogRecords = mh.getLocalLog()
      .getLocalLogRecordsFromLastDisconnection(
```

```
          lastDisconnectionTime);
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    if (localLogRecord.getAction().equals("READOFFLINE")) {
      Transaction t = mh.getLocalTransactionManager()
          .getTransaction(localLogRecord.getTransactionId());
      if (t.getStatus().equals("ACTIVE")) {
        Collection<Lock> locks = t.getLocks();
        boolean lostLock = true;
        for (Iterator<Lock> j = locks.iterator(); j.hasNext();) {
          Lock lock = j.next();
          if (lock.getResource().getId().equals(
              localLogRecord.getResourceId())) {
            if (lock.getLockType().equals("WIOFFLOCK")
                || lock.getLockType().equals("BROWSELOCK")
                || lock.getLockType().equals("WOFFLOCK")) {
              lostLock = false;
            }
          }
        }
        if (lostLock) {
          localLogRecordsWhichNotPassesCheck.add(localLogRecord);
          System.out.println("Found operation with lost lock "
              + localLogRecord.getTransactionId() + " "
              + localLogRecord.getResourceId());
        }
      }

    }
  }
  rectifyOperations(mh, localLogRecordsWhichNotPassesCheck);
}

/**
 */
public void rectifyOperations(MobileHost mh,
    Collection<LocalLogRecord> localLogRecords) {
  Collection<Transaction> t = new ArrayList<Transaction>();
  for (Iterator<LocalLogRecord> i = localLogRecords.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    t.add(mh.getLocalTransactionManager().getTransaction(
        localLogRecord.getTransactionId()));
  }
  for (Iterator<Transaction> i = t.iterator(); i.hasNext();) {
    Transaction transaction = i.next();
    mh.getLocalTransactionManager().executeAbortOperation(
        transaction.getId());
```

```
  }
}

/**
 */
public void initiatePlannedDisconnection(MobileHost mobileHost) {
  for (Iterator<Transaction> i = mobileHost
      .getLocalTransactionManager().getTransaction().iterator(); i
      .hasNext();) {
    Transaction transaction = i.next();
    getLockManager().handleDisconnectedTransaction(transaction,
        true);
  }
  getConnectivityManager().getOnlineMobileHosts().remove(
      mobileHost);
  getConnectivityManager().getOfflineMobileHosts().add(mobileHost);
  System.out.println("Mobilehost " + mobileHost.getId()
      + " disconnects planned at time " + getTime());

}

public void handleUnplannedDisconnection(MobileHost mobileHost) {
  for (Iterator<Transaction> i = getActiveTransactions()
      .iterator(); i.hasNext();) {
    Transaction transaction = i.next();
    if (transaction.getOwner().getId().equals(mobileHost.getId())) {
      getLockManager().handleDisconnectedTransaction(transaction,
          false);
    }
  }
  System.out.println("DB found disconnected mobilehost "
      + mobileHost.getId() + " at time " + getTime());
}

/**
 */
public void initiateTransaction(Transaction transaction) {
  activeTransactions.add(transaction);
  log.addLogRecord(transaction.getId(), "START", "", "", "");
}

/**
 */
public void notifyAboutLostWIOffLock(Lock lostLock) {
  if (lostLock.getTransaction().getOwner().getConnectionStatus()) {
    lostLock.getTransaction().getOwner().notifyAboutLostWIOffLock(
        lostLock);
  } else {
    lostLocksWhenMobileHostIsDisconnected.add(lostLock);
```

```
  }
}

/**
 */
public Resource getResource(String id) {
  for (Iterator<Resource> i = resource.iterator(); i.hasNext();) {
    Resource r = i.next();
    if (r.getId().equals(id)) {
      return r;
    }
  }
  return null;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>time</tt>
 *
 * @return Returns the time.
 * @uml.property name="time"
 */
public long getTime() {
  time = System.nanoTime() - nanoStartTime;
  return time;
}

/**
 * Setter of the property <tt>time</tt>
```

```
 *
 * @param time
 *          The time to set.
 * @uml.property name="time"
 */
public void setTime(long time) {
  this.time = time;
}

/**
 * Getter of the property <tt>databaseSize</tt>
 *
 * @return Returns the databaseSize.
 * @uml.property name="databaseSize"
 */
public int getDatabaseSize() {
  return databaseSize;
}

/**
 * Setter of the property <tt>databaseSize</tt>
 *
 * @param databaseSize
 *          The databaseSize to set.
 * @uml.property name="databaseSize"
 */
public void setDatabaseSize(int databaseSize) {
  this.databaseSize = databaseSize;
}

/**
 * Getter of the property <tt>lockLogId</tt>
 *
 * @return Returns the lockLogId.
 * @uml.property name="lockLogId"
 */
public int getLockLogId() {
  return lockLogId;
}

/**
 * Setter of the property <tt>lockLogId</tt>
 *
 * @param lockLogId
 *          The lockLogId to set.
 * @uml.property name="lockLoggId"
 */
public void setLockLogId(int lockLogId) {
  this.lockLogId = lockLogId;
```

```
}

/**
 * Getter of the property <tt>mobileSupportStation</tt>
 *
 * @return Returns the mobileSupportStation.
 * @uml.property name="mobileSupportStation"
 */
public Collection<MobileSupportStation> getMobileSupportStation() {
  return mobileSupportStation;
}

/**
 * Setter of the property <tt>mobileSupportStation</tt>
 *
 * @param mobileSupportStation
 *          The mobileSupportStation to set.
 * @uml.property name="mobileSupportStation"
 */
public void setMobileSupportStation(
    Collection<MobileSupportStation> mobileSupportStation) {
  this.mobileSupportStation = mobileSupportStation;
}

/**
 * Getter of the property <tt>lockLog</tt>
 *
 * @return Returns the lockLog.
 * @uml.property name="lockLog"
 */
public LockLog getLockLog() {
  return lockLog;
}

/**
 * Setter of the property <tt>lockLog</tt>
 *
 * @param lockLog
 *          The lockLog to set.
 * @uml.property name="lockLog"
 */
public void setLockLog(LockLog lockLog) {
  this.lockLog = lockLog;
}

/**
 * Getter of the property <tt>convertionLog</tt>
 *
 * @return Returns the convertionLog.
```

```
 * @uml.property name="convertionLog"
 */
public ConvertionLog getConvertionLog() {
  return convertionLog;
}

/**
 * Setter of the property <tt>convertionLog</tt>
 *
 * @param convertionLog
 *          The convertionLog to set.
 * @uml.property name="convertionLog"
 */
public void setConvertionLog(ConvertionLog convertionLog) {
  this.convertionLog = convertionLog;
}

/**
 * Getter of the property <tt>lockManager</tt>
 *
 * @return Returns the lockManager.
 * @uml.property name="lockManager"
 */
public LockManager getLockManager() {
  return lockManager;
}

/**
 * Setter of the property <tt>lockManager</tt>
 *
 * @param lockManager
 *          The lockManager to set.
 * @uml.property name="lockManager"
 */
public void setLockManager(LockManager lockManager) {
  this.lockManager = lockManager;
}

/**
 * Getter of the property <tt>connectivityManager</tt>
 *
 * @return Returns the connectivityManager.
 * @uml.property name="connectivityManager"
 */
public ConnectivityManager getConnectivityManager() {
  return connectivityManager;
}

/**
```

```
 * Setter of the property <tt>connectivityManager</tt>
 *
 * @param connectivityManager
 *          The connectivityManager to set.
 * @uml.property name="connectivityManager"
 */
public void setConnectivityManager(
    ConnectivityManager connectivityManager) {
  this.connectivityManager = connectivityManager;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Collection<Resource> getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Collection<Resource> resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>input</tt>
 *
 * @return Returns the input.
 * @uml.property name="inputFileReader"
 */
public InputFileReader getInputFileReader() {
  return inputFileReader;
}

/**
 * Setter of the property <tt>input</tt>
 *
 * @param input
 *          The input to set.
 * @uml.property name="inputFileReader"
 */
```

```
public void setInputFileReader(InputFileReader inputFileReader) {
  this.inputFileReader = inputFileReader;
}

/**
 * Getter of the property <tt>log</tt>
 *
 * @return Returns the log.
 * @uml.property name="log"
 */
public Log getLog() {
  return log;
}

/**
 * Setter of the property <tt>log</tt>
 *
 * @param log
 *          The log to set.
 * @uml.property name="log"
 */
public void setLog(Log log) {
  this.log = log;
}

public long getNanoStartTime() {
  return nanoStartTime;
}

/**
 * Getter of the property
  <tt>lostLocksWhenMobileHostIsDisconnected</tt>
 *
 * @return Returns the lostLocksWhenMobileHostIsDisconnected.
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
 */
public Collection<Lock> getLostLocksWhenMobileHostIsDisconnected() {
  return lostLocksWhenMobileHostIsDisconnected;
}

/**
 * Setter of the property
  <tt>lostLocksWhenMobileHostIsDisconnected</tt>
 *
 * @param lostLocksWhenMobileHostIsDisconnected
 *          The lostLocksWhenMobileHostIsDisconnected to set.
 * @uml.property name="lostLocksWhenMobileHostIsDisconnected"
 */
public void setLostLocksWhenMobileHostIsDisconnected(
```

```
    Collection<Lock> lostLocksWhenMobileHostIsDisconnected) {
  this.lostLocksWhenMobileHostIsDisconnected =
    lostLocksWhenMobileHostIsDisconnected;
}

/**
 * Getter of the property <tt>activeTransactions</tt>
 *
 * @return Returns the activeTransactions.
 * @uml.property name="activeTransactions"
 */
public Collection<Transaction> getActiveTransactions() {
  return activeTransactions;
}

/**
 * Setter of the property <tt>activeTransactions</tt>
 *
 * @param activeTransactions
 *          The activeTransactions to set.
 * @uml.property name="activeTransactions"
 */
public void setActiveTransactions(
    Collection<Transaction> activeTransactions) {
  this.activeTransactions = activeTransactions;
}

public Collection<Lock> getGainedLocksWhenMobileHostIsDisconnected()
  {
  return gainedLocksWhenMobileHostIsDisconnected;
}

public void setGainedLocksWhenMobileHostIsDisconnected(
    Collection<Lock> gainedLocksWhenMobileHostIsDisconnected) {
  this.gainedLocksWhenMobileHostIsDisconnected =
    gainedLocksWhenMobileHostIsDisconnected;
}

/**
 * Getter of the property <tt>transactionManager</tt>
 *
 * @return Returns the transactionManager.
 * @uml.property name="transactionManager"
 */
public TransactionManager getTransactionManager() {
  return transactionManager;
}

/**
```

```
 * Setter of the property <tt>transactionManager</tt>
 *
 * @param transactionManager
 *          The transactionManager to set.
 * @uml.property name="transactionManager"
 */
public void setTransactionManager(
    TransactionManager transactionManager) {
  this.transactionManager = transactionManager;
}

@SuppressWarnings("unused")
private void readInputFile(String inputFile) {
  if (inputFile != null) {
    inputFileReader = new InputFileReader(inputFile);
    String line = inputFileReader.readLine();
    if (line == null || !line.startsWith("CREATE_SERVER")) {
      unexpectedInput();
    } else {
      setId(line.substring(14));
      System.out.println("A new server is created with id "
          + line.substring(14));
      line = inputFileReader.readLine();
      if (line == null || !line.startsWith("NOF_RESOURCES")) {
        unexpectedInput();
      } else {
        int numberOfResources = new Integer(line.substring(14));
        setDatabaseSize(numberOfResources);
        createResources(numberOfResources);
      }
    }
    line = inputFileReader.readLine();
    if (line == null
        || !line.startsWith("NOF_MOBILESUPPORTSTATIONS")) {
      unexpectedInput();
    } else {
      int numberOfMSS = new Integer(line.substring(26));
      for (int i = 0; i < numberOfMSS; i++) {
        line = inputFileReader.readLine();
        if (line == null
            || !line.startsWith("CREATE_MOBILESUPPORTSTATION")) {
          unexpectedInput();
        } else {
          String subString = line.substring(28);
          String[] params = subString.split(":");
          MobileSupportStation mss = new MobileSupportStation(
              params[0], this);
          getMobileSupportStation().add(mss);
          System.out
```

```
            .println("A new mobile support station is created with
              id "+ params[0]);
      }
    }

  }
  line = inputFileReader.readLine();
  if (line == null || !line.startsWith("NOF_MOBILEHOSTS")) {
    unexpectedInput();
  } else {
    int numberOfMH = new Integer(line.substring(16));
    for (int i = 0; i < numberOfMH; i++) {
      line = inputFileReader.readLine();
      if (line == null || !line.startsWith("CREATE_MOBILEHOST")) {
        unexpectedInput();
      } else {
        String subString = line.substring(18);
        String[] params = subString.split(":");
        MobileHost mh = new MobileHost(params[0],
            getMobileSupportStation(params[1]), new Integer(
                params[2]), new Integer(params[3]));
        getMobileSupportStation(params[1]).register(mh);
        System.out
            .println("A new mobile host is created with id "
                + params[0]);
      }
    }
  }
  line = inputFileReader.readLine();
  while (!(line == null || line.startsWith("END_FILE"))) {
    if (line.startsWith("START_ONLINETRANSACTION")) {
      String subString = line.substring(24);
      String[] params = subString.split(":");

      MobileHost mh = getMobileHost(params[0]);
      mh.startNewOnlineTransaction(params[1]);
    }
    if (line.startsWith("START_OFFLINETRANSACTION")) {
      String subString = line.substring(25);
      String[] params = subString.split(":");

      MobileHost mh = getMobileHost(params[0]);
      mh.startNewOfflineTransaction(params[1]);
    }
    if (line.startsWith("READ_ONLINETRANSACTION")) {
      String subString = line.substring(23);
      String[] params = subString.split(":");

      MobileHost mh = getMobileHost(params[0]);
```

```
  mh.getMobileSupportStation().getDatabaseServer()
      .getTransactionManager().executeReadOperation(
          params[1], params[2], mh.getId());
}
if (line.startsWith("READ_OFFLINETRANSACTION")) {
  String subString = line.substring(24);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeReadOperation(
      params[1], params[2]);
}
if (line.startsWith("WRITE_ONLINETRANSACTION")) {
  String subString = line.substring(24);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getMobileSupportStation().getDatabaseServer()
      .getTransactionManager().executeWriteOperation(
          params[1], params[2], new Integer(params[3]),
          mh.getId());
}

if (line.startsWith("WRITE_OFFLINETRANSACTION")) {
  String subString = line.substring(25);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeWriteOperation(
      params[1], params[2], new Integer(params[3]));
}
if (line.startsWith("COMMIT_ONLINETRANSACTION")) {
  String subString = line.substring(25);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getMobileSupportStation().getDatabaseServer()
      .getTransactionManager().executeCommitOperation(
          params[1], mh.getId());
}
if (line.startsWith("COMMIT_OFFLINETRANSACTION")) {
  String subString = line.substring(26);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeCommitOperation(
      params[1]);
}
if (line.startsWith("ABORT_ONLINETRANSACTION")) {
```

```
  String subString = line.substring(24);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getMobileSupportStation().getDatabaseServer()
      .getTransactionManager().executeAbortOperation(
          params[1], mh.getId());
}
if (line.startsWith("ABORT_OFFLINETRANSACTION")) {
  String subString = line.substring(25);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().executeAbortOperation(
      params[1]);
}
if (line.startsWith("REQUEST_RONLOCK")) {
  String subString = line.substring(16);
  String[] params = subString.split(":");

  getTransactionManager().requestROnLock(params[1],
      params[2], params[0]);
}
if (line.startsWith("REQUEST_WONLOCK")) {
  String subString = line.substring(16);
  String[] params = subString.split(":");
  if (getTransactionManager().getTransaction(params[1],
      params[0]) != null) {
    getTransactionManager().requestWOnLock(params[1],
        params[2], params[0]);
  } else {
    MobileHost mh = getMobileHost(params[0]);
    mh.getLocalTransactionManager().requestWOnLock(
        params[1], params[2]);
  }
}
if (line.startsWith("REQUEST_WIOFFLOCK")) {
  String subString = line.substring(18);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().requestWIOffLock(
      params[1], params[2]);
}
if (line.startsWith("REQUEST_WOFFLOCK")) {
  String subString = line.substring(17);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
```

```
  mh.getLocalTransactionManager().requestWOffLock(params[1],
      params[2]);
}
if (line.startsWith("REQUEST_BROWSELOCK_ONLINETRANSACTION")) {
  String subString = line.substring(37);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getMobileSupportStation().getDatabaseServer()
      .getTransactionManager().requestBrowseLock(params[1],
          params[2], mh.getId());
}
if (line.startsWith("REQUEST_BROWSELOCK_OFFLINETRANSACTION"))
  {
  String subString = line.substring(38);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager().requestBrowseLock(
      params[1], params[2]);
}
if (line.startsWith("DISCONNECT_PLANNED")) {
  String subString = line.substring(19);
  MobileHost mh = getMobileHost(subString);
  mh.disconnectPlanned();
}
if (line.startsWith("DISCONNECT_UNPLANNED")) {
  String subString = line.substring(21);
  MobileHost mh = getMobileHost(subString);
  mh.disconnectUnplanned();
}
if (line.startsWith("RECONNECT")) {
  String subString = line.substring(10);
  MobileHost mh = getMobileHost(subString);
  mh.reconnect();
}
if (line.startsWith("INITIATE_MA")) {
  String subString = line.substring(12);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  mh.getLocalTransactionManager()
      .initiatieMobileAffiliation(params[1], params[2]);
}
if (line.startsWith("REGISTER_MA")) {
  String subString = line.substring(12);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
```

```
  MobileHost iniatorMobileHost = getMobileHost(params[2]);
  Transaction initiator = iniatorMobileHost
      .getLocalTransactionManager()
      .getTransaction(params[3]);
  mh.getLocalTransactionManager()
      .registerToMobileAffiliation(params[1], initiator,
          params[4]);
}
if (line.startsWith("UNREGISTER_MA")) {
  String subString = line.substring(14);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);

  MobileHost iniatorMobileHost = getMobileHost(params[2]);
  Transaction initiator = iniatorMobileHost
      .getLocalTransactionManager()
      .getTransaction(params[3]);
  mh.getLocalTransactionManager()
      .unRegisterFromMobileAffiliation(params[1], initiator,
          params[4]);
}
if (line.startsWith("EXPORT_SHARE_TO_READ")) {
  String subString = line.substring(21);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  MobileHost initiatorMobileHost = getMobileHost(params[3]);
  mh.getLocalTransactionManager().shareToRead(params[6],
      params[1], params[2], initiatorMobileHost, params[4],
      params[5]);
}
if (line.startsWith("EXPORT_SHARE_TO_WRITE")) {
  String subString = line.substring(22);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  MobileHost initiatorMobileHost = getMobileHost(params[3]);
  mh.getLocalTransactionManager().shareToWrite(params[6],
      params[1], params[2], initiatorMobileHost, params[4],
      params[5]);
}
if (line.startsWith("IMPORT_SHARE_TO_READ")) {
  String subString = line.substring(21);
  String[] params = subString.split(":");

  MobileHost mh = getMobileHost(params[0]);
  MobileHost initiatorMobileHost = getMobileHost(params[3]);
```

```
        mh.getLocalTransactionManager()
            .importShareToReadSharedData(params[6], params[1],
                params[2], initiatorMobileHost, params[4],
                params[5]);
      }
      if (line.startsWith("IMPORT_SHARE_TO_WRITE")) {
        String subString = line.substring(22);
        String[] params = subString.split(":");

        MobileHost mh = getMobileHost(params[0]);
        MobileHost initiatorMobileHost = getMobileHost(params[3]);
        mh.getLocalTransactionManager()
            .importShareToWriteSharedData(params[6], params[1],
                params[2], initiatorMobileHost, params[4],
                params[5]);
      }

      line = inputFileReader.readLine();
    }
    inputFileReader.close();
  }

}

public MobileHost getMobileHost(String id) {
  for (Iterator<MobileSupportStation> i = mobileSupportStation
      .iterator(); i.hasNext();) {
    MobileSupportStation mss = i.next();
    for (Iterator<MobileHost> j = mss.getMobileHost().iterator(); j
        .hasNext();) {
      MobileHost mh = j.next();
      if (mh.getId().equals(id)) {
        return mh;
      }
    }
  }
  return null;
}

/**
 */
private void unexpectedInput() {
  System.out
      .println("ERROR: Input file is incorrectly formatted. Program
        aborted.");
  System.exit(1);
}

/**
```

```
 */
private void addResource(Resource resource) {
  this.resource.add(resource);
}

private MobileSupportStation getMobileSupportStation(String id) {
  for (Iterator<MobileSupportStation> i = mobileSupportStation
      .iterator(); i.hasNext();) {
    MobileSupportStation mss = i.next();
    if (mss.getId().equals(id)) {
      return mss;
    }
  }
  return null;
}

private void createResources(int numberOfResources) {
  double maxValue = 10;
  for (int i = 0; i < numberOfResources; i++) {
    int resourceValue = new Double(Math.random() * (maxValue + 1))
        .intValue();
    System.out.println("A new resource is created with id " + i
        + " and value " + resourceValue);
    addResource(new Resource(this, resourceValue, "" + i));
  }
}

/**
 * @param args
 */
public static void main(String[] args) {
  DatabaseServer db = new DatabaseServer("0", 0, 0);
  db.readInputFile("../input/adaptiv/testcase4.txt");
  db.getLockLog().printLockLog();
  db.getLog().printLog();
  for (Iterator<MobileHost> i = db.getMobileSupportStation("0")
      .getMobileHost().iterator(); i.hasNext();) {
    MobileHost mh = i.next();
    mh.getLocalLockLog().printLocalLockLog();
    mh.getLocalLog().printLocalLog();
  }
  db.getConnectivityManager().setTerminate(true);
}

/**
 * Getter of the property <tt>dependentCommitOperations</tt>
 * @return  Returns the dependentCommitOperations.
 * @uml.property  name="dependentCommitOperations"
 */
```

```
  public Collection<CommitOperation> getDependentCommitOperations() {
    return dependentCommitOperations;
  }

  /**
   * Setter of the property <tt>dependentCommitOperations</tt>
   * @param dependentCommitOperations  The dependentCommitOperations
     to set.
   * @uml.property  name="dependentCommitOperations"
   */
  public void setDependentCommitOperations(
      Collection<CommitOperation> dependentCommitOperations) {
    this.dependentCommitOperations = dependentCommitOperations;
  }
}
```

C.1.2 LockManager

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import transaction.Transaction;

import lock.BrowseLock;
import lock.Lock;
import lock.ROnLock;
import lock.WIOffLock;
import lock.WOffLock;
import lock.WOnLock;
import mobileHost.LocalLogRecord;

public class LockManager {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd inverse="lockManager:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  public LockManager(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  /**
```

```
 */
public boolean handleROnLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();

  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }

  if (currentLocks.isEmpty()) {
    // No locks
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "RONLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new ROnLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // Other readlocks
    boolean readLocks = true;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof ROnLock) {
      } else {
        readLocks = false;
      }
    }
    if (readLocks) {
      boolean alreadyGotPendingWIOffLock = false;
      Lock wIOffLockWhichTransactionHasPending = null;
      for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();) {
        Lock lock = i.next();
        if (lock instanceof WIOffLock) {
          if (lock.getTransaction().getId().equals(
              transaction.getId())) {
```

```
            alreadyGotPendingWIOffLock = true;
            wIOffLockWhichTransactionHasPending = lock;
          }
        } else {
        }
      }
      if (alreadyGotPendingWIOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADE");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, false,
            true, wIOffLockWhichTransactionHasPending);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTED");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        transaction.getLocks().add(newLock);
        return true;
      }
    }
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotWIOffLock = true;
          wIOffLockWhichTransactionIsAlreadyHolding = lock;
        }
      } else {
```

```
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      if (alreadyGotWIOffLock) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDUPGRADE");
        Lock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, wIOffLockWhichTransactionIsAlreadyHolding);
        return true;
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDANDTAKEOVER");
        pendingLocks.addAll(currentLocks);
        currentLocks.clear();
        ROnLock newLock = new ROnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().add(newLock);
        return true;
      }
    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "RONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWOnLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
```

```
Collection<Lock> pendingLocks = resource.getPendingLocks();
if (!currentLocks.isEmpty()) {
  for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock instanceof WOffLock
        && (lock.getTransaction().getId().equals(transaction
            .getId()))) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WONLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDSELFUPGRADE");
      Lock newLock = new WOnLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      upgradeLock(newLock, currentLocks, pendingLocks, true,
          false, lock);
      for (Iterator<Lock> j = pendingLocks.iterator(); j
          .hasNext();) {
        Lock browseLock = j.next();
        if (browseLock instanceof BrowseLock) {
          BrowseLock browseLock2 = (BrowseLock) browseLock;
          browseLock2.setGrantedWOnLock((WOnLock) newLock);
          notifyAboutWOffUpgrade(browseLock);
        }
      }

      return true;
    }
  }
}
if (resource.getAllowMoreLocks() == false) {
  databaseServer.getLockLog()
      .addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WONLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "REJECTED");
  return false;
}
if (currentLocks.isEmpty()) {
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
  Lock newLock = new WOnLock(transaction, resource);
```

```
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    resource.setAllowMoreLocks(false);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotWIOffLock = true;
          wIOffLockWhichTransactionIsAlreadyHolding = lock;
        }
      } else {
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      if (alreadyGotWIOffLock) {
        if (currentLocks.size() == 1) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "WONLOCK", getLocks(currentLocks),
              getLocks(pendingLocks), "GRANTEDUPGRADE");
          Lock newLock = new WOnLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, wIOffLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        } else {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "WONLOCK", getLocks(currentLocks),
              getLocks(pendingLocks), "GRANTEDUPGRADEANDDELETE");
          Lock newLock = new WOnLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
```

```
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, wIOffLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        }
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WONLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDANDTAKEOVERANDDELETE");
        for (Iterator<Lock> i = currentLocks.iterator(); i
            .hasNext();) {
          Lock lock = i.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        WOnLock newLock = new WOnLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().add(newLock);
        resource.setAllowMoreLocks(false);
        return true;
      }

    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WONLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWIOffLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();
  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
```

```
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }
  if (currentLocks.isEmpty()) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WIOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new WIOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // Other WIOffLocks
    boolean wioffLocks = true;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
      } else {
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTED");
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      currentLocks.add(newLock);
      transaction.getLocks().add(newLock);
      return true;
    }
  }
  if (!currentLocks.isEmpty()) {
    // readlocks
    boolean readLocks = true;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
```

```
      if (lock instanceof ROnLock) {
      } else {
        readLocks = false;
      }
    }
    if (readLocks) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WIOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDPENDING");
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      pendingLocks.add(newLock);
      transaction.getLocks().add(newLock);
      return true;
    }
  }
  databaseServer.getLockLog().addLockLogRecord(
      new Integer(databaseServer.getLockLogId()).toString(),
      databaseServer.getTime(), transaction.getOwner().getId(),
      transaction.getId(), resource.getId(), "WIOFFLOCK",
      getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
  return false;
}

/**
 */
public boolean handleWOffLockRequest(Resource resource,
    Transaction transaction) {
  Collection<Lock> currentLocks = resource.getCurrentLocks();
  Collection<Lock> pendingLocks = resource.getPendingLocks();
  if (resource.getAllowMoreLocks() == false) {
    databaseServer.getLockLog()
        .addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "REJECTED");
    return false;
  }
  if (currentLocks.isEmpty()) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WOFFLOCK",
```

```
        getLocks(currentLocks), getLocks(pendingLocks), "GRANTED");
    Lock newLock = new WOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    currentLocks.add(newLock);
    transaction.getLocks().add(newLock);
    return true;
  }
  if (!currentLocks.isEmpty()) {
    // WIOffLocks
    boolean wioffLocks = true;
    boolean alreadyGotWIOffLock = false;
    Lock wIOffLockWhichTransactionIsAlreadyHolding = null;
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof WIOffLock) {
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotWIOffLock = true;
          wIOffLockWhichTransactionIsAlreadyHolding = lock;
        }
      } else {
        wioffLocks = false;
      }
    }
    if (wioffLocks) {
      if (alreadyGotWIOffLock) {
        if (currentLocks.size() == 1) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "GRANTEDUPGRADE");
          Lock newLock = new WOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, wIOffLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        } else {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
```

```
            "GRANTEDUPGRADEANDDELETE");
          Lock newLock = new WOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, wIOffLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        }
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDANDTAKEOVER");
        for (Iterator<Lock> i = currentLocks.iterator(); i
            .hasNext();) {
          Lock lock = i.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        WOffLock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        currentLocks.add(newLock);
        transaction.getLocks().add(newLock);
        resource.setAllowMoreLocks(false);
        return true;
      }
    }
  }
  if (!currentLocks.isEmpty()) {
    // readlocks
    boolean readLocks = true;
    boolean alreadyGotROnLock = false;
    Lock rOnLockWhichTransactionIsAlreadyHolding = null;
    boolean alreadyGotPendingWIOffLock = false;
    Lock wIOffLockWhichTransactionHasPending = null;
    ArrayList<Lock> rOnLocks = new ArrayList<Lock>();
    for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock instanceof ROnLock) {
        rOnLocks.add(lock);
        if (lock.getTransaction().getId().equals(
            transaction.getId())) {
          alreadyGotROnLock = true;
          rOnLockWhichTransactionIsAlreadyHolding = lock;
          rOnLocks.remove(lock);
```

```
      }
    } else {
      readLocks = false;
    }
  }
  for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock instanceof WIOffLock) {
      if (lock.getTransaction().getId().equals(
          transaction.getId())) {
        alreadyGotPendingWIOffLock = true;
        wIOffLockWhichTransactionHasPending = lock;
      }
    }
  }
  if (readLocks) {
    if (alreadyGotROnLock) {
      if (currentLocks.size() == 1) {
        if (pendingLocks.isEmpty()) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(),
              transaction.getId(), resource.getId(), "WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "GRANTEDUPGRADE");
          Lock newLock = new WOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, rOnLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        } else {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(),
              transaction.getId(), resource.getId(), "WOFFLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "GRANTEDUPGRADEANDDELETE");
          Lock newLock = new WOffLock(transaction, resource);
          Integer lockLogId = databaseServer.getLockLogId() - 1;
          newLock.setLockLogId(lockLogId.toString());
          upgradeLock(newLock, currentLocks, pendingLocks, true,
              false, rOnLockWhichTransactionIsAlreadyHolding);
          resource.setAllowMoreLocks(false);
          return true;
        }
```

```
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "WOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "GRANTEDUPGRADEPENDING");
        Lock newLock = new WOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        upgradeLock(newLock, currentLocks, pendingLocks, true,
            false, rOnLockWhichTransactionIsAlreadyHolding);
        resource.setAllowMoreLocks(false);
        notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
        return true;
      }
    } else if (alreadyGotPendingWIOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDUPGRADEPENDING");
      Lock newLock = new WOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      upgradeLock(newLock, currentLocks, pendingLocks, false,
          true, wIOffLockWhichTransactionHasPending);
      resource.setAllowMoreLocks(false);
      notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
      return true;
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "WOFFLOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDPENDING");
      Lock newLock = new WOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      pendingLocks.add(newLock);
      transaction.getLocks().add(newLock);
      resource.setAllowMoreLocks(false);
      notifyROnLockOwnersAboutGrantedWOffLock(rOnLocks);
      return true;
    }
  }
```

```
    }
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(), transaction.getOwner().getId(),
        transaction.getId(), resource.getId(), "WOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks), "REJECTED");
    return false;
  }

  /**
   */
  public boolean handleBrowseLockRequest(Resource resource,
      Transaction transaction) {
    Collection<Lock> currentLocks = resource.getCurrentLocks();
    Collection<Lock> pendingLocks = resource.getPendingLocks();
    if (!currentLocks.isEmpty()) {
      // WOffLock?
      boolean wOffLockInCurrentLocks = false;
      for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();) {
        Lock lock = i.next();
        if (lock instanceof WOffLock) {
          wOffLockInCurrentLocks = true;
        }
      }
      if (wOffLockInCurrentLocks) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId()).toString(),
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "BROWSELOCK", getLocks(currentLocks),
            getLocks(pendingLocks), "GRANTEDPENDING");
        BrowseLock newLock = new BrowseLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        transaction.getLocks().add(newLock);
        pendingLocks.add(newLock);
        return true;
      }
    }
    if (!pendingLocks.isEmpty()) {
      boolean wOffLockInPendingLocks = false;
      for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();) {
        Lock lock = i.next();
        if (lock instanceof WOffLock) {
          wOffLockInPendingLocks = true;
        }
      }
      if (wOffLockInPendingLocks) {
        databaseServer.getLockLog().addLockLogRecord(
```

```
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "BROWSELOCK", getLocks(currentLocks),
          getLocks(pendingLocks), "GRANTEDPENDING");
      BrowseLock newLock = new BrowseLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      transaction.getLocks().add(newLock);
      pendingLocks.add(newLock);
      return true;
    }
  }
  return false;
}

/**
 */
public void releaseLocks(Transaction transaction) {
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    Resource resource = lock.getResource();
    Collection<Lock> currentLocks = lock.getResource()
        .getCurrentLocks();
    Collection<Lock> pendingLocks = lock.getResource()
        .getPendingLocks();
    if (currentLocks.contains(lock)) {
      if (lock instanceof ROnLock) {
        if (currentLocks.size() == 1 && pendingLocks.size() > 0) {
          boolean existPendingWOffLock = false;
          Lock pendingWOffLock = null;
          for (Iterator<Lock> j = pendingLocks.iterator(); j
              .hasNext();) {
            Lock lock2 = j.next();
            if (lock2 instanceof WOffLock) {
              existPendingWOffLock = true;
              pendingWOffLock = lock2;
            }
          }
          if (existPendingWOffLock) {
            databaseServer.getLockLog().addLockLogRecord(
                new Integer(databaseServer.getLockLogId())
                    .toString(), databaseServer.getTime(),
                transaction.getOwner().getId(),
                transaction.getId(), resource.getId(),
                "RELEASE_RONLOCK", getLocks(currentLocks),
                getLocks(pendingLocks),
                "RELEASEDDELEGATEWOFFLOCKSANDDELETE");
```

```
            currentLocks.remove(lock);
            Collection<Lock> newCurrentLock = new ArrayList<Lock>();
            Collection<Lock> newPendingLock = new ArrayList<Lock>();
            newCurrentLock.add(pendingWOffLock);
            pendingLocks.remove(pendingWOffLock);
            for (Iterator<Lock> j = pendingLocks.iterator(); j
                .hasNext();) {
              Lock lock2 = j.next();
              if (lock2 instanceof WIOffLock) {
                notifyWIOffOwnerAboutLostWIOffLock(lock2);
                pendingLocks.remove(lock2);
              }
              if (lock2 instanceof BrowseLock) {
                // Dont have to do anything. These locks will remain
                   pending.
              }
            }
            delegateLock(resource, newCurrentLock, newPendingLock);
          } else {
            databaseServer.getLockLog().addLockLogRecord(
                new Integer(databaseServer.getLockLogId())
                    .toString(), databaseServer.getTime(),
                transaction.getOwner().getId(),
                transaction.getId(), resource.getId(),
                "RELEASE_RONLOCK", getLocks(currentLocks),
                getLocks(pendingLocks),
                "RELEASEDDELEGATEWIOFFLOCKS");
            currentLocks.remove(lock);
            delegateLock(resource, pendingLocks, null);
          }
        } else {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "RELEASE_RONLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "RELEASED");
          currentLocks.remove(lock);
        }
      } else if (lock instanceof WOnLock) {
        if (pendingLocks.size() > 0) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
              resource.getId(), "RELEASE_WONLOCK",
              getLocks(currentLocks), getLocks(pendingLocks),
              "RELEASEDDELEGATEBROWSELOCKS");
```

```
currentLocks.remove(lock);
resource.setAllowMoreLocks(true);

Integer lockLogId = databaseServer.getLockLogId() - 1;
boolean onlineTransactionBrowseLocks = false;
if (!lock.getResource().getPendingLocks().isEmpty()) {
  for (Iterator<Lock> j = lock.getResource()
      .getPendingLocks().iterator(); j.hasNext();) {
    Lock browseLock = j.next();
    if (browseLock instanceof BrowseLock) {
      if (browseLock.getTransaction()
          .getOnlineTransaction()) {
        onlineTransactionBrowseLocks = true;
      }
    }
  }
}

Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
Collection<Lock> newPendingLocks = new ArrayList<Lock>();
Collection<Lock> browseLocksToRemove = new
  ArrayList<Lock>();
for (Iterator<Lock> j = pendingLocks.iterator(); j
    .hasNext();) {
  Lock browseLock = j.next();
  if (browseLock instanceof BrowseLock) {
    browseLocksToRemove.add(browseLock);
    if (browseLock.getTransaction()
        .getOnlineTransaction()) {
      Lock newROnLock = new ROnLock(browseLock
          .getTransaction(), resource);
      newROnLock.setLockLogId(lockLogId.toString());
      newCurrentLocks.add(newROnLock);
      browseLock.getTransaction().getLocks().remove(
          browseLock);
      browseLock.getTransaction().getLocks().add(
          newROnLock);
    } else {
      if (onlineTransactionBrowseLocks) {
        Lock newWIOffLock = new WIOffLock(browseLock
            .getTransaction(), resource);
        newWIOffLock.setLockLogId(lockLogId.toString());
        newPendingLocks.add(newWIOffLock);
        if (browseLock.getTransaction().getOwner()
            .getConnectionStatus()) {
          browseLock.getTransaction().getLocks().remove(
              browseLock);
          browseLock.getTransaction().getLocks().add(
              newWIOffLock);
```

```
            } else {
              databaseServer
                  .getLostLocksWhenMobileHostIsDisconnected()
                  .add(browseLock);
              databaseServer
                  .getGainedLocksWhenMobileHostIsDisconnected()
                  .add(newWIOffLock);
            }
          } else {
            Lock newWIOffLock = new WIOffLock(browseLock
                .getTransaction(), resource);
            newWIOffLock.setLockLogId(lockLogId.toString());
            newCurrentLocks.add(newWIOffLock);
            if (browseLock.getTransaction().getOwner()
                .getConnectionStatus()) {
              browseLock.getTransaction().getLocks().remove(
                  browseLock);
              browseLock.getTransaction().getLocks().add(
                  newWIOffLock);
            } else {
              databaseServer
                  .getLostLocksWhenMobileHostIsDisconnected()
                  .add(browseLock);
              databaseServer
                  .getGainedLocksWhenMobileHostIsDisconnected()
                  .add(newWIOffLock);
            }
          }
        }
      }
      pendingLocks.removeAll(browseLocksToRemove);
      delegateLock(resource, newCurrentLocks, newPendingLocks);
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_WONLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      currentLocks.remove(lock);
      resource.setAllowMoreLocks(true);
    }
  } else if (lock instanceof WIOffLock) {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId()).toString(),
        databaseServer.getTime(),
        transaction.getOwner().getId(), transaction.getId(),
```

```
        resource.getId(), "RELEASE_WIOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks),
        "RELEASED");
    currentLocks.remove(lock);
  } else if (lock instanceof WOffLock) {
    if (pendingLocks.size() > 0) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
              .toString(), databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_WOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASEDDELEGATEBROWSELOCKS");
      currentLocks.remove(lock);
      resource.setAllowMoreLocks(true);

      Integer lockLogId = databaseServer.getLockLogId() - 1;
      boolean onlineTransactionBrowseLocks = false;
      if (!lock.getResource().getPendingLocks().isEmpty()) {
        for (Iterator<Lock> j = lock.getResource()
            .getPendingLocks().iterator(); j.hasNext();) {
          Lock browseLock = j.next();
          if (browseLock instanceof BrowseLock) {
            if (browseLock.getTransaction()
                .getOnlineTransaction()) {
              onlineTransactionBrowseLocks = true;
            }
          }
        }
      }

      Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
      Collection<Lock> newPendingLocks = new ArrayList<Lock>();
      Collection<Lock> browseLocksToRemove = new
        ArrayList<Lock>();
      for (Iterator<Lock> j = pendingLocks.iterator(); j
          .hasNext();) {
        Lock browseLock = j.next();
        if (browseLock instanceof BrowseLock) {
          browseLocksToRemove.add(browseLock);
          if (browseLock.getTransaction()
              .getOnlineTransaction()) {
            Lock newROnLock = new ROnLock(browseLock
                .getTransaction(), resource);
            newROnLock.setLockLogId(lockLogId.toString());
            newCurrentLocks.add(newROnLock);
            browseLock.getTransaction().getLocks().remove(
                browseLock);
            browseLock.getTransaction().getLocks().add(
```

```
            newROnLock);
        } else {
          if (onlineTransactionBrowseLocks) {
            Lock newWIOffLock = new WIOffLock(browseLock
                .getTransaction(), resource);
            newWIOffLock.setLockLogId(lockLogId.toString());
            newPendingLocks.add(newWIOffLock);
            if (browseLock.getTransaction().getOwner()
                .getConnectionStatus()) {
              browseLock.getTransaction().getLocks().remove(
                  browseLock);
              browseLock.getTransaction().getLocks().add(
                  newWIOffLock);
            } else {
              databaseServer
                  .getLostLocksWhenMobileHostIsDisconnected()
                  .add(browseLock);
              databaseServer
                  .getGainedLocksWhenMobileHostIsDisconnected()
                  .add(newWIOffLock);
            }
          } else {
            Lock newWIOffLock = new WIOffLock(browseLock
                .getTransaction(), resource);
            newWIOffLock.setLockLogId(lockLogId.toString());
            newCurrentLocks.add(newWIOffLock);
            if (browseLock.getTransaction().getOwner()
                .getConnectionStatus()) {
              browseLock.getTransaction().getLocks().remove(
                  browseLock);
              browseLock.getTransaction().getLocks().add(
                  newWIOffLock);
            } else {
              databaseServer
                  .getLostLocksWhenMobileHostIsDisconnected()
                  .add(browseLock);
              databaseServer
                  .getGainedLocksWhenMobileHostIsDisconnected()
                  .add(newWIOffLock);
            }
          }
        }
      }
      pendingLocks.removeAll(browseLocksToRemove);
      delegateLock(resource, newCurrentLocks, newPendingLocks);
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId())
```

```
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_WOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASED");
        currentLocks.remove(lock);
        resource.setAllowMoreLocks(true);
      }
    }
  } else if (pendingLocks.contains(lock)) {
    if (lock instanceof WIOffLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "RELEASE_PENDINGWIOFFLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "RELEASED");
      pendingLocks.remove(lock);
    } else if (lock instanceof WOffLock) {

      boolean pendingBrowseLocks = false;
      for (Iterator<Lock> j = pendingLocks.iterator(); j
          .hasNext();) {
        Lock lock2 = j.next();
        if (lock2 instanceof BrowseLock) {
          pendingBrowseLocks = true;
        }
      }
      if (pendingBrowseLocks) {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_PENDINGWOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASEDDELEGATEBROWSELOCKS");
        pendingLocks.remove(lock);
        resource.setAllowMoreLocks(true);

        Integer lockLogId = databaseServer.getLockLogId() - 1;
        Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
        Collection<Lock> newPendingLocks = new ArrayList<Lock>();
        Collection<Lock> browseLocksToRemove = new
          ArrayList<Lock>();
        for (Iterator<Lock> j = pendingLocks.iterator(); j
            .hasNext();) {
          Lock browseLock = j.next();
          if (browseLock instanceof BrowseLock) {
```

```
            browseLocksToRemove.add(browseLock);
            if (browseLock.getTransaction()
                .getOnlineTransaction()) {
              Lock newROnLock = new ROnLock(browseLock
                  .getTransaction(), resource);
              newROnLock.setLockLogId(lockLogId.toString());
              newCurrentLocks.add(newROnLock);
              browseLock.getTransaction().getLocks().remove(
                  browseLock);
              browseLock.getTransaction().getLocks().add(
                  newROnLock);
            } else {
              Lock newWIOffLock = new WIOffLock(browseLock
                  .getTransaction(), resource);
              newWIOffLock.setLockLogId(lockLogId.toString());
              newPendingLocks.add(newWIOffLock);
              if (browseLock.getTransaction().getOwner()
                  .getConnectionStatus()) {
                browseLock.getTransaction().getLocks().remove(
                    browseLock);
                browseLock.getTransaction().getLocks().add(
                    newWIOffLock);
              } else {
                databaseServer
                    .getLostLocksWhenMobileHostIsDisconnected()
                    .add(browseLock);
                databaseServer
                    .getGainedLocksWhenMobileHostIsDisconnected()
                    .add(newWIOffLock);
              }
            }
          }
        }
        pendingLocks.removeAll(browseLocksToRemove);
        delegateLock(resource, newCurrentLocks, newPendingLocks);
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_PENDINGWOFFLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASED");
        pendingLocks.remove(lock);
        resource.setAllowMoreLocks(true);
      }
    } else if (lock instanceof BrowseLock) {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
```

```
            databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "RELEASE_PENDINGBROWSELOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "RELEASED");
        pendingLocks.remove(lock);
      }
    }
  }
  transaction.getLocks().clear();
}

/**
 */
public void upgradeLock(Lock newLock,
    Collection<Lock> currentLocks, Collection<Lock> pendingLocks,
    boolean alreadyGotCurrentLock, boolean alreadyGotPendingLock,
    Lock oldLock) {
  if (alreadyGotCurrentLock) {
    if (newLock instanceof ROnLock) {
      currentLocks.remove(oldLock);
      pendingLocks.addAll(currentLocks);
      currentLocks.clear();
      currentLocks.add(newLock);
      Transaction t = newLock.getTransaction();
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);

    } else if (newLock instanceof WOffLock) {
      if (oldLock instanceof ROnLock) {
        if (currentLocks.size() == 1) {
          if (pendingLocks.size() > 0) {
            for (Iterator<Lock> i = pendingLocks.iterator(); i
                .hasNext();) {
              Lock lock = i.next();
              notifyWIOffOwnerAboutLostWIOffLock(lock);
            }
            pendingLocks.clear();
          }
          currentLocks.remove(oldLock);
          currentLocks.add(newLock);
          Transaction t = newLock.getTransaction();
          t.getLocks().remove(oldLock);
          t.getLocks().add(newLock);
        } else {
          currentLocks.remove(oldLock);
          pendingLocks.add(newLock);
          Transaction t = newLock.getTransaction();
          t.getLocks().remove(oldLock);
```

```
          t.getLocks().add(newLock);
        }
      } else {
        currentLocks.remove(oldLock);
        for (Iterator<Lock> i = currentLocks.iterator(); i
            .hasNext();) {
          Lock lock = i.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock);
        }
        currentLocks.clear();
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      }
    } else if (newLock instanceof WOnLock) {
      if (oldLock instanceof WIOffLock) {
        currentLocks.remove(oldLock);
        if (currentLocks.size() > 0) {
          for (Iterator<Lock> i = currentLocks.iterator(); i
              .hasNext();) {
            Lock lock = i.next();
            notifyWIOffOwnerAboutLostWIOffLock(lock);
          }
          currentLocks.clear();
        }
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      } else {
        currentLocks.remove(oldLock);
        currentLocks.add(newLock);
        Transaction t = newLock.getTransaction();
        t.getLocks().remove(oldLock);
        t.getLocks().add(newLock);
      }
    }
  } else if (alreadyGotPendingLock) {
    if (newLock instanceof ROnLock) {
      pendingLocks.remove(oldLock);
      currentLocks.add(newLock);
      Transaction t = newLock.getTransaction();
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);
    } else if (newLock instanceof WOffLock) {
      pendingLocks.remove(oldLock);
      pendingLocks.add(newLock);
      Transaction t = newLock.getTransaction();
```

```
      t.getLocks().remove(oldLock);
      t.getLocks().add(newLock);
    }
  } else {
  }
}

/**
 */
public void delegateLock(Resource resource,
    Collection<Lock> newCurrentLocks,
    Collection<Lock> newPendingLocks) {
  resource.getCurrentLocks().addAll(newCurrentLocks);
  if (newPendingLocks == null) {
    resource.getPendingLocks().clear();
  } else {
    resource.getPendingLocks().addAll(newPendingLocks);
  }
}

/**
 */
public void handleDisconnectedTransaction(Transaction transaction,
    boolean planned) {
  Collection<Lock> locks = transaction.getLocks();
  Collection<Lock> locksToAdd = new ArrayList<Lock>();
  Collection<Lock> locksToRemove = new ArrayList<Lock>();
  for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    if (lock instanceof ROnLock) {
      Resource resource = lock.getResource();
      Collection<Lock> currentLocks = resource.getCurrentLocks();
      Collection<Lock> pendingLocks = resource.getPendingLocks();
      if (currentLocks.size() == 1) {
        boolean existPendingWOffLock = false;
        Lock pendingWOffLock = null;
        for (Iterator<Lock> j = pendingLocks.iterator(); j
            .hasNext();) {
          Lock lock2 = j.next();
          if (lock2 instanceof WOffLock) {
            existPendingWOffLock = true;
            pendingWOffLock = lock2;
          }
        }
        if (existPendingWOffLock) {
          databaseServer.getLockLog().addLockLogRecord(
              new Integer(databaseServer.getLockLogId())
                  .toString(), databaseServer.getTime(),
              transaction.getOwner().getId(), transaction.getId(),
```

```
            resource.getId(), "DELEGATE_RONLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "DISCONNECTEDDELEGATEANDDELETE");
        currentLocks.remove(lock);
        Collection<Lock> newCurrentLock = new ArrayList<Lock>();
        newCurrentLock.add(pendingWOffLock);
        pendingLocks.remove(pendingWOffLock);
        for (Iterator<Lock> j = pendingLocks.iterator(); j
            .hasNext();) {
          Lock lock2 = j.next();
          notifyWIOffOwnerAboutLostWIOffLock(lock2);
        }
        delegateLock(resource, newCurrentLock, null);
        locksToRemove.add(lock);
      } else {
        databaseServer.getLockLog().addLockLogRecord(
            new Integer(databaseServer.getLockLogId())
                .toString(), databaseServer.getTime(),
            transaction.getOwner().getId(), transaction.getId(),
            resource.getId(), "DELEGATE_RONLOCK",
            getLocks(currentLocks), getLocks(pendingLocks),
            "DISCONNECTEDDELEGATE");
        currentLocks.remove(lock);
        Lock newLock = new WIOffLock(transaction, resource);
        Integer lockLogId = databaseServer.getLockLogId() - 1;
        newLock.setLockLogId(lockLogId.toString());
        Collection<Lock> newCurrentLock = new ArrayList<Lock>();
        newCurrentLock.add(newLock);
        newCurrentLock.addAll(pendingLocks);
        delegateLock(resource, newCurrentLock, null);
        locksToRemove.add(lock);
        locksToAdd.add(newLock);
      }
    } else {
      databaseServer.getLockLog().addLockLogRecord(
          new Integer(databaseServer.getLockLogId()).toString(),
          databaseServer.getTime(),
          transaction.getOwner().getId(), transaction.getId(),
          resource.getId(), "DELEGATE_RONLOCK",
          getLocks(currentLocks), getLocks(pendingLocks),
          "DISCONNECTEDDELEGATE");
      currentLocks.remove(lock);
      Lock newLock = new WIOffLock(transaction, resource);
      Integer lockLogId = databaseServer.getLockLogId() - 1;
      newLock.setLockLogId(lockLogId.toString());
      Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
      Collection<Lock> newPendingLocks = new ArrayList<Lock>();
      newCurrentLocks.addAll(currentLocks);
      pendingLocks.add(newLock);
```

```
        newPendingLocks.addAll(pendingLocks);
        currentLocks.clear();
        pendingLocks.clear();
        delegateLock(resource, newCurrentLocks, newPendingLocks);
        locksToRemove.add(lock);
        locksToAdd.add(newLock);
      }
    }
  }
  if (planned) {
    locks.removeAll(locksToRemove);
    locks.addAll(locksToAdd);
  } else {
    databaseServer.getLostLocksWhenMobileHostIsDisconnected()
        .addAll(locksToRemove);
    databaseServer.getGainedLocksWhenMobileHostIsDisconnected()
        .addAll(locksToAdd);
  }
}

/**
 */
public void handleReconnectedTransaction(Transaction transaction) {
  Collection<Lock> locksToAdd = new ArrayList<Lock>();
  Collection<Lock> locksToRemove = new ArrayList<Lock>();
  for (Iterator<Lock> i = transaction.getLocks().iterator(); i
      .hasNext();) {
    Lock lock = i.next();
    if (lock.getLockType().equals("WOFFLOCK")) {
      Resource resource = lock.getResource();
      Collection<Lock> currentLocks = resource.getCurrentLocks();
      Collection<Lock> pendingLocks = resource.getPendingLocks();
      boolean valueChanged = false;
      for (Iterator<LocalLogRecord> j = transaction.getOwner()
          .getLocalLog().getLocalLogRecordsFromLastDisconnection(
              transaction.getOwner().getLastDisconnectionTime())
          .iterator(); j.hasNext();) {
        LocalLogRecord localLogRecord = j.next();
        if (localLogRecord.getTransactionId().equals(
            transaction.getId())
            && localLogRecord.getResourceId().equals(
                lock.getResource().getId())
            && localLogRecord.getAction().equals("WRITEOFFLINE")) {
          valueChanged = true;
        }
      }
      if (!valueChanged) {
        if (lock.getResource().getCurrentLocks().contains(lock)) {
          databaseServer.getLockLog().addLockLogRecord(
```

```
    new Integer(databaseServer.getLockLogId())
        .toString(), databaseServer.getTime(),
    transaction.getOwner().getId(), transaction.getId(),
    resource.getId(), "DELEGATE_WOFFLOCK",
    getLocks(currentLocks), getLocks(pendingLocks),
    "RECONNECTDELEGATE");
boolean onlineTransactionBrowseLocks = false;
if (!lock.getResource().getPendingLocks().isEmpty()) {
  for (Iterator<Lock> j = lock.getResource()
      .getPendingLocks().iterator(); j.hasNext();) {
    Lock browseLock = j.next();
    if (browseLock instanceof BrowseLock) {
      if (browseLock.getTransaction()
          .getOnlineTransaction()) {
        onlineTransactionBrowseLocks = true;
      }
    }
  }
}

currentLocks.remove(lock);
Lock newLock = new WIOffLock(transaction, resource);
Integer lockLogId = databaseServer.getLockLogId() - 1;
newLock.setLockLogId(lockLogId.toString());
Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
Collection<Lock> newPendingLocks = new ArrayList<Lock>();
if (onlineTransactionBrowseLocks) {
  newPendingLocks.add(newLock);
} else {
  newCurrentLocks.add(newLock);
}
locksToAdd.add(newLock);
locksToRemove.add(lock);
resource.setAllowMoreLocks(true);

Collection<Lock> browseLocksToRemove = new
  ArrayList<Lock>();
if (!pendingLocks.isEmpty()) {
  for (Iterator<Lock> j = pendingLocks.iterator(); j
      .hasNext();) {
    Lock browseLock = j.next();
    if (browseLock instanceof BrowseLock) {
      browseLocksToRemove.add(browseLock);
      if (browseLock.getTransaction()
          .getOnlineTransaction()) {
        Lock newROnLock = new ROnLock(browseLock
            .getTransaction(), resource);
        newROnLock.setLockLogId(lockLogId.toString());
        newCurrentLocks.add(newROnLock);
```

```
            browseLock.getTransaction().getLocks().remove(
                browseLock);
            browseLock.getTransaction().getLocks().add(
                newROnLock);
          } else {
            if (onlineTransactionBrowseLocks) {
              Lock newWIOffLock = new WIOffLock(browseLock
                  .getTransaction(), resource);
              newWIOffLock
                  .setLockLogId(lockLogId.toString());
              newPendingLocks.add(newWIOffLock);
              if (browseLock.getTransaction().getOwner()
                  .getConnectionStatus()) {
                browseLock.getTransaction().getLocks()
                    .remove(browseLock);
                browseLock.getTransaction().getLocks().add(
                    newWIOffLock);
              } else {
                databaseServer
                    .getLostLocksWhenMobileHostIsDisconnected()
                    .add(browseLock);
                databaseServer
                    .getGainedLocksWhenMobileHostIsDisconnected()
                    .add(newWIOffLock);
              }
            } else {
              Lock newWIOffLock = new WIOffLock(browseLock
                  .getTransaction(), resource);
              newWIOffLock
                  .setLockLogId(lockLogId.toString());
              newCurrentLocks.add(newWIOffLock);
              if (browseLock.getTransaction().getOwner()
                  .getConnectionStatus()) {
                browseLock.getTransaction().getLocks()
                    .remove(browseLock);
                browseLock.getTransaction().getLocks().add(
                    newWIOffLock);
              } else {
                databaseServer
                    .getLostLocksWhenMobileHostIsDisconnected()
                    .add(browseLock);
                databaseServer
                    .getGainedLocksWhenMobileHostIsDisconnected()
                    .add(newWIOffLock);
              }
            }
          }
        }
      }
```

```
      pendingLocks.removeAll(browseLocksToRemove);
    }
    delegateLock(resource, newCurrentLocks, newPendingLocks);
  } else {
    databaseServer.getLockLog().addLockLogRecord(
        new Integer(databaseServer.getLockLogId())
            .toString(), databaseServer.getTime(),
        transaction.getOwner().getId(), transaction.getId(),
        resource.getId(), "DELEGATE_WOFFLOCK",
        getLocks(currentLocks), getLocks(pendingLocks),
        "RECONNECTDELEGATE");
    pendingLocks.remove(lock);
    Lock newLock = new WIOffLock(transaction, resource);
    Integer lockLogId = databaseServer.getLockLogId() - 1;
    newLock.setLockLogId(lockLogId.toString());
    Collection<Lock> newCurrentLocks = new ArrayList<Lock>();
    Collection<Lock> newPendingLocks = new ArrayList<Lock>();
    newPendingLocks.add(newLock);
    locksToAdd.add(newLock);
    locksToRemove.add(lock);
    resource.setAllowMoreLocks(true);

    Collection<Lock> browseLocksToRemove = new ArrayList<Lock>();

    for (Iterator<Lock> j = pendingLocks.iterator(); j
        .hasNext();) {
      Lock browseLock = j.next();
      if (browseLock instanceof BrowseLock) {
        if (browseLock.getTransaction()
            .getOnlineTransaction()) {
          Lock newROnLock = new ROnLock(browseLock
              .getTransaction(), resource);
          newROnLock.setLockLogId(lockLogId.toString());
          newCurrentLocks.add(newROnLock);
          browseLocksToRemove.add(browseLock);
          browseLock.getTransaction().getLocks().remove(
              browseLock);
          browseLock.getTransaction().getLocks().add(
              newROnLock);
        } else {
          Lock newWIOffLock = new WIOffLock(browseLock
              .getTransaction(), resource);
          newWIOffLock.setLockLogId(lockLogId.toString());
          newPendingLocks.add(newWIOffLock);
          browseLocksToRemove.add(browseLock);
          if (browseLock.getTransaction().getOwner()
              .getConnectionStatus()) {
            browseLock.getTransaction().getLocks().remove(
                browseLock);
```

```
                  browseLock.getTransaction().getLocks().add(
                      newWIOffLock);
                } else {
                  databaseServer
                      .getLostLocksWhenMobileHostIsDisconnected()
                      .add(browseLock);
                  databaseServer
                      .getGainedLocksWhenMobileHostIsDisconnected()
                      .add(newWIOffLock);
                }

              }
            }
          }
          pendingLocks.removeAll(browseLocksToRemove);
          delegateLock(resource, newCurrentLocks, newPendingLocks);
        }
      }
    }
  }
  transaction.getLocks().removeAll(locksToRemove);
  transaction.getLocks().addAll(locksToAdd);
}

/**
 */
private void notifyROnLockOwnersAboutGrantedWOffLock(
    Collection<Lock> rOnLocks) {
  for (Iterator<Lock> i = rOnLocks.iterator(); i.hasNext();) {
    Lock lock = i.next();
    lock.getTransaction().getOwner()
        .notifyAboutGrantedWOffLockOfROnLockedResource(
            (ROnLock) lock);
  }
}

private void notifyWIOffOwnerAboutLostWIOffLock(Lock lostLock) {
  databaseServer.notifyAboutLostWIOffLock(lostLock);
}

private void notifyAboutWOffUpgrade(Lock browseLock) {
  Transaction transaction = browseLock.getTransaction();
  if (transaction.getOnlineTransaction()) {
    transaction.getTransactionManager().notifyAboutWOffUpgrade(
        browseLock);
  } else {
    transaction.getAnchorTransaction().getTransactionManager()
        .notifyAboutWOffUpgrade(browseLock);
  }
```

```
  }

  /**
   * Getter of the property <tt>databaseServer</tt>
   *
   * @return Returns the databaseServer.
   * @uml.property name="databaseServer"
   */
  public DatabaseServer getDatabaseServer() {
    return databaseServer;
  }

  /**
   * Setter of the property <tt>databaseServer</tt>
   *
   * @param databaseServer
   *          The databaseServer to set.
   * @uml.property name="databaseServer"
   */
  public void setDatabaseServer(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
  }

  private ArrayList<String> getLocks(Collection<Lock> locks) {
    ArrayList<String> returnArrayList = new ArrayList<String>();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      String lockString = "";
      Lock lock = i.next();
      lockString = lockString + lock.getResource().getId();
      lockString = lockString + "(" + lock.getTransaction().getId()
          + "," + lock.getLockType() + ")";
      returnArrayList.add(lockString);
    }
    return returnArrayList;
  }
}
```

C.1.3 TransactionManager

```
package dataBase;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

import lock.BrowseLock;
import lock.Lock;
```

```
import operation.AbortOperation;
import operation.CommitOperation;
import operation.Operation;
import operation.ReadOperation;
import operation.StartOperation;
import operation.WriteOperation;
import transaction.AnchorTransaction;
import transaction.Transaction;

public class TransactionManager {

  /**
   * @uml.property name="databaseServer"
   * @uml.associationEnd
    inverse="transactionManager:dataBase.DatabaseServer"
   */
  private DatabaseServer databaseServer;

  /**
   * @uml.property name="anchorTransaction"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
     inverse="transactionManager:transaction.AnchorTransaction"
   */
  private Collection<AnchorTransaction> anchorTransaction;

  /**
   * @uml.property name="transaction"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="transactionManager:transaction.Transaction"
   */
  private Collection<Transaction> transaction;

  public TransactionManager(DatabaseServer databaseServer) {
    this.databaseServer = databaseServer;
    anchorTransaction = new ArrayList<AnchorTransaction>();
    transaction = new ArrayList<Transaction>();
  }

  /**
   */
  public Transaction getTransaction(String transactionId,
      String mobileHostId) {
    for (Iterator<Transaction> i = transaction.iterator(); i
        .hasNext();) {
      Transaction t = i.next();
      if (t.getId().equals(transactionId)
          && t.getOwner().getId().equals(mobileHostId)) {
```

```
      return t;
    }
  }
  return null;
}

/**
 */
public void executeStartOperation(String transactionId,
    String mobileHostId) {
  Transaction t = new Transaction(transactionId, databaseServer
      .getMobileHost(mobileHostId), "ACTIVE", true, this, null,
      null);
  StartOperation startOperation = new StartOperation(t);
  t.getOperation().add(startOperation);
  transaction.add(t);
  databaseServer.initiateTransaction(t);
}

/**
 */
public void executeReadOperation(String transactionId,
    String resourceId, String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = databaseServer.getResource(resourceId);
    ReadOperation readOperation = new ReadOperation(t, r);
    t.getOperation().add(readOperation);
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("RONLOCK")) {
      databaseServer.read(readOperation);
    } else if (lockType.equals("WONLOCK")) {
      databaseServer.read(readOperation);
    } else if (lockType.equals("BROWSELOCK")) {
      System.out.println("Transaction " + transactionId
          + " browses resource " + resourceId);
      databaseServer.read(readOperation);
    } else {
      executeAbortOperation(transactionId, mobileHostId);
    }
  }
}
```

```
/**
 */
public void executeWriteOperation(String transactionId,
    String resourceId, int newValue, String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = databaseServer.getResource(resourceId);
    WriteOperation writeOperation = new WriteOperation(t, r,
        newValue);
    t.getOperation().add(writeOperation);
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("WONLOCK")) {
      databaseServer.write(writeOperation);
    } else {
      executeAbortOperation(transactionId, mobileHostId);
    }
  }
}

/**
 */
public void executeCommitOperation(String transactionId,
    String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  if (t.getStatus().equals("ACTIVE")) {
    CommitOperation commitOperation = new CommitOperation(t);
    t.getOperation().add(commitOperation);
    databaseServer.commit(commitOperation);
  }
}

/**
 */
public void executeAbortOperation(String transactionId,
    String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  if (t.getStatus().equals("ACTIVE")) {
    AbortOperation abortOperation = new AbortOperation(t);
    t.getOperation().add(abortOperation);
    databaseServer.abort(abortOperation);
```

```
  }
}

/**
 */
public void requestROnLock(String transactionId,
    String resourceId, String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  Resource r = databaseServer.getResource(resourceId);
  if (t.getStatus().equals("ACTIVE")) {
    if (databaseServer.requestROnLock(r, t)) {
    } else {
      Collection<Lock> currentLocks = r.getCurrentLocks();
      Collection<Lock> pendingLocks = r.getPendingLocks();
      boolean woffLockOnResource = false;
      for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();)
      {
        Lock lock = i.next();
        if (lock.getLockType().equals("WOFFLOCK")) {
          woffLockOnResource = true;
        }
      }
      for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();)
      {
        Lock lock = i.next();
        if (lock.getLockType().equals("WOFFLOCK")) {
          woffLockOnResource = true;
        }
      }
      if (woffLockOnResource) {
        requestBrowseLock(t.getId(), r.getId(), mobileHostId);
      } else {
        executeAbortOperation(transactionId, mobileHostId);
      }
    }
  }
}

/**
 */
public void requestWOnLock(String transactionId,
    String resourceId, String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  Resource r = databaseServer.getResource(resourceId);
  if (t.getStatus().equals("ACTIVE")) {

    if (databaseServer.requestWOnLock(r, t)) {
    } else {
      executeAbortOperation(transactionId, mobileHostId);
```

```
    }
  }
}

/**
 */
public void requestBrowseLock(String transactionId,
    String resourceId, String mobileHostId) {
  Transaction t = getTransaction(transactionId, mobileHostId);
  if (t.getStatus().equals("ACTIVE")) {
    if (databaseServer.requestBrowseLock(databaseServer
        .getResource(resourceId), t)) {
    } else {
      executeAbortOperation(transactionId, mobileHostId);
    }
  }
}

/**
 */
public void notifyAboutWOffUpgrade(Lock browseLock) {
  if (browseLock.getTransaction().getOnlineTransaction()) {
    for (Iterator<Transaction> i = transaction.iterator(); i
        .hasNext();) {
      Transaction transaction = i.next();
      if (browseLock.getTransaction().equals(transaction)) {
        registerForReadingBrowsedValueAgain(transaction,
            browseLock);
      }
    }
  } else {
    if (browseLock.getTransaction().getOwner()
        .getConnectionStatus()) {
      browseLock.getTransaction().getOwner()
          .notifyAboutWOffUpgrade(browseLock);
    } else {
      for (Iterator<AnchorTransaction> i = anchorTransaction
          .iterator(); i.hasNext();) {
        AnchorTransaction anchorTransaction = i.next();
        if (browseLock.getTransaction().getAnchorTransaction()
            .equals(anchorTransaction)) {
          anchorTransaction.getInconsistentBrowseLocks().add(
              (BrowseLock) browseLock);
        }
      }
    }
  }
}
```

```
/**
 */
public void registerForReadingBrowsedValueAgain(
    Transaction transaction, Lock browseLock) {
  WriteOperation writeOperationWithNewValue = null;
  BrowseLock browseLock2 = (BrowseLock) browseLock;
  Transaction wOnTransaction = browseLock2.getGrantedWOnLock()
      .getTransaction();
  for (Iterator<Operation> i = wOnTransaction.getOperation()
      .iterator(); i.hasNext();) {
    Operation operation = i.next();
    if (operation instanceof WriteOperation) {
      WriteOperation writeOperation = (WriteOperation) operation;
      if (writeOperation.getResource().equals(
          browseLock2.getGrantedWOnLock().getResource())) {
        writeOperationWithNewValue = writeOperation;
      }
    }
  }
  for (Iterator<Operation> i = transaction.getOperation()
      .iterator(); i.hasNext();) {
    Operation operation = i.next();
    if (operation instanceof ReadOperation) {
      ReadOperation readOperation = (ReadOperation) operation;
      if (readOperation.getTransaction().equals(transaction)
          && readOperation.getResource().equals(
              browseLock.getResource())) {
        writeOperationWithNewValue
            .registerForReadingBrowsedValueAgain(readOperation);
      }
    }
  }
}

/**
 */
public void registerAnchorTransaction(
    AnchorTransaction anchorTransaction) {
  this.anchorTransaction.add(anchorTransaction);
}

/**
 */
public AnchorTransaction getAnchorTransaction(String mobileHostId) {
  for (Iterator<AnchorTransaction> i = anchorTransaction
      .iterator(); i.hasNext();) {
    AnchorTransaction t = i.next();
    if (t.getMobileHost().getId().equals(mobileHostId)) {
      return t;
```

```
    }
  }
  return null;
}

/**
 * Getter of the property <tt>databaseServer</tt>
 *
 * @return Returns the databaseServer.
 * @uml.property name="databaseServer"
 */
public DatabaseServer getDatabaseServer() {
  return databaseServer;
}

/**
 * Setter of the property <tt>databaseServer</tt>
 *
 * @param databaseServer
 *          The databaseServer to set.
 * @uml.property name="databaseServer"
 */
public void setDatabaseServer(DatabaseServer databaseServer) {
  this.databaseServer = databaseServer;
}

/**
 * Getter of the property <tt>anchorTransaction</tt>
 *
 * @return Returns the anchorTransaction.
 * @uml.property name="anchorTransaction"
 */
public Collection<AnchorTransaction> getAnchorTransaction() {
  return anchorTransaction;
}

/**
 * Setter of the property <tt>anchorTransaction</tt>
 * @param anchorTransaction  The anchorTransaction to set.
 * @uml.property  name="anchorTransaction"
 */
public void setAnchorTransaction(
    Collection<AnchorTransaction> anchorTransaction) {
  this.anchorTransaction = anchorTransaction;
}

/**
 * @uml.property  name="transaction"
 */
```

```
  public Collection<Transaction> getTransaction() {
    return transaction;
  }

  /**
   * Setter of the property <tt>transaction</tt>
   * @param transaction  The transaction to set.
   * @uml.property  name="transaction"
   */
  public void setTransaction(Collection<Transaction> transaction) {
    this.transaction = transaction;
  }
}
```

C.2 Modulen *mobileHost*

De utvidede klassene i modulen *mobileHost* er:

- `MobileHost`
- `LocalLog`
- `LocalTransactionManager`

C.2.1 `MobileHost`

```
package mobileHost;

import dataBase.MobileSupportStation;
import lock.Lock;
import lock.ROnLock;
import transaction.AnchorTransaction;

public class MobileHost {

  /**
   * @uml.property name="id"
   */
  private String id = "";

  /**
   * @uml.property name="localTime"
   */
  private long localTime;

  /**
   * @uml.property name="lastDisconnectionTime"
   */
  private long lastDisconnectionTime;

  /**
   * @uml.property name="connectionStatus"
   */
  private boolean connectionStatus;

  /**
   * @uml.property name="mobileSupportStation"
   * @uml.associationEnd
    inverse="mobileHost:database.MobileSupportStation"
   */
```

```
private MobileSupportStation mobileSupportStation;

/**
 * @uml.property name="localTransactionManager"
 * @uml.associationEnd
  inverse="mobileHost:mobileHost.LocalTransactionManager"
 */
private LocalTransactionManager localTransactionManager;

/**
 * @uml.property name="localCacheManager"
 * @uml.associationEnd
  inverse="mobileHost:mobileHost.LocalCacheManager"
 */
private LocalCacheManager localCacheManager;

/**
 * @uml.property name="cache"
 * @uml.associationEnd inverse="mobileHost:mobileHost.Cache"
 */
private Cache cache;

/**
 * @uml.property name="localLockLog"
 * @uml.associationEnd inverse="mobileHost:mobileHost.LocalLockLog"
 */
private LocalLockLog localLockLog;

/**
 * @uml.property name="localLog"
 * @uml.associationEnd inverse="mobileHost:mobileHost.LocalLog"
 */
private LocalLog localLog;

private long nanoStartTime;

/**
 * @uml.property name="anchorTransaction"
 * @uml.associationEnd
  inverse="mobileHost:transaction.AnchorTransaction"
 */
private AnchorTransaction anchorTransaction;

public MobileHost(String id, MobileSupportStation mss,
    int cacheSize, long localTime) {
  this.id = id;
  mobileSupportStation = mss;
  this.localTime = localTime;
  nanoStartTime = mobileSupportStation.getDatabaseServer()
```

```
        .getNanoStartTime();
    lastDisconnectionTime = 0;
    connectionStatus = true;
    localTransactionManager = new LocalTransactionManager(this);
    cache = new Cache(this, cacheSize);
    localCacheManager = new LocalCacheManager(this, cache);
    localLockLog = new LocalLockLog(this);
    localLog = new LocalLog(this);
    anchorTransaction = new AnchorTransaction(this,
        mobileSupportStation.getDatabaseServer()
            .getTransactionManager());
}

/**
 */
public void reconnect() {
    connectionStatus = true;
    mobileSupportStation.getDatabaseServer().reconnect(this);
}

/**
 */
public void disconnectPlanned() {
    localTime = mobileSupportStation.getDatabaseServer().getTime();
    lastDisconnectionTime = localTime;
    mobileSupportStation.getDatabaseServer()
        .initiatePlannedDisconnection(this);
    connectionStatus = false;
}

/**
 */
public void disconnectUnplanned() {
    localTime = mobileSupportStation.getDatabaseServer().getTime();
    lastDisconnectionTime = localTime;
    connectionStatus = false;
    try {
      Thread.sleep(new Integer(500).longValue());
    } catch (InterruptedException e) {
      e.printStackTrace();
    }
}

/**
 */
public void startNewOnlineTransaction(String transactionId) {
    mobileSupportStation.getDatabaseServer().getTransactionManager()
        .executeStartOperation(transactionId, id);
}
```

```
/**
 */
public void startNewOfflineTransaction(String transactionId) {
  localTransactionManager.executeStartOperation(transactionId);
}

/**
 */
public void notifyAboutGrantedWOffLockOfROnLockedResource(
    ROnLock lock) {
  lock.setGrantedWOffLock(true);
}

/**
 */
public void notifyAboutLostWIOffLock(Lock lostLock) {
  lostLock.getTransaction().getLocks().remove(lostLock);
  localCacheManager.removeResource(lostLock.getResource(),
      lostLock.getTransaction());
}

/**
 */
public void notifyAboutWOffUpgrade(Lock browseLock) {
  localTransactionManager
      .registerForReadingBrowsedValueAgain(browseLock);
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}
```

```
/**
 * Getter of the property <tt>localTime</tt>
 *
 * @return Returns the localTime.
 * @uml.property name="localTime"
 */
public long getLocalTime() {
  return System.nanoTime() - nanoStartTime;
}

/**
 * Setter of the property <tt>localTime</tt>
 *
 * @param localTime
 *          The localTime to set.
 * @uml.property name="localTime"
 */
public void setLocalTime(long localTime) {
  this.localTime = localTime;
}

/**
 * Getter of the property <tt>connectionStatus</tt>
 *
 * @return Returns the connectionStatus.
 * @uml.property name="connectionStatus"
 */
public boolean getConnectionStatus() {
  return connectionStatus;
}

/**
 * Setter of the property <tt>connectionStatus</tt>
 *
 * @param connectionStatus
 *          The connectionStatus to set.
 * @uml.property name="connectionStatus"
 */
public void setConnectionStatus(boolean connectionStatus) {
  this.connectionStatus = connectionStatus;
}

/**
 * Getter of the property <tt>mobileSupportStation</tt>
 *
 * @return Returns the mobileSupportStation.
 * @uml.property name="mobileSupportStation"
 */
```

```
public MobileSupportStation getMobileSupportStation() {
  return mobileSupportStation;
}

/**
 * Setter of the property <tt>mobileSupportStation</tt>
 *
 * @param mobileSupportStation
 *          The mobileSupportStation to set.
 * @uml.property name="mobileSupportStation"
 */
public void setMobileSupportStation(
    MobileSupportStation mobileSupportStation) {
  this.mobileSupportStation = mobileSupportStation;
}

/**
 * Getter of the property <tt>localTransactionManager</tt>
 *
 * @return Returns the localTransactionManager.
 * @uml.property name="localTransactionManager"
 */
public LocalTransactionManager getLocalTransactionManager() {
  return localTransactionManager;
}

/**
 * Setter of the property <tt>localTransactionManager</tt>
 *
 * @param localTransactionManager
 *          The localTransactionManager to set.
 * @uml.property name="localTransactionManager"
 */
public void setLocalTransactionManager(
    LocalTransactionManager localTransactionManager) {
  this.localTransactionManager = localTransactionManager;
}

/**
 * Getter of the property <tt>localCacheManager</tt>
 *
 * @return Returns the localCacheManager.
 * @uml.property name="localCacheManager"
 */
public LocalCacheManager getLocalCacheManager() {
  return localCacheManager;
}

/**
```

```
 * Setter of the property <tt>localCacheManager</tt>
 *
 * @param localCacheManager
 *          The localCacheManager to set.
 * @uml.property name="localCacheManager"
 */
public void setLocalCacheManager(
    LocalCacheManager localCacheManager) {
  this.localCacheManager = localCacheManager;
}

/**
 * Getter of the property <tt>cache</tt>
 *
 * @return Returns the cache.
 * @uml.property name="cache"
 */
public Cache getCache() {
  return cache;
}

/**
 * Setter of the property <tt>cache</tt>
 *
 * @param cache
 *          The cache to set.
 * @uml.property name="cache"
 */
public void setCache(Cache cache) {
  this.cache = cache;
}

/**
 * Getter of the property <tt>localLockLog</tt>
 *
 * @return Returns the localLockLog.
 * @uml.property name="localLockLog"
 */
public LocalLockLog getLocalLockLog() {
  return localLockLog;
}

/**
 * Setter of the property <tt>localLockLog</tt>
 *
 * @param localLockLog
 *          The localLockLog to set.
 * @uml.property name="localLockLog"
 */
```

```
public void setLocalLockLog(LocalLockLog localLockLog) {
  this.localLockLog = localLockLog;
}

/**
 * Getter of the property <tt>localLog</tt>
 *
 * @return Returns the localLog.
 * @uml.property name="localLog"
 */
public LocalLog getLocalLog() {
  return localLog;
}

/**
 * Setter of the property <tt>localLog</tt>
 *
 * @param localLog
 *          The localLog to set.
 * @uml.property name="localLog"
 */
public void setLocalLog(LocalLog localLog) {
  this.localLog = localLog;
}

/**
 * Getter of the property <tt>lastDisconnectionTime</tt>
 *
 * @return Returns the lastDisconnectionTime.
 * @uml.property name="lastDisconnectionTime"
 */
public long getLastDisconnectionTime() {
  return lastDisconnectionTime;
}

/**
 * Setter of the property <tt>lastDisconnectionTime</tt>
 *
 * @param lastDisconnectionTime
 *          The lastDisconnectionTime to set.
 * @uml.property name="lastDisconnectionTime"
 */
public void setLastDisconnectionTime(long lastDisconnectionTime) {
  this.lastDisconnectionTime = lastDisconnectionTime;
}

/**
 * Getter of the property <tt>anchorTransaction</tt>
 *
```

```
   * @return Returns the anchorTransaction.
   * @uml.property name="anchorTransaction"
   */
  public AnchorTransaction getAnchorTransaction() {
    return anchorTransaction;
  }

  /**
   * Setter of the property <tt>anchorTransaction</tt>
   * @param anchorTransaction  The anchorTransaction to set.
   * @uml.property  name="anchorTransaction"
   */
  public void setAnchorTransaction(
      AnchorTransaction anchorTransaction) {
    this.anchorTransaction = anchorTransaction;
  }
}
```

C.2.2 LocalLog

```
package mobileHost;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class LocalLog {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd inverse="localLog:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="localLogRecord"
   * @uml.associationEnd multiplicity="(0 -1)"
   *                     inverse="localLog:mobileHost.LocalLogRecord"
   */
  private Collection<LocalLogRecord> localLogRecord;

  /**
   * @uml.property name="localLogId"
   */
  private int localLogId;

  public LocalLog(MobileHost mobileHost) {
```

```
  this.mobileHost = mobileHost;
  localLogRecord = new ArrayList<LocalLogRecord>();
  localLogId = 0;
}

/**
 */
public void addLocalLogRecord(String transactionId, String action,
    String resourceId, String value, String newValue) {
  localLogRecord.add(new LocalLogRecord(this, new String(""
      + localLogId), new String("" + mobileHost.getLocalTime()),
      transactionId, action, resourceId, value, newValue));
  localLogId = localLogId + 1;
}

/**
 */
public Collection<LocalLogRecord>
  getLocalLogRecordsFromLastDisconnection(
    long lastDisconnectionTime) {
  Collection<LocalLogRecord> returnList = new
    ArrayList<LocalLogRecord>();
  for (Iterator<LocalLogRecord> i = localLogRecord.iterator(); i
      .hasNext();) {
    LocalLogRecord localLogRec = i.next();
    int time = new Integer(localLogRec.getLocalTime());
    if (time > lastDisconnectionTime
        && !localLogRec.getAction().equals(
            "STARTEXPORTTRANSACTION")
        && !localLogRec.getAction().equals(
            "STARTIMPORTTRANSACTION")) {
      returnList.add(localLogRec);
    }
  }
  return returnList;
}

/**
 */
public void printLocalLog() {
  System.out.println("");
  System.out.println("Mobilehost " + mobileHost.getId()
      + "s local log:");
  System.out.println("");
  System.out
      .println("Localtime TransactionId action ResourceId  value
        newValue");
  System.out.println("");
  for (Iterator<LocalLogRecord> i = localLogRecord.iterator(); i
```

```
      .hasNext();) {
    LocalLogRecord localLogRecord = i.next();
    System.out.println(localLogRecord.getLocalTime() + " "
        + localLogRecord.getTransactionId() + " "
        + localLogRecord.getAction() + " "
        + localLogRecord.getResourceId() + " "
        + localLogRecord.getValue() + " "
        + localLogRecord.getNewValue());
  }
}

/**
 * Getter of the property <tt>mobileHost</tt>
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
  return mobileHost;
}

/**
 * Setter of the property <tt>mobileHost</tt>
 *
 * @param mobileHost
 *          The mobileHost to set.
 * @uml.property name="mobileHost"
 */
public void setMobileHost(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
}

/**
 * Getter of the property <tt>localLogRecord</tt>
 *
 * @return Returns the localLogRecord.
 * @uml.property name="localLogRecord"
 */
public Collection<LocalLogRecord> getLocalLogRecord() {
  return localLogRecord;
}

/**
 * Setter of the property <tt>localLogRecord</tt>
 *
 * @param localLogRecord
 *          The localLogRecord to set.
 * @uml.property name="localLogRecord"
 */
```

```
  public void setLocalLogRecord(
      Collection<LocalLogRecord> localLogRecord) {
    this.localLogRecord = localLogRecord;
  }

  /**
   * Getter of the property <tt>logId</tt>
   *
   * @return Returns the logId.
   * @uml.property name="localLogId"
   */
  public int getLocalLogId() {
    return localLogId;
  }

  /**
   * Setter of the property <tt>logId</tt>
   * @param logId  The logId to set.
   * @uml.property  name="localLogId"
   */
  public void setLocalLogId(int localLogId) {
    this.localLogId = localLogId;
  }
}
```

C.2.3 LocalTransactionManager

```
package mobileHost;

import dataBase.Resource;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import lock.BrowseLock;
import lock.Lock;
import mobileAffiliation.MobileAffiliation;
import operation.AbortOperation;
import operation.CommitOperation;
import operation.Operation;
import operation.ReadOperation;
import operation.StartOperation;
import operation.WriteOperation;
import transaction.ExportTransaction;
import transaction.ImportTransaction;
import transaction.Transaction;

public class LocalTransactionManager {
```

```
/**
 * @uml.property name="mobileHost"
 * @uml.associationEnd
   inverse="localTransactionManager:mobileHost.MobileHost"
 */
private MobileHost mobileHost;

/**
 * @uml.property name="transaction"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
   inverse="localTransactionManager:transaction.Transaction"
 */
private Collection<Transaction> transaction;

public LocalTransactionManager(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
  transaction = new ArrayList<Transaction>();
}

/**
 */
public Transaction getTransaction(String transactionId) {
  for (Iterator<Transaction> i = transaction.iterator(); i
      .hasNext();) {
    Transaction t = i.next();
    if (t.getId().equals(transactionId)) {
      return t;
    }
  }
  return null;
}

/**
 */
public void executeStartOperation(String transactionId) {
  Transaction t = new Transaction(transactionId, getMobileHost(),
      "ACTIVE", false, null, this, mobileHost
          .getAnchorTransaction());
  StartOperation startOperation = new StartOperation(t);
  t.getOperation().add(startOperation);
  transaction.add(t);
  if (mobileHost.getConnectionStatus() == true) {
    mobileHost.getAnchorTransaction().initiateTransaction(t);
  } else {
    startTransactionOffline(t);
  }
}
```

```
/**
 */
public void executeReadOperation(String transactionId,
    String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    ReadOperation readOperation = new ReadOperation(t, r);
    t.getOperation().add(readOperation);
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("WOFFLOCK")) {
      if (mobileHost.getCache().isResourceCached(r)) {
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else if (lockType.equals("WIOFFLOCK")) {
      if (mobileHost.getCache().isResourceCached(r)) {
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else if (lockType.equals("BROWSELOCK")) {
      if (mobileHost.getCache().isResourceCached(r)) {
        System.out.println("Transaction " + transactionId
            + " browses resource " + resourceId + " offline");
        readOffline(readOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void executeWriteOperation(String transactionId,
    String resourceId, int newValue) {
```

```
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    WriteOperation writeOperation = new WriteOperation(t, r,
        newValue);
    t.getOperation().add(writeOperation);
    String lockType = "NONE";
    Collection<Lock> locks = t.getLocks();
    for (Iterator<Lock> i = locks.iterator(); i.hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().getId().equals(resourceId)) {
        lockType = lock.getLockType();
      }
    }
    if (lockType.equals("WOFFLOCK")) {
      if (mobileHost.getCache().isResourceCached(r)) {
        writeOffline(writeOperation);
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}

/**
 */
public void executeCommitOperation(String transactionId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    CommitOperation commitOperation = new CommitOperation(t);
    t.getOperation().add(commitOperation);
    if (mobileHost.getConnectionStatus() == true) {
      mobileHost.getAnchorTransaction().commitTransaction(t);
    } else {
      commitOffline(commitOperation);
    }
  }
}

/**
 */
public void executeAbortOperation(String transactionId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    AbortOperation abortOperation = new AbortOperation(t);
    t.getOperation().add(abortOperation);
```

```
    if (mobileHost.getConnectionStatus() == true) {
      mobileHost.getAnchorTransaction().abortTransaction(t);
    } else {
      abortOffline(abortOperation);
    }
  }
}

/**
 */
public void requestWIOffLock(String transactionId,
    String resourceId) {
  Transaction t = getTransaction(transactionId);
  Resource r = mobileHost.getMobileSupportStation()
      .getDatabaseServer().getResource(resourceId);
  if (mobileHost.getConnectionStatus()) {
    if (mobileHost.getAnchorTransaction().requestWIOffLock(t, r)) {
    } else {
      Collection<Lock> currentLocks = r.getCurrentLocks();
      Collection<Lock> pendingLocks = r.getPendingLocks();
      boolean woffLockOnResource = false;
      for (Iterator<Lock> i = currentLocks.iterator(); i.hasNext();)
      {
        Lock lock = i.next();
        if (lock.getLockType().equals("WOFFLOCK")) {
          woffLockOnResource = true;
        }
      }
      for (Iterator<Lock> i = pendingLocks.iterator(); i.hasNext();)
      {
        Lock lock = i.next();
        if (lock.getLockType().equals("WOFFLOCK")) {
          woffLockOnResource = true;
        }
      }
      if (woffLockOnResource) {
        requestBrowseLock(t.getId(), r.getId());
      } else {
        executeAbortOperation(transactionId);
      }
    }
  } else {
    executeAbortOperation(transactionId);
  }
}

/**
 */
public void requestWOffLock(String transactionId, String resourceId) {
```

```
  Transaction t = getTransaction(transactionId);
  Resource r = mobileHost.getMobileSupportStation()
      .getDatabaseServer().getResource(resourceId);
  if (mobileHost.getConnectionStatus()) {
    if (mobileHost.getAnchorTransaction().requestWOffLock(t, r)) {
    } else {
      executeAbortOperation(transactionId);
    }
  } else {
    executeAbortOperation(transactionId);
  }
}

/**
 */
public void requestBrowseLock(String transactionId,
    String resourceId) {
  Transaction t = getTransaction(transactionId);
  Resource r = mobileHost.getMobileSupportStation()
      .getDatabaseServer().getResource(resourceId);
  if (mobileHost.getConnectionStatus()) {
    if (mobileHost.getAnchorTransaction().requestBrowseLock(t, r)) {

    } else {
      executeAbortOperation(transactionId);
    }
  } else {
    executeAbortOperation(transactionId);
  }
}

/**
 */
public void requestWOnLock(String transactionId, String resourceId) {
  Transaction t = getTransaction(transactionId);
  if (t.getStatus().equals("ACTIVE")) {
    Resource r = mobileHost.getMobileSupportStation()
        .getDatabaseServer().getResource(resourceId);
    if (mobileHost.getConnectionStatus() == true) {
      if (mobileHost.getAnchorTransaction().requestWOnLock(t, r)) {
      } else {
        executeAbortOperation(transactionId);
      }
    } else {
      executeAbortOperation(transactionId);
    }
  }
}
```

```
/**
 */
public void startTransactionOffline(Transaction transaction) {
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "STARTOFFLINE", "", "", "");
}

/**
 */
public int readOffline(ReadOperation readOperation) {
  Transaction transaction = readOperation.getTransaction();
  Resource resource = readOperation.getResource();
  int readValue;
  if (transaction.isResourceWritten(resource)) {
    readValue = transaction.getNewestWrittenValue(resource,
        readOperation);
  } else {
    readValue = resource.getValue();
  }
  readOperation.setReadValue(readValue);
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "READOFFLINE", resource.getId(), new String("" + readValue),
      "");
  return readValue;
}

/**
 */
public void writeOffline(WriteOperation writeOperation) {
  Transaction transaction = writeOperation.getTransaction();
  Resource resource = writeOperation.getResource();
  int newValue = writeOperation.getNewValue();
  transaction.getWrittenResources().add(resource);
  mobileHost.getLocalLog().addLocalLogRecord(transaction.getId(),
      "WRITEOFFLINE", resource.getId(),
      new String("" + resource.getValue()),
      new String("" + newValue));
}

/**
 */
public void commitOffline(CommitOperation commitOperation) {
  Transaction t = commitOperation.getTransaction();
  t.setStatus("COMMITEDOFFLINE");
  mobileHost.getLocalLog().addLocalLogRecord(t.getId(),
      "COMMITOFFLINE", "", "", "");
}

/**
```

```
 */
public void abortOffline(AbortOperation abortOperation) {
  Transaction t = abortOperation.getTransaction();
  t.setStatus("ABORTEDOFFLINE");
  mobileHost.getLocalLog().addLocalLogRecord(t.getId(),
      "ABORTOFFLINE", "", "", "");
}

/**
 */
public void registerForReadingBrowsedValueAgain(Lock browseLock) {
  WriteOperation writeOperationWithNewValue = null;
  BrowseLock browseLock2 = (BrowseLock) browseLock;
  Transaction wOnTransaction = browseLock2.getGrantedWOnLock()
      .getTransaction();
  for (Iterator<Operation> i = wOnTransaction.getOperation()
      .iterator(); i.hasNext();) {
    Operation operation = i.next();
    if (operation instanceof WriteOperation) {
      WriteOperation writeOperation = (WriteOperation) operation;
      if (writeOperation.getResource().equals(
          browseLock2.getGrantedWOnLock().getResource())) {
        writeOperationWithNewValue = writeOperation;
      }
    }
  }
  Transaction transaction = browseLock.getTransaction();
  for (Iterator<Operation> i = transaction.getOperation()
      .iterator(); i.hasNext();) {
    Operation operation = i.next();
    if (operation instanceof ReadOperation) {
      ReadOperation readOperation = (ReadOperation) operation;
      if (readOperation.getTransaction().equals(transaction)
          && readOperation.getResource().equals(
              browseLock.getResource())) {

        writeOperationWithNewValue
            .registerForReadingBrowsedValueAgain(readOperation);
      }
    }
  }
}

/**
 */
public void initiatieMobileAffiliation(String transactionId,
    String id) {
  Transaction transaction = getTransaction(transactionId);
  MobileAffiliation mobileAffiliation = new MobileAffiliation(
```

```
      transaction, id);
  transaction.getMobileAffiliation().add(mobileAffiliation);
}

/**
 */
public void registerToMobileAffiliation(String transactionId,
    Transaction initiator, String id) {
  Transaction transaction = getTransaction(transactionId);
  MobileAffiliation mobileAffiliation = initiator
      .getMobileAffiliation(id);
  mobileAffiliation.register(transaction);
  transaction.getMobileAffiliation().add(mobileAffiliation);
}

/**
 */
public void unRegisterFromMobileAffiliation(String transactionId,
    Transaction initiator, String id) {
  Transaction transaction = getTransaction(transactionId);
  MobileAffiliation mobileAffiliation = initiator
      .getMobileAffiliation(id);
  mobileAffiliation.unRegister(transaction);
  transaction.getMobileAffiliation().remove(mobileAffiliation);
}

/**
 */
public void initiateImportTransaction(String id,
    Transaction transaction, MobileAffiliation mobileAffiliation,
    boolean shareToRead, Resource resource) {
  ImportTransaction importTransaction = new ImportTransaction(id,
      transaction, mobileAffiliation, shareToRead, resource);
  transaction.getImportTransaction().add(importTransaction);
  importTransaction.importSharedData();
  mobileHost.getLocalLog().addLocalLogRecord(
      importTransaction.getId(), "STARTIMPORTTRANSACTION", "", "",
      "");
}

/**
 */
public void initiateExportTransaction(String id,
    Transaction transaction, MobileAffiliation mobileAffiliation,
    boolean shareToRead, Resource resource, int newValue) {
  ExportTransaction exportTransaction = new ExportTransaction(id,
      transaction, mobileAffiliation, shareToRead, resource,
      newValue);
  transaction.getExportTransaction().add(exportTransaction);
```

```
    exportTransaction.export();
    mobileHost.getLocalLog().addLocalLogRecord(
        exportTransaction.getId(), "STARTEXPORTTRANSACTION", "", "",
        "");
  }

  /**
   */
  public void shareToRead(String exportTransactionId,
      String transactionId, String resourceId,
      MobileHost initiatorMobileHost, String initiatorTransactionId,
      String mAId) {
    Transaction transaction = getTransaction(transactionId);
    Resource resource = mobileHost.getLocalCacheManager()
        .getResource(resourceId);
    int newValue = transaction.getNewestWrittenValue(resource);
    MobileAffiliation mobileAffiliation = initiatorMobileHost
        .getLocalTransactionManager().getTransaction(
            initiatorTransactionId).getMobileAffiliation(mAId);

    initiateExportTransaction(exportTransactionId, transaction,
        mobileAffiliation, true, resource, newValue);

    System.out.println("Transaction " + transaction.getId()
        + " shareToRead resource " + resource.getId() + "s value "
        + newValue + " in MA " + mobileAffiliation.getId());
  }

  /**
   */
  public void shareToWrite(String exportTransactionId,
      String transactionId, String resourceId,
      MobileHost initiatorMobileHost, String initiatorTransactionId,
      String mAId) {
    Transaction transaction = getTransaction(transactionId);
    Resource resource = mobileHost.getLocalCacheManager()
        .getResource(resourceId);
    int newValue = transaction.getNewestWrittenValue(resource);
    MobileAffiliation mobileAffiliation = initiatorMobileHost
        .getLocalTransactionManager().getTransaction(
            initiatorTransactionId).getMobileAffiliation(mAId);

    initiateExportTransaction(exportTransactionId, transaction,
        mobileAffiliation, false, resource, newValue);
    System.out.println("Transaction " + transaction.getId()
        + " shareToWrite resource " + resource.getId() + "s value "
        + newValue + " in MA " + mobileAffiliation.getId());
  }
```

```
/**
 */
public void importShareToReadSharedData(
    String importTransactionId, String transactionId,
    String resourceId, MobileHost initiatorMobileHost,
    String initiatorTransactionId, String mAId) {
  Transaction transaction = getTransaction(transactionId);
  MobileAffiliation mobileAffiliation = initiatorMobileHost
      .getLocalTransactionManager().getTransaction(
          initiatorTransactionId).getMobileAffiliation(mAId);
  Resource resource = mobileAffiliation
      .getExportImportRepository().getSharedData(resourceId)
      .getResource();

  initiateImportTransaction(importTransactionId, transaction,
      mobileAffiliation, true, resource);
  System.out.println("Transaction " + transaction.getId()
      + " import shareToRead resource " + resource.getId()
      + " in MA " + mobileAffiliation.getId());
}

/**
 */
public void importShareToWriteSharedData(
    String importTransactionId, String transactionId,
    String resourceId, MobileHost initiatorMobileHost,
    String initiatorTransactionId, String mAId) {
  Transaction transaction = getTransaction(transactionId);
  MobileAffiliation mobileAffiliation = initiatorMobileHost
      .getLocalTransactionManager().getTransaction(
          initiatorTransactionId).getMobileAffiliation(mAId);
  Resource resource = mobileAffiliation
      .getExportImportRepository().getSharedData(resourceId)
      .getResource();

  initiateImportTransaction(importTransactionId, transaction,
      mobileAffiliation, false, resource);
  System.out.println("Transaction " + transaction.getId()
      + " import shareToWrite resource " + resource.getId()
      + " in MA " + mobileAffiliation.getId());
}

/**
 * Getter of the property <tt>mobileHost</tt>
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
```

```
    return mobileHost;
  }

  /**
   * Setter of the property <tt>mobileHost</tt>
   *
   * @param mobileHost
   *          The mobileHost to set.
   * @uml.property name="mobileHost"
   */
  public void setMobileHost(MobileHost mobileHost) {
    this.mobileHost = mobileHost;
  }

  /**
   * Getter of the property <tt>transaction</tt>
   * @return  Returns the transaction.
   * @uml.property  name="transaction"
   */
  public Collection<Transaction> getTransaction() {
    return transaction;
  }

  /**
   * Setter of the property <tt>transaction</tt>
   * @param transaction  The transaction to set.
   * @uml.property  name="transaction"
   */
  public void setTransaction(Collection<Transaction> transaction) {
    this.transaction = transaction;
  }
}
```

C.3 Modulen *transaction*

De utvidede klassene i modulen *transaction* er:

- `Transaction`

Nye klasser i modulen *transaction* er:

- `AnchorTransaction`
- `ExportTransaction`
- `ImportTransaction`

C.3.1 `Transaction`

```
package transaction;

import dataBase.Resource;
import dataBase.TransactionManager;
import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;
import lock.Lock;
import mobileHost.LocalTransactionManager;
import mobileHost.MobileHost;
import operation.Operation;
import operation.ReadOperation;
import operation.WriteOperation;
import mobileAffiliation.MobileAffiliation;

public class Transaction {

  /**
   * @uml.property name="owner"
   */
  private MobileHost owner;

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="status"
```

```
 */
private String status = "";

/**
 * @uml.property name="onlineTransaction"
 */
private boolean onlineTransaction;

/**
 * @uml.property name="operation"
 * @uml.associationEnd multiplicity="(0 -1)"
 *                     inverse="transaction:operation.Operation"
 */
private Collection<Operation> operation;

/**
 * @uml.property name="transactionManager"
 * @uml.associationEnd
   inverse="transaction:database.TransactionManager"
 */
private TransactionManager transactionManager;

/**
 * @uml.property name="localTransactionManager"
 * @uml.associationEnd
   inverse="transaction:mobileHost.LocalTransactionManager"
 */
private LocalTransactionManager localTransactionManager;

/**
 * @uml.property name="anchorTransaction"
 * @uml.associationEnd
   inverse="transaction:transaction.AnchorTransaction"
 */
private AnchorTransaction anchorTransaction;

/**
 * @uml.property name="locks"
 */
private Collection<Lock> locks;

/**
 * @uml.property name="mobileAffiliation"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
   inverse="transaction:mobileAffiliation.MobileAffiliation"
 */
private Collection<MobileAffiliation> mobileAffiliation;
```

```
/**
 * @uml.property name="importTransaction"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
   inverse="transaction:transaction.ImportTransaction"
 */
private Collection<ImportTransaction> importTransaction;

/**
 * @uml.property name="exportTransaction"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
   inverse="transaction:transaction.ExportTransaction"
 */
private Collection<ExportTransaction> exportTransaction;

private Collection<Resource> writtenResources;

/**
 * @uml.property
  name="forcedCommitOnAbortDependentExportTransaction"
 */
private Collection<ExportTransaction>
  forcedCommitOnAbortDependentExportTransaction;

public Transaction(String id, MobileHost owner, String status,
    boolean onlineTransaction,
    TransactionManager transactionManager,
    LocalTransactionManager localTransactionManager,
    AnchorTransaction anchorTransaction) {
  this.id = id;
  this.owner = owner;
  this.status = status;
  this.onlineTransaction = onlineTransaction;
  this.localTransactionManager = localTransactionManager;
  this.transactionManager = transactionManager;
  this.anchorTransaction = anchorTransaction;
  locks = new ArrayList<Lock>();
  operation = new ArrayList<Operation>();
  writtenResources = new ArrayList<Resource>();
  mobileAffiliation = new ArrayList<MobileAffiliation>();
  importTransaction = new ArrayList<ImportTransaction>();
  exportTransaction = new ArrayList<ExportTransaction>();
  forcedCommitOnAbortDependentExportTransaction = new
    ArrayList<ExportTransaction>();
}

public boolean isResourceWritten(Resource resource) {
  for (Iterator<Resource> i = writtenResources.iterator(); i
```

```
      .hasNext();) {
    Resource r = i.next();
    if (r.getId().equals(resource.getId())) {
      return true;
    }
  }
  return false;
}

public int getNewestWrittenValue(Resource resource,
    ReadOperation ro) {
  int returnValue = 0;
  for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      if (wo.getResource().getId().equals(resource.getId())) {
        returnValue = wo.getNewValue();
      }
    }
    if (op.equals(ro)) {
      break;
    }
  }
  return returnValue;
}

public int getNewestWrittenValue(Resource resource) {
  int returnValue = 0;
  for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      if (wo.getResource().getId().equals(resource.getId())) {
        returnValue = wo.getNewValue();
      }
    }
  }
  return returnValue;
}

public WriteOperation getWriteOperation(Resource resource) {
  for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      if (wo.getResource().getId().equals(resource.getId())) {
        return wo;
      }
```

```
    }
  }
  return null;
}

public WriteOperation getNewestWriteOperation(Resource resource) {
  WriteOperation returnOperation = null;
  for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
    Operation op = i.next();
    if (op.getOperationType().equals("WRITEOPERATION")) {
      WriteOperation wo = (WriteOperation) op;
      if (wo.getResource().getId().equals(resource.getId())) {
        returnOperation = wo;
      }
    }
  }
  return returnOperation;
}

/**
 */
public Operation getOperation(String type) {
  if (type.equals("COMMITOPERATION")) {
    for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
      Operation op = i.next();
      if (op.getOperationType().equals("COMMITOPERATION")) {
        return op;
      }
    }
  }
  if (type.equals("ABORTOPERATION")) {
    for (Iterator<Operation> i = operation.iterator(); i.hasNext();) {
      Operation op = i.next();
      if (op.getOperationType().equals("ABORTOPERATION")) {
        return op;
      }
    }
  }
  return null;
}

/**
 */
public MobileAffiliation getMobileAffiliation(String id) {
  for (Iterator<MobileAffiliation> i = mobileAffiliation
      .iterator(); i.hasNext();) {
    MobileAffiliation ma = i.next();
    if (ma.getId().equals(id) && ma.getInitiator().equals(this)) {
      return ma;
```

```
    }
  }
  return null;
}

/**
 * Getter of the property <tt>owner</tt>
 *
 * @return Returns the owner.
 * @uml.property name="owner"
 */
public MobileHost getOwner() {
  return owner;
}

/**
 * Setter of the property <tt>owner</tt>
 *
 * @param owner
 *          The owner to set.
 * @uml.property name="owner"
 */
public void setOwner(MobileHost owner) {
  this.owner = owner;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>status</tt>
```

```
 *
 * @return Returns the status.
 * @uml.property name="status"
 */
public String getStatus() {
  return status;
}

/**
 * Setter of the property <tt>status</tt>
 *
 * @param status
 *          The status to set.
 * @uml.property name="status"
 */
public void setStatus(String status) {
  this.status = status;
}

/**
 * Getter of the property <tt>locks</tt>
 *
 * @return Returns the locks.
 * @uml.property name="locks"
 */
public Collection<Lock> getLocks() {
  return locks;
}

/**
 * Setter of the property <tt>locks</tt>
 *
 * @param locks
 *          The locks to set.
 * @uml.property name="locks"
 */
public void setLocks(Collection<Lock> locks) {
  this.locks = locks;
}

public Collection<Resource> getWrittenResources() {
  return writtenResources;
}

/**
 * Getter of the property <tt>transactionManager</tt>
 *
 * @return Returns the transactionManager.
 * @uml.property name="transactionManager"
```

```
 */
public TransactionManager getTransactionManager() {
  return transactionManager;
}

/**
 * Setter of the property <tt>transactionManager</tt>
 *
 * @param transactionManager
 *          The transactionManager to set.
 * @uml.property name="transactionManager"
 */
public void setTransactionManager(
    TransactionManager transactionManager) {
  this.transactionManager = transactionManager;
}

/**
 * Getter of the property <tt>localTransactionManager</tt>
 *
 * @return Returns the localTransactionManager.
 * @uml.property name="localTransactionManager"
 */
public LocalTransactionManager getLocalTransactionManager() {
  return localTransactionManager;
}

/**
 * Setter of the property <tt>localTransactionManager</tt>
 *
 * @param localTransactionManager
 *          The localTransactionManager to set.
 * @uml.property name="localTransactionManager"
 */
public void setLocalTransactionManager(
    LocalTransactionManager localTransactionManager) {
  this.localTransactionManager = localTransactionManager;
}

/**
 * Getter of the property <tt>operation</tt>
 *
 * @return Returns the operation.
 * @uml.property name="operation"
 */
public Collection<Operation> getOperation() {
  return operation;
}
```

```
/**
 * Setter of the property <tt>operation</tt>
 *
 * @param operation
 *          The operation to set.
 * @uml.property name="operation"
 */
public void setOperation(Collection<Operation> operation) {
  this.operation = operation;
}

/**
 * Getter of the property <tt>onlineTransaction</tt>
 *
 * @return Returns the onlineTransaction.
 * @uml.property name="onlineTransaction"
 */
public boolean getOnlineTransaction() {
  return onlineTransaction;
}

/**
 * Setter of the property <tt>onlineTransaction</tt>
 *
 * @param onlineTransaction
 *          The onlineTransaction to set.
 * @uml.property name="onlineTransaction"
 */
public void setOnlineTransaction(boolean onlineTransaction) {
  this.onlineTransaction = onlineTransaction;
}

/**
 * Getter of the property <tt>anchorTransaction</tt>
 *
 * @return Returns the anchorTransaction.
 * @uml.property name="anchorTransaction"
 */
public AnchorTransaction getAnchorTransaction() {
  return anchorTransaction;
}

/**
 * Setter of the property <tt>anchorTransaction</tt>
 *
 * @param anchorTransaction
 *          The anchorTransaction to set.
 * @uml.property name="anchorTransaction"
 */
```

```
public void setAnchorTransaction(
    AnchorTransaction anchorTransaction) {
  this.anchorTransaction = anchorTransaction;
}

/**
 * Getter of the property <tt>mobileAffiliation</tt>
 *
 * @return Returns the mobileAffiliation.
 * @uml.property name="mobileAffiliation"
 */
public Collection<MobileAffiliation> getMobileAffiliation() {
  return mobileAffiliation;
}

/**
 * Setter of the property <tt>mobileAffiliation</tt>
 *
 * @param mobileAffiliation
 *          The mobileAffiliation to set.
 * @uml.property name="mobileAffiliation"
 */
public void setMobileAffiliation(
    Collection<MobileAffiliation> mobileAffiliation) {
  this.mobileAffiliation = mobileAffiliation;
}

/**
 * Getter of the property <tt>exportTransaction</tt>
 *
 * @return Returns the exportTransaction.
 * @uml.property name="exportTransaction"
 */
public Collection<ExportTransaction> getExportTransaction() {
  return exportTransaction;
}

/**
 * Setter of the property <tt>exportTransaction</tt>
 *
 * @param exportTransaction
 *          The exportTransaction to set.
 * @uml.property name="exportTransaction"
 */
public void setExportTransaction(
    Collection<ExportTransaction> exportTransaction) {
  this.exportTransaction = exportTransaction;
}
```

```
/**
 * Getter of the property <tt>importTransaction</tt>
 *
 * @return Returns the importTransaction.
 * @uml.property name="importTransaction"
 */
public Collection<ImportTransaction> getImportTransaction() {
  return importTransaction;
}

/**
 * Setter of the property <tt>importTransaction</tt>
 *
 * @param importTransaction
 *          The importTransaction to set.
 * @uml.property name="importTransaction"
 */
public void setImportTransaction(
    Collection<ImportTransaction> importTransaction) {
  this.importTransaction = importTransaction;
}

/**
 * Getter of the property
 * <tt>forcedCommitOnAbortDependentExportTransaction</tt>
 *
 * @return Returns the
   forcedCommitOnAbortDependentExportTransaction.
 * @uml.property
   name="forcedCommitOnAbortDependentExportTransaction"
 */
public Collection<ExportTransaction>
  getForcedCommitOnAbortDependentExportTransaction() {
    return forcedCommitOnAbortDependentExportTransaction;
}

/**
 * Setter of the property
 * <tt>forcedCommitOnAbortDependentExportTransaction</tt>
 *
 * @param forcedCommitOnAbortDependentExportTransaction
 *          The forcedCommitOnAbortDependentExportTransaction to set.
 * @uml.property name="forcedCommitOnAbortDependentExportTransaction"
 */
public void setForcedCommitOnAbortDependentExportTransaction(
    Collection<ExportTransaction>
      forcedCommitOnAbortDependentExportTransaction) {
  this.forcedCommitOnAbortDependentExportTransaction =
    forcedCommitOnAbortDependentExportTransaction;
```

```
  }
}
```

C.3.2 AnchorTransaction

```
package transaction;

import dataBase.Resource;
import dataBase.TransactionManager;
import java.util.ArrayList;
import java.util.Collection;
import lock.BrowseLock;
import mobileHost.MobileHost;
import operation.AbortOperation;
import operation.CommitOperation;

public class AnchorTransaction {

  /**
   * @uml.property name="mobileHost"
   * @uml.associationEnd
    inverse="anchorTransaction:mobileHost.MobileHost"
   */
  private MobileHost mobileHost;

  /**
   * @uml.property name="transactionManager"
   * @uml.associationEnd
    inverse="anchorTransaction:dataBase.TransactionManager"
   */
  private TransactionManager transactionManager;

  /**
   * @uml.property name="transaction"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="anchorTransaction:transaction.Transaction"
   */
  private Collection<Transaction> transaction;

  /**
   * @uml.property name="inconsistentBrowseLocks"
   */
  private Collection<BrowseLock> inconsistentBrowseLocks;

  public AnchorTransaction(MobileHost mobileHost,
      TransactionManager transactionManager) {
```

```
  this.mobileHost = mobileHost;
  this.transactionManager = transactionManager;
  transactionManager.registerAnchorTransaction(this);
  transaction = new ArrayList<Transaction>();
  inconsistentBrowseLocks = new ArrayList<BrowseLock>();
}

/**
 */
public void initiateTransaction(Transaction transaction) {
  this.transaction.add(transaction);
  transactionManager.getDatabaseServer().initiateTransaction(
      transaction);
}

/**
 */
public boolean requestWIOffLock(Transaction transaction,
    Resource resource) {
  if (transaction.getStatus().equals("ACTIVE")) {
    if (transactionManager.getDatabaseServer().requestWIOffLock(
        resource, transaction)) {
      long timeWhenRequestWasGranted = transactionManager
          .getDatabaseServer().getLockLog()
          .getTimeWhenLastLockWasGranted();
      mobileHost.getLocalLockLog().addLocalLockLogRecord(
          timeWhenRequestWasGranted, transaction.getId(),
          resource.getId(), "WIOFFLOCK");
      mobileHost.getLocalCacheManager().addResource(resource,
          transaction);
      return true;
    }
  }
  return false;
}

/**
 */
public boolean requestWOffLock(Transaction transaction,
    Resource resource) {
  if (transaction.getStatus().equals("ACTIVE")) {
    if (transactionManager.getDatabaseServer().requestWOffLock(
        resource, transaction)) {
      mobileHost.getLocalCacheManager().addResource(resource,
          transaction);
      return true;
    }
  }
  return false;
```

```
}

/**
 */
public boolean requestBrowseLock(Transaction transaction,
    Resource resource) {
  if (transaction.getStatus().equals("ACTIVE")) {
    if (transactionManager.getDatabaseServer().requestBrowseLock(
        resource, transaction)) {
      mobileHost.getLocalCacheManager().addResource(resource,
          transaction);
      return true;
    }
  }
  return false;
}

/**
 */
public boolean requestWOnLock(Transaction transaction,
    Resource resource) {
  if (transaction.getStatus().equals("ACTIVE")) {
    if (transactionManager.getDatabaseServer().requestWOnLock(
        resource, transaction)) {
      return true;
    }
  }
  return false;
}

/**
 */
public void commitTransaction(Transaction transaction) {
  transactionManager.getDatabaseServer().commit(
      (CommitOperation) transaction
          .getOperation("COMMITOPERATION"));
  this.transaction.remove(transaction);
}

/**
 */
public void abortTransaction(Transaction transaction) {
  transactionManager.getDatabaseServer().abort(
      (AbortOperation) transaction.getOperation("ABORTOPERATION"));
  this.transaction.remove(transaction);
}

/**
 * Getter of the property <tt>mobileHost</tt>
```

```
 *
 * @return Returns the mobileHost.
 * @uml.property name="mobileHost"
 */
public MobileHost getMobileHost() {
  return mobileHost;
}

/**
 * Setter of the property <tt>mobileHost</tt>
 *
 * @param mobileHost
 *          The mobileHost to set.
 * @uml.property name="mobileHost"
 */
public void setMobileHost(MobileHost mobileHost) {
  this.mobileHost = mobileHost;
}

/**
 * Getter of the property <tt>transactionManager</tt>
 *
 * @return Returns the transactionManager.
 * @uml.property name="transactionManager"
 */
public TransactionManager getTransactionManager() {
  return transactionManager;
}

/**
 * Setter of the property <tt>transactionManager</tt>
 *
 * @param transactionManager
 *          The transactionManager to set.
 * @uml.property name="transactionManager"
 */
public void setTransactionManager(
    TransactionManager transactionManager) {
  this.transactionManager = transactionManager;
}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Collection<Transaction> getTransaction() {
  return transaction;
```

```
  }

  /**
   * Setter of the property <tt>transaction</tt>
   *
   * @param transaction
   *          The transaction to set.
   * @uml.property name="transaction"
   */
  public void setTransaction(Collection<Transaction> transaction) {
    this.transaction = transaction;
  }

  /**
   * Getter of the property <tt>inconsistenBrowseLocks</tt>
   *
   * @return Returns the inconsistenBrowseLocks.
   * @uml.property name="inconsistentBrowseLocks"
   */
  public Collection<BrowseLock> getInconsistentBrowseLocks() {
    return inconsistentBrowseLocks;
  }

  /**
   * Setter of the property <tt>inconsistenBrowseLocks</tt>
   * @param inconsistenBrowseLocks  The inconsistenBrowseLocks to
    set.
   * @uml.property  name="inconsistentBrowseLocks"
   */
  public void setInconsistentBrowseLocks(
      Collection<BrowseLock> inconsistentBrowseLocks) {
    this.inconsistentBrowseLocks = inconsistentBrowseLocks;
  }
}
```

C.3.3 ExportTransaction

```
package transaction;

import java.util.Iterator;

import dataBase.Resource;
import lock.Lock;
import mobileAffiliation.MobileAffiliation;
import mobileAffiliation.SharedData;

public class ExportTransaction {
```

```
/**
 * @uml.property name="id"
 */
private String id;

/**
 * @uml.property name="transaction"
 * @uml.associationEnd
  inverse="exportTransaction:transaction.Transaction"
 */
private Transaction transaction;

/**
 * @uml.property name="mobileAffiliation"
 * @uml.associationEnd
  inverse="exportTransaction:mobileAffiliation.MobileAffiliation"
 */
private MobileAffiliation mobileAffiliation;

/**
 * @uml.property name="newValue"
 */
private int newValue;

/**
 * @uml.property name="shareToRead"
 */
private boolean shareToRead;

/**
 * @uml.property name="resource"
 */
private Resource resource;

/**
 * @uml.property name="sharedData"
 */
private SharedData sharedData;

/**
 * @uml.property name="forcedCommitOnAbortDependency"
 */
private boolean forcedCommitOnAbortDependency;

public ExportTransaction(String id, Transaction transaction,
    MobileAffiliation mobileAffiliation, boolean shareToRead,
    Resource resource, int newValue) {
  this.id = id;
```

```
  this.transaction = transaction;
  this.mobileAffiliation = mobileAffiliation;
  this.shareToRead = shareToRead;
  this.resource = resource;
  this.newValue = newValue;
  sharedData = null;
  forcedCommitOnAbortDependency = false;
}

/**
 */
public void export() {
  if (!shareToRead) {
    transaction.getOwner().getLocalCacheManager().removeResource(
        resource, transaction);
    Lock wOffLock = null;
    for (Iterator<Lock> i = transaction.getLocks().iterator(); i
        .hasNext();) {
      Lock lock = i.next();
      if (lock.getResource().equals(this.resource)) {
        wOffLock = lock;
      }
    }
    wOffLock.setTransaction(null);
    transaction.getNewestWriteOperation(this.resource)
        .setCommitResponsibility(false);
    sharedData = this.mobileAffiliation.shareToWrite(this,
        this.resource, this.newValue, wOffLock);
  } else {
    sharedData = this.mobileAffiliation.shareToRead(this,
        this.resource, this.newValue);
  }
}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Transaction getTransaction() {
  return transaction;
}

/**
 * Setter of the property <tt>transaction</tt>
 *
 * @param transaction
 *          The transaction to set.
```

```
 * @uml.property name="transaction"
 */
public void setTransaction(Transaction transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>mobileAffiliation</tt>
 *
 * @return Returns the mobileAffiliation.
 * @uml.property name="mobileAffiliation"
 */
public MobileAffiliation getMobileAffiliation() {
  return mobileAffiliation;
}

/**
 * Setter of the property <tt>mobileAffiliation</tt>
 *
 * @param mobileAffiliation
 *          The mobileAffiliation to set.
 * @uml.property name="mobileAffiliation"
 */
public void setMobileAffiliation(
    MobileAffiliation mobileAffiliation) {
  this.mobileAffiliation = mobileAffiliation;
}

/**
 * Getter of the property <tt>shareToRead</tt>
 *
 * @return Returns the shareToRead.
 * @uml.property name="shareToRead"
 */
public boolean getShareToRead() {
  return shareToRead;
}

/**
 * Setter of the property <tt>shareToRead</tt>
 *
 * @param shareToRead
 *          The shareToRead to set.
 * @uml.property name="shareToRead"
 */
public void setShareToRead(boolean shareToRead) {
  this.shareToRead = shareToRead;
}
```

```
/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>newValue</tt>
 *
 * @return Returns the newValue.
 * @uml.property name="newValue"
 */
public int getNewValue() {
  return newValue;
}

/**
 * Setter of the property <tt>newValue</tt>
 *
 * @param newValue
 *          The newValue to set.
 * @uml.property name="newValue"
 */
public void setNewValue(int newValue) {
  this.newValue = newValue;
}

/**
 * Getter of the property <tt>sharedData</tt>
 *
 * @return Returns the sharedData.
 * @uml.property name="sharedData"
 */
public SharedData getSharedData() {
```

```
  return sharedData;
}

/**
 * Setter of the property <tt>sharedData</tt>
 *
 * @param sharedData
 *          The sharedData to set.
 * @uml.property name="sharedData"
 */
public void setSharedData(SharedData sharedData) {
  this.sharedData = sharedData;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>forcedCommitOnAbortDependency</tt>
 *
 * @return Returns the forcedCommitOnAbortDependency.
 * @uml.property name="forcedCommitOnAbortDependency"
 */
public boolean getForcedCommitOnAbortDependency() {
  return forcedCommitOnAbortDependency;
}

/**
 * Setter of the property <tt>forcedCommitOnAbortDependency</tt>
 *
 * @param forcedCommitOnAbortDependency
```

```
   *          The forcedCommitOnAbortDependency to set.
   * @uml.property name="forcedCommitOnAbortDependency"
   */
  public void setForcedCommitOnAbortDependency(
      boolean forcedCommitOnAbortDependency) {
    this.forcedCommitOnAbortDependency =
      forcedCommitOnAbortDependency;
  }
}
```

C.3.4 ImportTransaction

```
package transaction;

import operation.WriteOperation;
import dataBase.Resource;
import mobileAffiliation.MobileAffiliation;
import mobileAffiliation.SharedData;

public class ImportTransaction {

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="transaction"
   * @uml.associationEnd
    inverse="importTransaction:transaction.Transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="mobileAffiliation"
   * @uml.associationEnd
    inverse="importTransaction:mobileAffiliation.MobileAffiliation"
   */
  private MobileAffiliation mobileAffiliation;

  /**
   * @uml.property name="shareToRead"
   */
  private boolean shareToRead;

  /**
   * @uml.property name="resource"
```

```
 */
private Resource resource;

/**
 * @uml.property name="sharedData"
 */
private SharedData sharedData;

public ImportTransaction(String id, Transaction transaction,
    MobileAffiliation mobileAffiliation, boolean shareToRead,
    Resource resource) {
  this.id = id;
  this.transaction = transaction;
  this.mobileAffiliation = mobileAffiliation;
  this.shareToRead = shareToRead;
  this.resource = resource;
}

/**
 */
public void importSharedData() {
  if (shareToRead) {
    sharedData = this.mobileAffiliation
        .importSharedToReadSharedData(this.resource.getId());
    transaction.getOwner().getLocalCacheManager().addResource(
        sharedData.getResource(), transaction);
  } else {
    sharedData = this.mobileAffiliation
        .importSharedToWriteSharedData(this.resource.getId());
    transaction.getForcedCommitOnAbortDependentExportTransaction()
        .add(sharedData.getExportTransaction());
    sharedData.getExportTransaction()
        .setForcedCommitOnAbortDependency(true);
    if (sharedData.getWritePermission()) {
      sharedData.getLock().setTransaction(transaction);
      transaction.getLocks().add(sharedData.getLock());
      transaction.getOwner().getLocalCacheManager().addResource(
          sharedData.getResource(), transaction);
      WriteOperation newWriteOperation = new WriteOperation(
          transaction, sharedData.getResource(), sharedData
              .getNewValue());
      transaction.getOperation().add(newWriteOperation);
      transaction.getWrittenResources().add(
          sharedData.getResource());
    } else {
      // This dosent happen. If a transaction wants to import shared
      // data with shareToWrite,
      // someone has exported it with shareToWrite.
    }
```

```
  }
}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Transaction getTransaction() {
  return transaction;
}

/**
 * Setter of the property <tt>transaction</tt>
 *
 * @param transaction
 *          The transaction to set.
 * @uml.property name="transaction"
 */
public void setTransaction(Transaction transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>mobileAffiliation</tt>
 *
 * @return Returns the mobileAffiliation.
 * @uml.property name="mobileAffiliation"
 */
public MobileAffiliation getMobileAffiliation() {
  return mobileAffiliation;
}

/**
 * Setter of the property <tt>mobileAffiliation</tt>
 *
 * @param mobileAffiliation
 *          The mobileAffiliation to set.
 * @uml.property name="mobileAffiliation"
 */
public void setMobileAffiliation(
    MobileAffiliation mobileAffiliation) {
  this.mobileAffiliation = mobileAffiliation;
}

/**
 * Getter of the property <tt>shareToRead</tt>
 *
```

```
 * @return Returns the shareToRead.
 * @uml.property name="shareToRead"
 */
public boolean getShareToRead() {
  return shareToRead;
}

/**
 * Setter of the property <tt>shareToRead</tt>
 *
 * @param shareToRead
 *          The shareToRead to set.
 * @uml.property name="shareToRead"
 */
public void setShareToRead(boolean shareToRead) {
  this.shareToRead = shareToRead;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}
```

```
  /**
   * Setter of the property <tt>id</tt>
   *
   * @param id
   *          The id to set.
   * @uml.property name="id"
   */
  public void setId(String id) {
    this.id = id;
  }

  /**
   * Getter of the property <tt>sharedData</tt>
   *
   * @return Returns the sharedData.
   * @uml.property name="sharedData"
   */
  public SharedData getSharedData() {
    return sharedData;
  }

  /**
   * Setter of the property <tt>sharedData</tt>
   * @param sharedData  The sharedData to set.
   * @uml.property  name="sharedData"
   */
  public void setSharedData(SharedData sharedData) {
    this.sharedData = sharedData;
  }
}
```

C.4 Modulen *lock*

Den nye klassen i modulen *lock* er:

- `BrowseLock`

package lock;

import transaction.Transaction; import dataBase.Resource;

/** */ public class BrowseLock implements Lock /** * @uml.property name=transaction"*/ private Transaction transaction;

/** * @uml.property name=resource"*/ private Resource resource;

/** * @uml.property name=lockLogId"*/ private String lockLogId;

/** * @uml.property name=grantedWOnLock"*/ private WOnLock grantedWOnLock;

public BrowseLock(Transaction transaction,Resource resource)

this.transaction = transaction; this.resource = resource; lockLogId = ; grantedWOnLock = null;

public String getLockType() return BROWSELOCK";

/** * Getter of the property <tt>lockLogId</tt> * * @return Returns the lockLogId. * @uml.property name=lockLogId"*/ public String getLockLogId() return lockLogId;

/** * Setter of the property <tt>lockLogId</tt> * * @param lockLogId * The lockLogId to set. * @uml.property name=lockLogId"*/ public void setLockLogId(String lockLogId) this.lockLogId = lockLogId;

/** * Getter of the property <tt>transaction</tt> * * @return Returns the transaction. * @uml.property name=transaction"*/ public Transaction getTransaction() return transaction;

/** * Setter of the property <tt>transaction</tt> * * @param transaction * The transaction to set. * @uml.property name=transaction"*/ public void setTransaction(Transaction transaction) this.transaction = transaction;

/** * Getter of the property <tt>resource</tt> * * @return Returns the resource. * @uml.property name=resource"*/ public Resource getResource() return resource;

/** * Setter of the property <tt>resource</tt> * * @param resource * The resource to set. * @uml.property name=resource"*/

public void setResource(Resource resource) this.resource = resource;

/** * Getter of the property <tt>grantedWOnLock</tt> * * @return Returns the grantedWOnLock. * @uml.property name=grantedWOnLock"*/ public WOnLock getGrantedWOnLock() return grantedWOnLock;

/** * Setter of the property <tt>grantedWOnLock</tt> * * @param grantedWOnLock * The grantedWOnLock to set. * @uml.property name=ggrantedWOnLock"*/

public void setGrantedWOnLock(WOnLock grantedWOnLock)

this.grantedWOnLock = grantedWOnLock;

C.4.1 `BrowseLock`

C.5 Modulen *operation*

Den utvidede klassen i modulen *operation* er:

- `WriteOperation`

C.5.1 `WriteOperation`

```
package operation;

import dataBase.Resource;

import java.util.ArrayList;
import java.util.Collection;
import transaction.Transaction;

public class WriteOperation implements Operation {

  /**
   * @uml.property name="transaction"
   */
  private Transaction transaction;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="newValue"
   */
  private int newValue;

  /**
   * @uml.property name="commitResponsibility"
   */
  private boolean commitResponsibility;

  /**
   * @uml.property name="readOperationsForReadBrowsedValueAgain"
   */
  private Collection<ReadOperation>
    readOperationsForReadBrowsedValueAgain;

  public WriteOperation(Transaction transaction, Resource resource,
      int newValue) {
```

```
  this.transaction = transaction;
  this.resource = resource;
  this.newValue = newValue;
  this.commitResponsibility = true;
  readOperationsForReadBrowsedValueAgain = new
    ArrayList<ReadOperation>();
}

public String getOperationType() {
  return "WRITEOPERATION";
}

/**
 */
public void registerForReadingBrowsedValueAgain(
    ReadOperation readOperation) {
  readOperationsForReadBrowsedValueAgain.add(readOperation);

}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Transaction getTransaction() {
  return transaction;
}

/**
 * Setter of the property <tt>transaction</tt>
 *
 * @param transaction
 *          The transaction to set.
 * @uml.property name="transaction"
 */
public void setTransaction(Transaction transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
```

```
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>newValue</tt>
 *
 * @return Returns the newValue.
 * @uml.property name="newValue"
 */
public int getNewValue() {
  return newValue;
}

/**
 * Setter of the property <tt>newValue</tt>
 *
 * @param newValue
 *          The newValue to set.
 * @uml.property name="newValue"
 */
public void setNewValue(int newValue) {
  this.newValue = newValue;
}

/**
 * Getter of the property <tt>commitResponsibility</tt>
 *
 * @return Returns the commitResponsibility.
 * @uml.property name="commitResponsibility"
 */
public boolean getCommitResponsibility() {
  return commitResponsibility;
}

/**
 * Setter of the property <tt>commitResponsibility</tt>
 *
 * @param commitResponsibility
 *          The commitResponsibility to set.
```

```
   * @uml.property name="commitResponsibility"
   */
  public void setCommitResponsibility(boolean commitResponsibility) {
    this.commitResponsibility = commitResponsibility;
  }

  /**
   * Getter of the property
    <tt>readOperationsForReadBrowsedValueAgain</tt>
   *
   * @return Returns the readOperationsForReadBrowsedValueAgain.
   * @uml.property name="readOperationsForReadBrowsedValueAgain"
   */
  public Collection<ReadOperation>
    getReadOperationsForReadBrowsedValueAgain() {
    return readOperationsForReadBrowsedValueAgain;
  }

  /**
   * Setter of the property
    <tt>readOperationsForReadBrowsedValueAgain</tt>
   *
   * @param readOperationsForReadBrowsedValueAgain
   *          The readOperationsForReadBrowsedValueAgain to set.
   * @uml.property name="readOperationsForReadBrowsedValueAgain"
   */
  public void setReadOperationsForReadBrowsedValueAgain(
      Collection<ReadOperation> readOperationsForReadBrowsedValueAgain) {
    this.readOperationsForReadBrowsedValueAgain =
      readOperationsForReadBrowsedValueAgain;
  }
}
```

C.6 Modulen *input*

Denne modulen er lik den i online-offline.

C.7 Modulen *mobileAffiliation*

Modulen *mobileAffiliation* er en ny modul. Klassene i denne modulen er:

- `MobileAffiliation`
- `ExportImportRepository`
- `SharedData`

C.7.1 `MobileAffiliation`

```
package mobileAffiliation;

import java.util.ArrayList;
import java.util.Collection;

import lock.Lock;

import dataBase.Resource;
import transaction.ExportTransaction;
import transaction.ImportTransaction;
import transaction.Transaction;

public class MobileAffiliation {

  /**
   * @uml.property name="exportImportRepository"
   * @uml.associationEnd
    inverse="mobileAffiliation:mobileAffiliation.ExportImportRepository"
   */
  private ExportImportRepository exportImportRepository;

  /**
   * @uml.property name="id"
   */
  private String id;

  /**
   * @uml.property name="initiator"
   */
  private Transaction initiator;

  /**
   * @uml.property name="transaction"
   * @uml.associationEnd multiplicity="(0 -1)"
```

```
 *
  inverse="mobileAffiliation:transaction.Transaction"
 */
private Collection<Transaction> transaction;

/**
 * @uml.property name="exportTransaction"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
  inverse="mobileAffiliation:transaction.ExportTransaction"
 */
private Collection<ExportTransaction> exportTransaction;

/**
 * @uml.property name="importTransaction"
 * @uml.associationEnd multiplicity="(0 -1)"
 *
  inverse="mobileAffiliation:transaction.ImportTransaction"
 */
private Collection<ImportTransaction> importTransaction;

public MobileAffiliation(Transaction initiator, String id) {
  exportImportRepository = new ExportImportRepository(this);
  this.initiator = initiator;
  transaction = new ArrayList<Transaction>();
  transaction.add(initiator);
  this.id = id;
  exportTransaction = new ArrayList<ExportTransaction>();
  importTransaction = new ArrayList<ImportTransaction>();
  System.out.println("Transaction " + initiator.getId()
      + " creates MA " + id);
}

/**
 */
public void register(Transaction transaction) {
  this.transaction.add(transaction);
  System.out.println("Transaction " + transaction.getId()
      + " registers to MA " + id);
}

/**
 */
public void unRegister(Transaction transaction) {
  this.transaction.remove(transaction);
  System.out.println("Transaction " + transaction.getId()
      + " unregisters from MA " + id);
}
```

```
/**
 */
public SharedData shareToRead(ExportTransaction exportTransaction,
    Resource resource, int newValue) {
  SharedData sharedData = new SharedData(exportImportRepository,
      exportTransaction, false, resource, newValue);
  exportImportRepository.getSharedData().add(sharedData);
  return sharedData;
}

/**
 */
public SharedData shareToWrite(
    ExportTransaction exportTransaction, Resource resource,
    int newValue, Lock lock) {
  SharedData sharedData = new SharedData(exportImportRepository,
      exportTransaction, true, resource, newValue);
  sharedData.setLock(lock);
  exportImportRepository.getSharedData().add(sharedData);
  return sharedData;
}

/**
 */
public SharedData importSharedToReadSharedData(String resourceId) {
  SharedData sharedData = exportImportRepository
      .getSharedData(resourceId);
  return sharedData;

}

/**
 */
public SharedData importSharedToWriteSharedData(String resourceId) {
  SharedData sharedData = exportImportRepository
      .getSharedData(resourceId);
  exportImportRepository.getSharedData().remove(sharedData);
  return sharedData;

}

/**
 * Getter of the property <tt>exportImportRepository</tt>
 *
 * @return Returns the exportImportRepository.
 * @uml.property name="exportImportRepository"
 */
public ExportImportRepository getExportImportRepository() {
  return exportImportRepository;
```

```
}

/**
 * Setter of the property <tt>exportImportRepository</tt>
 *
 * @param exportImportRepository
 *          The exportImportRepository to set.
 * @uml.property name="exportImportRepository"
 */
public void setExportImportRepository(
    ExportImportRepository exportImportRepository) {
  this.exportImportRepository = exportImportRepository;
}

/**
 * Getter of the property <tt>id</tt>
 *
 * @return Returns the id.
 * @uml.property name="id"
 */
public String getId() {
  return id;
}

/**
 * Setter of the property <tt>id</tt>
 *
 * @param id
 *          The id to set.
 * @uml.property name="id"
 */
public void setId(String id) {
  this.id = id;
}

/**
 * Getter of the property <tt>initiator</tt>
 *
 * @return Returns the initiator.
 * @uml.property name="initiator"
 */
public Transaction getInitiator() {
  return initiator;
}

/**
 * Setter of the property <tt>initiator</tt>
 *
 * @param initiator
```

```
 *          The initiator to set.
 * @uml.property name="initiator"
 */
public void setInitiator(Transaction initiator) {
  this.initiator = initiator;
}

/**
 * Getter of the property <tt>transaction</tt>
 *
 * @return Returns the transaction.
 * @uml.property name="transaction"
 */
public Collection<Transaction> getTransaction() {
  return transaction;
}

/**
 * Setter of the property <tt>transaction</tt>
 *
 * @param transaction
 *          The transaction to set.
 * @uml.property name="transaction"
 */
public void setTransaction(Collection<Transaction> transaction) {
  this.transaction = transaction;
}

/**
 * Getter of the property <tt>exportTransaction</tt>
 *
 * @return Returns the exportTransaction.
 * @uml.property name="exportTransaction"
 */
public Collection<ExportTransaction> getExportTransaction() {
  return exportTransaction;
}

/**
 * Setter of the property <tt>exportTransaction</tt>
 *
 * @param exportTransaction
 *          The exportTransaction to set.
 * @uml.property name="exportTransaction"
 */
public void setExportTransaction(
    Collection<ExportTransaction> exportTransaction) {
  this.exportTransaction = exportTransaction;
}
```

```
  /**
   * Getter of the property <tt>importTransaction</tt>
   *
   * @return Returns the importTransaction.
   * @uml.property name="importTransaction"
   */
  public Collection<ImportTransaction> getImportTransaction() {
    return importTransaction;
  }

  /**
   * Setter of the property <tt>importTransaction</tt>
   *
   * @param importTransaction
   *          The importTransaction to set.
   * @uml.property name="importTransaction"
   */
  public void setImportTransaction(
      Collection<ImportTransaction> importTransaction) {
    this.importTransaction = importTransaction;
  }
}
```

C.7.2 ExportImportRepository

```
package mobileAffiliation;

import java.util.ArrayList;
import java.util.Collection;
import java.util.Iterator;

public class ExportImportRepository {

  /**
   * @uml.property name="mobileAffiliation"
   * @uml.associationEnd
    inverse="exportImportRepository:mobileAffiliation.MobileAffiliation"
   */
  private MobileAffiliation mobileAffiliation;

  /**
   * @uml.property name="sharedData"
   * @uml.associationEnd multiplicity="(0 -1)"
   *
    inverse="exportImportRepository:mobileAffiliation.SharedData"
   */
```

```
private Collection<SharedData> sharedData;

public ExportImportRepository(MobileAffiliation mobileAffiliation) {
  this.mobileAffiliation = mobileAffiliation;
  sharedData = new ArrayList<SharedData>();
}

public SharedData getSharedData(String resourceId) {
  for (Iterator<SharedData> i = sharedData.iterator(); i.hasNext();) {
    SharedData sharedData = i.next();
    if (sharedData.getResource().getId().equals(resourceId)) {
      return sharedData;
    }
  }
  return null;
}

/**
 * Getter of the property <tt>mobileAffiliation</tt>
 *
 * @return Returns the mobileAffiliation.
 * @uml.property name="mobileAffiliation"
 */
public MobileAffiliation getMobileAffiliation() {
  return mobileAffiliation;
}

/**
 * Setter of the property <tt>mobileAffiliation</tt>
 *
 * @param mobileAffiliation
 *          The mobileAffiliation to set.
 * @uml.property name="mobileAffiliation"
 */
public void setMobileAffiliation(
    MobileAffiliation mobileAffiliation) {
  this.mobileAffiliation = mobileAffiliation;
}

/**
 * Getter of the property <tt>sharedData</tt>
 *
 * @return Returns the sharedData.
 * @uml.property name="sharedData"
 */
public Collection<SharedData> getSharedData() {
  return sharedData;
}
```

```
  /**
   * Setter of the property <tt>sharedData</tt>
   *
   * @param sharedData
   *          The sharedData to set.
   * @uml.property name="sharedData"
   */
  public void setSharedData(Collection<SharedData> sharedData) {
    this.sharedData = sharedData;
  }
}
```

C.7.3 SharedData

```
package mobileAffiliation;

import lock.Lock;
import dataBase.Resource;
import transaction.ExportTransaction;

public class SharedData {

  /**
   * @uml.property name="exportImportRepository"
   * @uml.associationEnd
    inverse="sharedData:mobileAffiliation.ExportImportRepository"
   */
  private ExportImportRepository exportImportRepository;

  /**
   * @uml.property name="writePermission"
   */
  private boolean writePermission;

  /**
   * @uml.property name="resource"
   */
  private Resource resource;

  /**
   * @uml.property name="newValue"
   */
  private int newValue;

  /**
   * @uml.property name="lock"
   */
```

```
private Lock lock;

/**
 * @uml.property name="exportTransaction"
 */
private ExportTransaction exportTransaction;

public SharedData(ExportImportRepository exportImportRepository,
    ExportTransaction exportTransaction, boolean writePermission,
    Resource resource, int newValue) {
  this.exportImportRepository = exportImportRepository;
  this.writePermission = writePermission;
  this.resource = resource;
  this.newValue = newValue;
  this.lock = null;
  this.exportTransaction = exportTransaction;
}

/**
 * Getter of the property <tt>exportImportRepository</tt>
 *
 * @return Returns the exportImportRepository.
 * @uml.property name="exportImportRepository"
 */
public ExportImportRepository getExportImportRepository() {
  return exportImportRepository;
}

/**
 * Setter of the property <tt>exportImportRepository</tt>
 *
 * @param exportImportRepository
 *          The exportImportRepository to set.
 * @uml.property name="exportImportRepository"
 */
public void setExportImportRepository(
    ExportImportRepository exportImportRepository) {
  this.exportImportRepository = exportImportRepository;
}

/**
 * Getter of the property <tt>writePermission</tt>
 *
 * @return Returns the writePermission.
 * @uml.property name="writePermission"
 */
public boolean getWritePermission() {
  return writePermission;
}
```

```
/**
 * Setter of the property <tt>writePermission</tt>
 *
 * @param writePermission
 *          The writePermission to set.
 * @uml.property name="writePermission"
 */
public void setWritePermission(boolean writePermission) {
  this.writePermission = writePermission;
}

/**
 * Getter of the property <tt>resource</tt>
 *
 * @return Returns the resource.
 * @uml.property name="resource"
 */
public Resource getResource() {
  return resource;
}

/**
 * Setter of the property <tt>resource</tt>
 *
 * @param resource
 *          The resource to set.
 * @uml.property name="resource"
 */
public void setResource(Resource resource) {
  this.resource = resource;
}

/**
 * Getter of the property <tt>newValue</tt>
 *
 * @return Returns the newValue.
 * @uml.property name="newValue"
 */
public int getNewValue() {
  return newValue;
}

/**
 * Setter of the property <tt>newValue</tt>
 *
 * @param newValue
 *          The newValue to set.
 * @uml.property name="newValue"
```

```
   */
  public void setNewValue(int newValue) {
    this.newValue = newValue;
  }

  /**
   * Getter of the property <tt>lock</tt>
   *
   * @return Returns the lock.
   * @uml.property name="lock"
   */
  public Lock getLock() {
    return lock;
  }

  /**
   * Setter of the property <tt>lock</tt>
   *
   * @param lock
   *          The lock to set.
   * @uml.property name="lock"
   */
  public void setLock(Lock lock) {
    this.lock = lock;
  }

  /**
   * Getter of the property <tt>exportTransaction</tt>
   *
   * @return Returns the exportTransaction.
   * @uml.property name="exportTransaction"
   */
  public ExportTransaction getExportTransaction() {
    return exportTransaction;
  }

  /**
   * Setter of the property <tt>exportTransaction</tt>
   *
   * @param exportTransaction
   *          The exportTransaction to set.
   * @uml.property name="exportTransaction"
   */
  public void setExportTransaction(
      ExportTransaction exportTransaction) {
    this.exportTransaction = exportTransaction;
  }
}
```

D Testdokumentasjon for den adaptive låseprotokollen

For å teste låseprotokollen skrives ulike testcase for de situasjonene som ønskes testet.

D.1 Mal for inndatafiler for den adaptive låseprotokollen

Malen for inndatafilene er forandret i forhold til online-offline. Linjer som starter med # er kommentarer og vil ignoreres av metoden som leser inn testcaset. Malen for inndatafilene ser slik ut:

```
#Create server (id)
CREATE_SERVER 0

#Number of resources at server
NOF_RESOURCES 2

#Number of MSS
NOF_MOBILESUPPORTSTATIONS 1

#Create MSS (mss_id, db_id)
CREATE_MOBILESUPPORTSTATION 0:0

#Number of MH
NOF_MOBILEHOSTS 2

#Create MH (mh_id, mss_id, casheSize, localTime)
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

#START_ONLINETRANSACTION (mh_id, t_id)
#START_OFFLINETRANSACTION (mh_id, t_id)
#READ_ONLINETRANSACTION (mh_id,t_id,r_id)
#READ_OFFLINETRANSACTION (mh_id,t_id,r_id)
#WRITE_ONLINETRANSACTION (mh_id,t_id,r_id, new_value)
#WRITE_OFFLINETRANSACTION (mh_id,t_id,r_id, new_value)
#COMMIT_ONLINETRANSACTION (mh_id,t_id)
#COMMIT_OFFLINETRANSACTION (mh_id,t_id)
#ABORT_ONLINETRANSACTION (mh_id,t_id)
#ABORT_OFFLINETRANSACTION (mh_id,t_id)

#REQUEST_RONLOCK (mh_id,t_id,r_id)
```

```
#REQUEST_WONLOCK (mh_id,t_id,r_id)
#REQUEST_WIOFFLOCK (mh_id,t_id,r_id)
#REQUEST_WOFFLOCK (mh_id,t_id,r_id)
#REQUEST_BROWSELOCK_ONLINETRANSACTION (mh_id,t_id,r_id)
#REQUEST_BROWSELOCK_OFFLINETRANSACTION (mh_id,t_id,r_id)

#DISCONNECT_PLANNED (mh_id)
#DISCONNECT_UNPLANNED (mh_id)
#RECONNECT (mh_id)

#INITIATE_MA (mh_id,t_id,ma_id)
#REGISTER_MA (mh_id,t_id,initiator_mh_id,initiator_t_id,ma_id)
#UNREGISTER_MA (mh_id,t_id,initiator_mh_id,initiator_t_id,ma_id)

#EXPORT_SHARE_TO_READ
  (mh_id,t_id,r_id,initiator_mh_id,initiator_t_id,ma_id,ext_id)
#EXPORT_SHARE_TO_WRITE
  (mh_id,t_id,r_id,initiator_mh_id,initiator_t_id,ma_id,ext_id)
#IMPORT_SHARE_TO_READ
  (mh_id,t_id,r_id,initiator_mh_id,initiator_t_id,ma_id,ext_id)
#IMPORT_SHARE_TO_WRITE
  (mh_id,t_id,r_id,initiator_mh_id,initiator_t_id,ma_id,ext_id)
END_FILE
```

D.2 Testcase 1

Testcase 1 ønsker å teste om onlinetransaksjoner og offlinetransaksjoner som browser en wofflåst ressurs, leser den modifiserte verdien igjen. Testcaset tester også om browselåser bevilges, når det eksisterer en wofflås på en ressurs. Testcaset tester også om browselåser delegeres til wiofflåser og ronlåser, når wonlåsen blir frigitt.

Beskrivelse av testcase 1
En offlinetransaksjon forespør etter en wofflås på en ressurs og får bevilget denne. Deretter startes en onlinetransaksjon og en offlinetransaksjon på en annen mobil enhet, som spør etter browselåser på ressursen. Begge mobile enhetene frakobles. Offlinetransaksjonen med wofflås modifiserer ressursen. Samtidig leser onlinetransaksjonen og offlinetransaksjonen på den andre mobile enhten den umodifiserte verdien. Offlinetransaksjonen med wofflåsen tilkobles og oppgraderer denne til en wonlås. Den committer den modifiserte verdien. Onlinetransaksjonen leser den modifiserte verdien når den committes. Offlinetransaksjonen med browselås tilkobles og leser nå den modifiserte verdien. Onlinetransaksjonen og offlinetransaksjonen committes til slutt.

Forventet resultat for testcase 1

Det forventede resultatet er at alle låsene bevilges. Offlinetransaksjonen med wofflås modifiserer ressursen og committer verdien. De to andre transaksjonene leser den umodifiserte verdien og leser den modifiserte verdien når den committes. Offlinetransaksjonen gjør dette først når den tilkobles. I tillegg skal browselåsen som onlinetransaksjonen holder, delegeres til en ronlås når transaksjonen som har oppgradert wofflåsen til wonlås committer. Browselåsen som offlinetransaksjonen holder skal delegeres til en wiofflås. For å vise at den modifiserte verdien er lest igjen, skrives leseverdiene til leseoperasjonene ut når transaksjonene committer.

Inndata for testcase 1

Inndata for testcase 1:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_OFFLINETRANSACTION 1:1
START_OFFLINETRANSACTION 2:2
START_ONLINETRANSACTION 2:3

REQUEST_WOFFLOCK 1:1:0
REQUEST_BROWSELOCK_OFFLINETRANSACTION 2:2:0
REQUEST_BROWSELOCK_ONLINETRANSACTION 2:3:0

DISCONNECT_PLANNED 1
DISCONNECT_PLANNED 2
WRITE_OFFLINETRANSACTION 1:1:0:7
READ_OFFLINETRANSACTION 2:2:0
READ_ONLINETRANSACTION 2:3:0

RECONNECT 1
REQUEST_WONLOCK 1:1:0
COMMIT_OFFLINETRANSACTION 1:1
RECONNECT 2
COMMIT_OFFLINETRANSACTION 2:2
COMMIT_ONLINETRANSACTION 2:3
```

```
END_FILE
```

Resultat for testcase 1

Resultat for testcase 1:

```
A new resource is created with id 0 and value 8
Mobilehost 1 disconnects planned at time 78688493
Mobilehost 2 disconnects planned at time 78807223
Mobilehost 1 reconnects at time 82066576
Mobilehost 2 reconnects at time 90654259
Transaction 2 committes read with value 7
Transaction 3 committes read with value 7

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

72251641 1 1 0 WOFFLOCK [] [] GRANTED
77042194 2 2 0 BROWSELOCK [0(1,WOFFLOCK)] [] GRANTEDPENDING
77948175 2 3 0 BROWSELOCK [0(1,WOFFLOCK)] [0(2,BROWSELOCK)]
  GRANTEDPENDING
82200950 1 1 0 WONLOCK [0(1,WOFFLOCK)]
  [0(2,BROWSELOCK), 0(3,BROWSELOCK)] GRANTEDSELFUPGRADE
85043211 1 1 0 RELEASE_WONLOCK [0(1,WONLOCK)]
  [0(2,BROWSELOCK), 0(3,BROWSELOCK)] RELEASEDDELEGATEBROWSELOCKS
90892558 2 2 0 RELEASE_PENDINGWIOFFLOCK [0(3,RONLOCK)]
  [0(2,WIOFFLOCK)] RELEASED
91128342 2 3 0 RELEASE_RONLOCK [0(3,RONLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue

71340352 1 START
71555184 2 START
71654638 3 START
81760671 3 READ 0 8
85365878 1 COMMIT
90983910 2 COMMIT
91162145 3 COMMIT

Mobilehost 1s local log:

Localtime TransactionId action ResourceId  value newValue

80652709 1 WRITEOFFLINE 0 8 7
```

```
Mobilehost 2s locklog:

Time TransactionId ResourceId grantedLock

85043211 2 0 WIOFFLOCK

Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

81557014 2 READOFFLINE 0 8
```

Diskusjon for testcase 1
Resultatet er likt det forventede resultatet. Man ser at broswelåsene blir bevilget og lagt i `pendingLocks`. Man ser at offlinetransaksjonen modifiserer ressursen med startverdi 8 til 7. Onlinetransaksjonen og offlinetransaksjonen leser verdien 8 når ressursen browses, men når de committer er denne verdien 7. I tillegg ser man at browselåsene er delegert riktig etter at wonlåsen er frigitt.

D.3 Testcase 2

Testcase 2 ønsker å teste om transaksjoner som browser en ressurs, men committer før den modifiserte verdien er committet, faktisk ikke leser den modifiserte verdien.

Beskrivelse av testcase 2
Testcase 2 er lik testcase 1, bortsett fra at her committes transaksjonene som browser ressursen før den modifiserte verdien har committet.

Forventet resultat for testcase 2
Onlinetransaksjonen og offlinetransaksjonen committer med den umodifiserte verdien.

Inndata for testcase 2
Inndata for testcase 2:

```
CREATE_SERVER 0
```

```
NOF_RESOURCES 2

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_OFFLINETRANSACTION 1:1
START_OFFLINETRANSACTION 2:2
START_ONLINETRANSACTION 2:3

REQUEST_WOFFLOCK 1:1:0
REQUEST_BROWSELOCK_OFFLINETRANSACTION 2:2:0
REQUEST_BROWSELOCK_ONLINETRANSACTION 2:3:0

DISCONNECT_PLANNED 1
DISCONNECT_PLANNED 2
WRITE_OFFLINETRANSACTION 1:1:0:7
READ_OFFLINETRANSACTION 2:2:0
READ_ONLINETRANSACTION 2:3:0

RECONNECT 1
REQUEST_WONLOCK 1:1:0
RECONNECT 2
COMMIT_OFFLINETRANSACTION 2:2
COMMIT_ONLINETRANSACTION 2:3
COMMIT_OFFLINETRANSACTION 1:1
END_FILE
```

Resultat for testcase 2

Resultat for testcase 2:

```
A new resource is created with id 0 and value 10
A new mobile support station is created with id 0
A new mobile host is created with id 1
A new mobile host is created with id 2
Mobilehost 1 disconnects planned at time 85180379
Mobilehost 2 disconnects planned at time 85389623
Mobilehost 1 reconnects at time 91723389
Mobilehost 2 reconnects at time 95054539
Transaction 2 committes read with value 10
Transaction 3 committes read with value 10

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
```

```
  PendingLocks ActionAtDB

72479044 1 1 0 WOFFLOCK [] [] GRANTED
81426829 2 2 0 BROWSELOCK [0(1,WOFFLOCK)] [] GRANTEDPENDING
83144645 2 3 0 BROWSELOCK [0(1,WOFFLOCK)] [0(2,BROWSELOCK)]
  GRANTEDPENDING
91979846 1 1 0 WONLOCK [0(1,WOFFLOCK)]
  [0(2,BROWSELOCK), 0(3,BROWSELOCK)] GRANTEDSELFUPGRADE
106373016 2 2 0 RELEASE_PENDINGBROWSELOCK [0(1,WONLOCK)]
  [0(2,BROWSELOCK), 0(3,BROWSELOCK)] RELEASED
106609918 2 3 0 RELEASE_PENDINGBROWSELOCK [0(1,WONLOCK)]
  [0(3,BROWSELOCK)] RELEASED
106824750 1 1 0 RELEASE_WONLOCK [0(1,WONLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue

e70819615 1 START
71187818 2 START
71382536 3 START
91135605 3 READ 0 10
106454311 2 COMMIT
106653219 3 COMMIT
106862185 1 COMMIT


Mobilehost 1s local log:

Localtime TransactionId action ResourceId  value newValue

88881128 1 WRITEOFFLINE 0 10 7

Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

90735833 2 READOFFLINE 0 10
```

Diskusjon for testcase 2

Resultatet er likt det forventede resultatet. Man ser at onlinetransaksjonen som browser ressursen committer med den umodifiserte verdien.

D.4 Testcase 3

Testcase 3 ønsker å teste eksportering og importering av data mellom to offlinetransaksjoner i en mobil affiliasjon. Dette skal skje med deling-for-skriving.

Beskrivelse av testcase 3
Offlinetransaksjon med id 1 på mobile enhet med id 1 spør etter en wofflås på ressurs med id 0. Offlinetransaksjon med id 2 på mobile enhet med id 2 spør etter en wofflås på ressurs med id 1. Begge transaksjonene frakobles og modifiserer sine ressuser. En mobil affiliasjon opprettes der begge transaksjonene registrerer seg. Offlinetransaksjon med id 2 eksporterer ressurs med id 1 og den modifiserte verdien til EI. Offlinetransaksjon med id 1 importerer denne, leser den modifiserte verdien og modifiserer denne på nytt. Begge transaksjonene tilkobles. Offlinetransaksjon med id 1 committer før, offlinetransaksjon med id 2 committer.

Forventet resultat for testcase 3
Forventet resultat er at offlinetransaksjon med id 1 får importert ressurs med id 1. Når den leser ressursen etter at den er importert skal den lese den modifiserte verdien. I tillegg skal den få lov å modifisere ressursen igjen. Offlinetransaksjon med id 1 prøver å committe før, offlinetransaksjon med id 2. Siden offlinetransaksjoner som importerer ressurser skal serialiseres etter offlinetransaksjonen som eksporterte verdien og tilhørende eksporteringstransaksjon, er forventet resultat at databasetjeneren sørger for at offlinetransaksjon med id 1 committer sist.

Inndata for testcase 3
Inndata for testcase 3:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_OFFLINETRANSACTION 1:1
```

```
START_OFFLINETRANSACTION 2:2

REQUEST_WOFFLOCK 1:1:0
REQUEST_WOFFLOCK 2:2:1

DISCONNECT_PLANNED 1
DISCONNECT_PLANNED 2

WRITE_OFFLINETRANSACTION 1:1:0:17
WRITE_OFFLINETRANSACTION 2:2:1:19

INITIATE_MA 1:1:1
REGISTER_MA 2:2:1:1:1

EXPORT_SHARE_TO_WRITE 2:2:1:1:1:1:7
IMPORT_SHARE_TO_WRITE 1:1:1:1:1:1:9

READ_OFFLINETRANSACTION 1:1:1
WRITE_OFFLINETRANSACTION 1:1:1:13

UNREGISTER_MA 2:2:1:1:1
UNREGISTER_MA 1:1:1:1:1

RECONNECT 1
RECONNECT 2
COMMIT_OFFLINETRANSACTION 1:1
COMMIT_OFFLINETRANSACTION 2:2
END_FILE
```

Resultat for testcase 3

Resultat for testcase 3:

```
A new resource is created with id 0 and value 3
A new resource is created with id 1 and value 1
Mobilehost 1 disconnects planned at time 86761865
Mobilehost 2 disconnects planned at time 86972507
Transaction 1 creates MA 1
Transaction 2 registers to MA 1
Transaction 2 shareToWrite resource 1s value 19 in MA 1
Transaction 1 import shareToWrite resource 1 in MA 1
Transaction 2 unregisters from MA 1
Transaction 1 unregisters from MA 1
Mobilehost 1 reconnects at time 156612084
Mobilehost 2 reconnects at time 156807360

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
```

```
  PendingLocks ActionAtDB

76394067 1 1 0 WOFFLOCK [] [] GRANTED
85436557 2 2 1 WOFFLOCK [] [] GRANTED
161234459 2 2 1 RELEASE_WOFFLOCK [1(1,WOFFLOCK)] [] RELEASED
161638142 1 1 0 RELEASE_WOFFLOCK [0(1,WOFFLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue

74905331 1 START
75284150 2 START
161331957 2 COMMIT
161690383 7 EXPORTTRANSACTIONCOMMIT
161704072 1 COMMIT
161723069 9 IMPORTTRANSACTIONCOMMIT


Mobilehost 1s local log:

Localtime TransactionId action ResourceId  value newValue

90446133 1 WRITEOFFLINE 0 3 17
101314553 9 STARTIMPORTTRANSACTION
155688782 1 READOFFLINE 1 19
155827627 1 WRITEOFFLINE 1 1 13


Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

90696164 2 WRITEOFFLINE 1 1 19
98865638 7 STARTEXPORTTRANSACTION
```

Diskusjon for testcase 3

Resultatet er likt det forventede resultatet. Man ser at transaksjon 1 importerer ressursen som transaksjon 2 har eksportert. Man ser også at transaksjon leser den modifiserte verdien og at den har skriverettigheter på denne ressursen. I tillegg ser man at transaksjon 1 serialiseres etter transaksjon 2 og eksporteringstransaksjonen.

D.5 Testcase 4

Testcase 4 ønsker å teste at den tvunget-commit-ved-abort-avhengigheter overholdes.

Beskrivelse av testcase 4

Testcase 4 er likt testcase 3, bortsett fra at transaksjonen som importerer den delte ressursen aborterer.

Forventet resultat for testcase 4

Forventet resultat er at eksporteringstransaksjonen committer den eksporterte verdien, siden transaksjonen som importerer denne aborterer.

Inndata for testcase 4

Inndata for testcase 4:

```
CREATE_SERVER 0

NOF_RESOURCES 2

NOF_MOBILESUPPORTSTATIONS 1
CREATE_MOBILESUPPORTSTATION 0:0

NOF_MOBILEHOSTS 2
CREATE_MOBILEHOST 1:0:10:0
CREATE_MOBILEHOST 2:0:10:0

START_OFFLINETRANSACTION 1:1
START_OFFLINETRANSACTION 2:2

REQUEST_WOFFLOCK 1:1:0
REQUEST_WOFFLOCK 2:2:1

DISCONNECT_PLANNED 1
DISCONNECT_PLANNED 2

WRITE_OFFLINETRANSACTION 1:1:0:17
WRITE_OFFLINETRANSACTION 2:2:1:19

INITIATE_MA 1:1:1
REGISTER_MA 2:2:1:1:1

EXPORT_SHARE_TO_WRITE 2:2:1:1:1:1:7
```

```
IMPORT_SHARE_TO_WRITE 1:1:1:1:1:1:9

READ_OFFLINETRANSACTION 1:1:1
WRITE_OFFLINETRANSACTION 1:1:1:13

UNREGISTER_MA 2:2:1:1:1
UNREGISTER_MA 1:1:1:1:1

RECONNECT 1
RECONNECT 2

ABORT_OFFLINETRANSACTION 1:1
COMMIT_OFFLINETRANSACTION 2:2
END_FILE
```

Resultat for testcase 4

Resultat for testcase 4:

```
A new resource is created with id 0 and value 7
A new resource is created with id 1 and value 10
Mobilehost 1 disconnects planned at time 95685904
Mobilehost 2 disconnects planned at time 95903809
Transaction 1 creates MA 1
Transaction 2 registers to MA 1
Transaction 2 shareToWrite resource 1s value 19 in MA 1
Transaction 1 import shareToWrite resource 1 in MA 1
Transaction 2 unregisters from MA 1
Transaction 1 unregisters from MA 1
Mobilehost 1 reconnects at time 123835571
Mobilehost 2 reconnects at time 124174720
Exporttransaction 7 commits resource 1

The databaseservers locklog:

Time MobileHostId TransactionId ResourceId RequestingLock CurrentLocks
  PendingLocks ActionAtDB

81853699 1 1 0 WOFFLOCK [] [] GRANTED
93532837 2 2 1 WOFFLOCK [] [] GRANTED
126273590 1 1 0 RELEASE_WOFFLOCK [0(1,WOFFLOCK)] [] RELEASED
126352092 1 1 1 RELEASE_WOFFLOCK [1(1,WOFFLOCK)] [] RELEASED

The databaseservers log:

Time TransactionId action ResourceId  value newValue

80346803 1 START
80720873 2 START
```

```
126388689 1 ABORT
126497362 7 EXPORTTRANSACTIONCOMMIT
127481565 2 COMMIT


Mobilehost 1s local log:

Localtime TransactionId action ResourceId  value newValue

99892304 1 WRITEOFFLINE 0 7 17
111382871 9 STARTIMPORTTRANSACTION
122151837 1 READOFFLINE 1 19
122365272 1 WRITEOFFLINE 1 10 13


Mobilehost 2s local log:

Localtime TransactionId action ResourceId  value newValue

100143174 2 WRITEOFFLINE 1 10 19
108433613 7 STARTEXPORTTRANSACTION
```

Diskusjon for testcase 4
Resultatet er likt det forventede resultatet. Man ser at eksporteringstransaksjonen committer ressurs 1.