T A R T U Ü L I K O O L
FÜÜSIKA-KEEMIATEADUSKOND

Arvutiteaduse instituut
Tarkvarasüsteemide õppetool

Aleksandr Bogdanov

Säilitava olekuga objektide kasutamine CORBA rakendustes
Magistritöö (20 AP)

Juhendajad: Eero Vainikko
Jüri Harju

Tartu 2007

Sisukord

Sissejuhatus....4
1 Lühiülevaade CORBA tehnoloogiast....6
1.1 Lühiülevaade CORBA arhitektuurist....6
1.2 IDL – Liideste defineerimise keel....8
1.3 CORBA lisateenused....8
1.4 CORBA Component Model (CCM)....11
2 Säilitavad objektid CORBA rakenduses....12
2.1 Objektide säilitamise vajadus objektorienteeritud süsteemis ja andmebaasidega integreerimine....12
2.2 Objektide hoidmise vahendid....12
3 CORBA Persistent State Service....15
3.1 PSS andmemudel....16
3.2 PSDL keele kompilaator....18
3.3 PSS realisatsiooni arhitektuur....18
3.4 PSS kasutamise võimalused ja tüüpilised stsenaariumid....20
3.5 Päringute API....23
3.6 PSS Transparent Persistence....24
4 Teised O-R kujutamise raamistikud CORBA rakendustes....26
4.1 Java Persistence API....26
4.2 PSS ja JPA võrdlus....30
5 Transaktsioonid hajussüsteemides....31
5.1 Transaktsioonide juhtimine....31
5.2 Transaktsioonide juhtimise mudelid....33
6 Näidisrakendus....37
6.1 Näidisrakenduse eesmärk....37
6.2 Tehnoloogiliste aspektide analüüs....37
6.3 Kasutatava PSS realisatsiooni eripärad....38
6.4 Testrakenduse valdkond....38
6.5 Kasutuslood....39
6.6 Kasutajaliides....40
6.7 Arhitektuur....41
6.8 Kasutatavad tehnoloogiad ja vahendid....42
6.9 Rakenduse realisatsioon....44
6.10 Tulemused ja järeldused....47
Kokkuvõte....51
Abstract....52
Allikad....53
Lisad....54
Lisa 1: Kasutajajuhend....55
Lisa 2: Testrakenduse IDL ja PSDL failide sisu....56

Lisa 3: Mõnede CORBA realisatsioonide lisateenused 57
Lisa 4: JPA kasutamise näide..60
Lisa 5: Rakenduse lähtekood 60

Sissejuhatus

Tänapäeva infosüsteemid on küllaltki keerulised programmid laia funktsionaalsusega ja rohkete võimalustega. Kui infosüsteem muutub suuremaks, siis reeglina tekib vajadus ühendada omavahel erinevad süsteemi osad, mis võivad olla realiseeritud erinevates programmeerimiskeeltes ja töötada erinevate operatsioonisüsteemide keskkondades. Kui riistvara erinevused on peidetud opsüsteemi kihiga, siis tarkvarakeskkonda erinevuste peitmine on keeruline ülesanne. Heterogeensete rakenduste loomise probleeme aitab edukalt lahendada CORBA tehnoloogia. CORBA võimaldab kasutada abstraktsed liidesed, mis ei sõltu konkreetsest platvormist või programmeerimiskeele omadustest, eraldades realisatsiooni ja selle esitlust. CORBA klient-server komponenttehnoloogia kasutamisel ilmnevad sellise lähenemise plussid – programmi komponendid on võimalik välja vahetada ilma kogu programmi kompileerimata, juhul kui liidesed ei ole muutunud; enamus funktsionaalsusest on realiseeritud serveris ja tänu sellele kliendiprogrammid muutuvad lihtsamateks *(thin client)*; tekib võimalus taaskasutada olemasolevad süsteemi osad; saab leevendada kasutatavate tehnoloogiate uuendamise ja vahetamise probleeme. CORBA suurepäraselt sobib hajussüsteemide loomiseks – kliendid kasutavad serveri funktsionaalsust samal viisil, nagu kasutaks lokaalset programmi (minimaalsete erinevustega).

CORBA kasutamiseks on vajalik selle tehnoloogia realisatsioon. Erinevad realisatsioonid realiseerivad hulka funktsionaalsust ja on omavahel ühendavad. Ühenduvus on saavutatud sellega, et CORBA tehnoloogia on spetsifitseeritud OMG *(Object Management Group)* poolt. CORBA tehnoloogia käsitleb mitmeid tarkvaraarendamise aspekte ja arvestab erinevate elu valdkondade nõudmisi – on loodud hulk lisateenuseid COS (*Common Object Service*), mis lihtsustavad rakenduse loomist. CORBA spetsifitseerimisel on arvestatud asjaoluga, et enamus rakendustest nõuavad kasutada objekte ja andmeid, mis on säilitavad näiteks andmebaasi abil ja on taastavad. Selle tulemusena on ilmunud CORBA säilitavate objektide lisateenuse PSS (*Persistent State Service*) spetsifikatsioon. Käesoleva töö eesmärgiks on uurida säilitatavate objektide kasutamist CORBA rakendustes.

Töö esimene osa on referatiivne. Esimene peatükk annab lühiülevaade CORBA tehnoloogiast ja kirjeldab selle tähtsamad osad ja kontseptsioone. Teine peatükk defineerib säilitavate objektide olemuse ja vajadust nende järgi. Kolmas peatükk kirjeldab OMG poolt spetsifitseeritud PSS teenust, selle arhitektuuri, andmemudeli, kasutamise võimalused jne. Neljas peatükis vaadeldakse teiste võimaluste kasutamist CORBA objektide säilitavuse saavutamiseks,

konkreetselt Java Persistence API. Viies peatükk annab ülevaadet transaktsioonide toimimisest hajusates süsteemides ja tutvustab CORBA OTS *(Object Transaction Service)* lisateenust, millele tugineb PSS implementatsioon.

Töö teine osa on pühendatud praktilisele poolele – töö raames loodud demorakendusele. Rakendus on realiseeritud kasutades nii PSS teenust, kui ka JPA võimalusi. Selle rakenduse varal on analüüsitud tähtsamad säilitavate objektide kasutamise aspektid mis tulevad esile just CORBA rakendustes. Läbiviidud testide tulemused abistavad analüüsi tegemist ja annavad parema ettekujutuse jõudluse kohta. Rakendus on lisatud magistritöö juurde.

1 Lühiülevaade CORBA tehnoloogiast

CORBA (*Common Object Request Broker Architecture*, *üldine objektipäringute vahendaja arhitektuur*) on hajussüsteemide loomise tehnoloogia, mille spetsifitseerimine toimub OMG (*Object Managment Group*) initsiatiivi all. CORBA tehnoloogia on loodud eesmärgiga ühenduda omavahel heterogeenseid hajussüsteeme, mis on realiseeritud erinevatel platvormidel ja erinevate programmeerimiskeelte abil. OMG koostas hulk spetsifikatsioone mis kirjeldavad CORBA tehnoloogiat. OMG spetsifikatsioonid kirjeldavad funktsionaalsust ning määravad realisatsioonide liideste komplekti. Liidesed on kirjeldatud IDL (*Interface Definition Language*) keele abil. IDL on OMG poolt spetsifitseeritud keel mis kirjeldab CORBA liidesed programmeerimiskeeltest sõltumatult. Lisaks on spetsifitseeritud ka IDL keele kujutamine (*mapping*) erinevatesse programmeerimiskeeltesse. IDL kujutamine on spetsifitseeritud järgmistele keeltele: C/C++, Java, Ada, COBOL, Lisp, PL/1, Python ja Smalltalk. Eksisteerivad ka mittestandardsed kujutamised Perl, Visual Basic, Tcl keelte jaoks. Lisaks on spetsifitseeritud Java liideste kujutamine IDL liidesteks. Range kujutamise spetsifikatsioon lubab luua automaatseid vahendeid IDL liideste kujutamiseks teistesse programmeerimiskeeltesse (nn. IDL kompilaatorid) [JH_MAG].

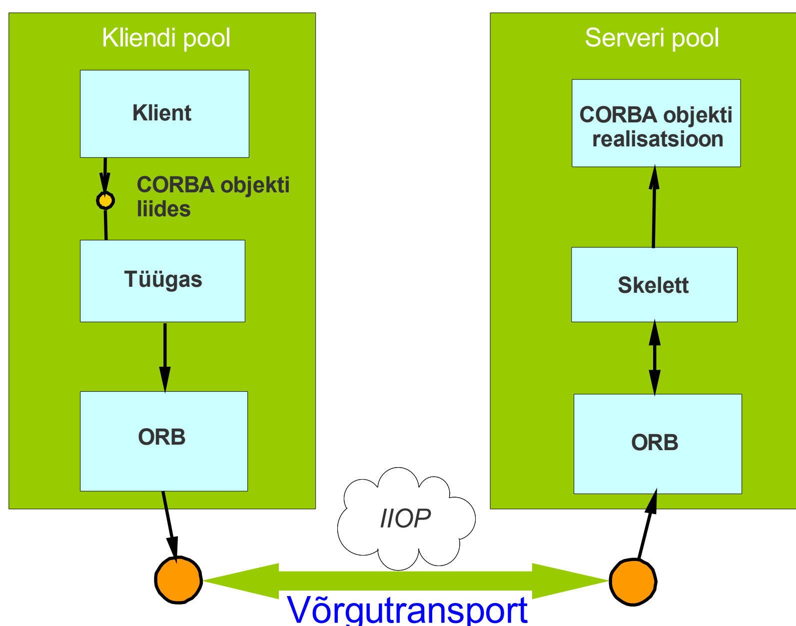
OMG spetsifikatsioonid katavad nii CORBA baasfunktsionaalsust (*Core*), kui ka rea lisateenuseid – COS (*Common Object Service*). OMG spetsifikatsioonidele on loodud CORBA realisatsioonid – vastavuses spetsifikatsioonidele implementeeritud tarkvaralahendused. OMG ise ei ole pakkunud enda poolt soovitusliku CORBA realisatsiooni (*Reference Implementation*) ja tegeleb ainult spetsifikatsioonide edasiarendamisega. On loodud palju CORBA realitsioone, mis erinevad kasutava spetsifikatsiooni versiooni, pakutavate lisateenuste, lisavõimaluste, mugavuse ja samuti ka jõudluse poolest [PUDER]. Realisatsioonid on implementeeritud erinevates programmeerimiskeeltes ja võimaldavad kasutada programmeerimiseks erinevaid keeli (enamasti aga C++ ja Java´t).

Samuti on spetsifitseeritud ka CORBA võrgusuhtluse protokoll – TCP/IP-l põhinev *Internet Inter-OMG Protocol, IIOP* [WIKI_CORBA].

1.1 Lühiülevaade CORBA arhitektuurist

CORBA tähtsaim osa on selle tuum (*Core*), mis realiseerib põhifunktsionaalsust. Iga CORBA realisatsioon peab implementeerima vähemalt tuuma spetsifikatsiooni. ORB (*Object Request Broker, objektipäringu vahendaja*) on keskne osa tuumast. ORB toetab lokaalsete ja

kaugobjektide koostoimet, realiseerides objektide koostoime mehhanismi, mis on sarnane RPC-ga (*Remote Procedure Call*). ORB ülesanneteks on objektide otsing, päringute saatmine ja vastuvõtmine. CORBA klientideks on olemid, mis teostavad teenusepäringuid serverile. Serveriks on olemid, mis pakkuvad teenuseid. Serveris asuvad IDL-ga spetsifitseeritud teenuste realisatsioonid. Teenuste pärimine ja kasutamine toimub objektiviide (*object reference*) abil. Objektiviidete haldamise funktsionaalsust servesis pakub objekti adapter (*Object Adapter*).



Joonis 1.1. CORBA kaugobjektide kasutamine [WIKI_CORBA]

Objekti adapteri abil toimub teenust pakkuvate objektide viidete vahendamine. Samuti adapteri abil toimuvad teenust pakkuvate objektide juhtimine ja haldamine. POA (*Portable Object Adapter, portatiivne objekti adapter*) – kasutuses olev ja OMG poolt spetsifitseeritud objektiadapter. POA võimaldab säilivate objektide loomist ning laiendab võimalusi objektide aktiveerimiseks. [JH_MAG]

CORBA spetsifikatsioonid toetavad hajussüsteemide vahetarkvara *(middleware)* loomist, CORBA on saanud standardiks heterogeensete hajussüsteemide loomiseks. Tänu põhi– ja lisafunktsionaalsuse valikule, saab CORBA-t kasutada erinevate suurustega ja keerukustega süsteemide loomiseks [JH_BAK].

1.2 IDL – Liideste defineerimise keel

OMG *Inteface Definition Language* – IDL määrab objektide tüübid, spetsifitseerides nende liidesed. Liides koosneb operatsioonide ja parameetrite kogust. Seega IDL kujutab ennast moodust, mille abil objektide realisatsioon annab klientidele teada mis operatsioonid on

kättesaadavad ja kuidas neid välja kutsuda. Kasutades IDL liideste kirjeldust, saab teisendada need liidesed konkreetse programmeerimiskeelte liidestekeks kasutades OMG kujutamise spetsifikatsioone. Teisendamine toimub automatiseeritud vahendite abil – IDL kompilaatoritega [JH_BAK].

CORBA objekti tüüp on rangelt määratud tema IDL liidesega. Liides on identifitseeritud tema nimega. IDL liidesed toetavad pärimist ja kõikide liideste baasliideseks on *CORBA::Object*. Kõik CORBA liidesed ja andmetüübid on kirjeldatavad IDL-ga.

IDL juures on vaja mainida, et CORBA rakenduse optimeerimine peab algama juba projekteerimise ajal. Kogu süsteemi jõudlus sõltub väga palju IDL-tüüpidest, mis antakse protseduuride parameetritena. Järgnev joonis illustreerib kulud kodeerimisele ja dekodeerimisele erinevatel andmetüüpidel:

Joonis 1.2.: Kodeerimise ja dekodeerimise kulud
[BOOK_CORBA]

Lisaks on tähtis ka see kas kutsutakse protseduuri mitu korda erinevatega parameetritega või kasutakse variandi, kus korraga parameetrina antakse terve massiiv andmetest. Reeglina viimane variant töötab palju paremini [BOOK_CORBA].

1.3 CORBA lisateenused

CORBA pakub võimaluse luua keerulisi tarkvaralahendusi erinevate valdkondade jaoks. Selleks, et see oleks võimalik, ei piisa sellest baasfunktsionaalsusest, mida pakub CORBA tuum. Arvestades hajussüsteemide loomise omapärad ja raskused, OMG spetsifitseeris hulga lisateenuseid mis on abiks tarkvaralahenduste loomise ajal. Funktsionaalselt sarnased võimalused on jagatud teenusteks. Teenuste kogum COS (*Common Object Service - objektide ühised teenused*) koosneb IDL liideste komplektist iga teenuse kohta, ning põhjalikust eesmärkide ja funktsionaalsuse kirjeldusest [JH_BK]. COS teenuste olemasolu konkreetses

CORBA realisatsioonis ei ole kohustuslik. Sellepärast erinevad CORBA realisatsioonid pakkuvad erineva hulka COS teenuseid – alates kõikide teenuste olemasolust ja lõpetades teenuste täieliku puudumisega. Tavaliselt erinevad ka realiseeritud spetsifikatsioonide versioonid, sest iga lisateenus on spetsifitseeritud eraldi sõltumata CORBA tuuma spetsifikatsiooni versioonist. Teenuste olemasolu tavaliselt ei sõltu sellest, kas konkreetne realisatsioon on loodud kommertsbaasil või on vabavara – lisatakse neid COS teenuseid, mis on olulised selles realisatsioonis (arvestades kasutamise mugavust ja jõudlust). Mõned OMG poolt spetsifitseeritud teenused ei olnud kunagi implementeeritud (näiteks *POS – Persistent Object Service*), põhiliseks takistuseks on olnud keerukas teenuse arhitektuur ja ennustatav nõrk jõudlus [JH_BK], [BOOK_CORBA].

Olulisemad COS teenused

Antakse tähtsamate ja kõige rohkem kasutatavate teenuste lühikirjeldused. See paragrahv suuresti baseerub allikal [JH_BK] ning kasutakse viitamata tekstilõike.

Nimeteenus (*Naming*) – see teenus võimaldab siduda (*bind*) objekti ja talle antud nime. Nimeteenus on üks tähtsamatest COS teenustest ja on olemas peaaegu kõikides CORBA realisatsioonides. Nimeteenus on kättesaadav nii kliendi, kui ka serveri pooltel ning võimaldab kaugobjekti kliendil saada kaugobjekti objektiviide nime järgi.

Objektivahetuse teenus (*Object Trader*) on mõeldud teenust pakkuvate objektide isenditevahetuseks ning nende objektide leidmiseks atribuutide järgi. Teenus võimaldab publitseerida objekte (*export*) ning kasutada neid objekte (*import*). Objektivahetuse teenus defineerib teenuse tüüpi, mis koosneb identifikaatorist, teenuse kirjeldusest ning atribuutide loetelust. Klientide rakendused saavad nii eksportida olemasolevad objektid, kui ka importida neid (pärides atribuutide järgi).

Transaktsiooni teenus (*Object Transaction Service*) võimaldab transaktsioone CORBA hajussüsteemides. Transaktsioonide teenus võimaldab täita operatsioone atomaarselt ja isoleeritult. Transaktsiooni teenuse kasutamisega peab arvestama nii rakenduse kliendi poolel kui ka serveri poolel. Transaktsioone saab kasutada nii ilmutatud kujul, kui ka ilmutamata kujul.

Turvalisuse teenus (*Security Sevice*) käsitleb turvalisuse aspekte. Turvalisuse teenus võimaldab turvalist ühendust, autentimist ja identifitseerimist. Samuti selle teenuse abil toimub turvalisuse administreerimine. Suhtlemiseks defineerib turvalisuse teenus uue protokolli SECIOP (*Secure*

Inter-ORB Protocol) GIOP ja IIOP protokollide vahel turvalise suhtluse teostamiseks.

Sündmuseteenus (*Event Service*) võimaldab objektivahelist suhtlust vahendaja toimel. Defineeritakse sündmuste varustajad (*suppliers*) ja tarbijad (*consumers*). Varustajad toodavad sündmuse (*event*) andmeid ning tarbijad kasutavad sündmuste andmeid.

Teatamisteenus (*Notification Service*) on sündmuseteenuse edasiarendus, on lisandunud funktsionaalsust sündmuste käsitlemiseks, ning juhtumeid on võimalik struktureerida, järjestada ja transleerida. Teatamisteenuse puhul on lubatud varustajate ja tarbijate vahelised keerukamad struktuurid.

Säilitava oleku teenuse (*Persistent State Service*) abil saab hoida objektide oleku füüsilises andmehoidlas. See teenus asendas Säiliva objekti teenust (*Persistent Object Service*) , kuna viimane osutus liiga keerukaks nii programmeerijate, kui ka CORBA realisatsioonide loojate jaoks.

Lisateenused erinevates CORBA realisatsioonides.

Nagu oli juba mainitud, erinevad CORBA realisatsioonid pakuvad erineva hulga teenuseid. Reeglina saab eeldada, et kindlasti on pakutud nimeteenus, kuna sellel põhineb baasfunktsionaalsus – objektiviidete kättesaamine nime järgi. Lisaks enamustes realisatsioonides on olemas sündmuseteenus ning teatamisteenus. Teiste teenuste toetus ei ole ühtlane [PUDER]. Tavaliselt realisatsioon pakub kas palju või vähe lisateenuseid. Kui CORBA rakenduses on planeeritud kasutada mingit teenust, siis kindlasti tuleb kontrollida konkreetse realisatsiooni dokumentatsioonis kas antud teenus on olemas ja kui on, siis mis spetsifikatsiooni versioonile see vastab. Mõnede teenuste olemasolu ei ole kriitiline ja paljud CORBA realisatsioonide tootjad ei kavatsegi realiseerida neid lisateenuseid, mõnikord pakkudes asendusena sarnase funktsionaalsusega spetsifitseerimata API-d. Väljaspool OMG spetsifikatsiooni realiseeritud funktsionaalsus on reeglina sõltuv platvormist ja programmeerimiskeelest, seega ei ole laialt kasutatav (erinevalt COS teenustest).

Ülevaade COS teenuste olemasolust erinevates CORBA realisatsioonides on esitatud *Lisas 3*. Tuleb mainida, et antud ressurss ei ole juba mõnda aega uuendatud, seega hetkeandmed võivad erineda.

1.4 CORBA Component Model (CCM)

CORBA Component Model – CORBA Komponendimudel, on defineeritud CORBA 3

spetsifikatsiooni osana. CCM kirjeldab standardset raamistikku komponentide defineerimiseks ja paigutamiseks. Komponendid töötavad konteineri sees, mis pakub hulga COS teenuseid (teatamine, autentimine, objektide säilitavus, transaktsioonid) [WIKI_CORBA]. Komponentide funktsionaalsus on kättesaadav määratud liideste kaudu (*ports*). CCM spetsifikatsioon võimaldab ühendada omavahel CORBA komponente ja EJB (*Enterprice Java Bans*) komponente [CCM_SPEC].

2 Säilitavad objektid CORBA rakenduses

2.1 Objektide säilitamise vajadus objektorienteeritud süsteemis ja andmebaasidega integreerimine

Relatsiooniliste andmebaaside teoreetiliseks aluseks on relatsioonilised arvutused. Kõik andmebaasi andmed hoitakse tabelites. Iga element kujutab endast rea mingis tabelis, iga rida võib koosneda mitmest veerust, iga veerg omab kindla andmetüübi. Tabeli rida on määratav võtme abil, võtmete abil ehitatakse ka seosed tabelite vahel. Suhtlemine RDBMS (*Relational Database Management System*) serveri ja kliendi vahel toimib SQL (*Structured Query Language*) keele abil. Kliendid saadavad päringud serverile andmete pärimiseks või muutmiseks, tulemusena on tavaliselt ridade hulk. Relatsioonilised andmebaasid sobivad väga hästi suurte andmemahtude hoidmiseks ja võimaldavad neid efektiivselt pärida ja muuta SQL abil. Objektorienteeritud keelte objektide hoidmist saab realiseerida, määrates objekti väljade sobiva tüübiga veerg ja realiseerides seosed tabeltena [WIKI_REL].

On olemas ka spetsiaalsed objektorienteeritud andmebaasid. Objektorienteeritud andmebaasid pakuvad mugavamaid vahendeid objektide hoidmiseks – programmeerimiskeeles defineeritud tüüpide otsese kasutamise võimalused, komplitseeritud puhverdamine (arvestab objektide seostega) jne. [BOOK_CORBA].

Kuigi objektide keeruliste seoste realiseerimine relatsioonilises andmebaasis võib olla liiga kallis jõudluse vaatenurgast (selles relatsioonilised andmebaasid jäävad alla objektorienteeritud andmebaasidele), tänapäeval säilitavate objektide hoidmiseks kasutatakse valdavalt relatsioonilise andmebaase. See võib olla seletatav sellega, et relatsioonilise andmebaase saab korraga kasutada mitme erineva kliendi poolt (objektorienteeritud programm, C programm, PHP skript). Lisaks relatsiooniliste andmebaaside tehnoloogia on piisavalt küps ja IT-ettevõtted juba tegid suured majanduslikud kulud RDBMS tarkvara ostmiseks [REL_OO].

2.2 Objektide hoidmise vahendid

On olemas mitu võimalust realiseerida objektide hoidmist relatsioonilises andmebaasis. Reeglina tuleb kasutada vahekihti, mis peidab andmebaasi tabelite ja programmeerimiskeele struktuuride erinevust. Vajalikud omadused, mis peaks olema kõikidel sellistel vahenditel, on [BOOK_CORBA]:

- Objektorienteeritud lähenemise toetus

Inkapsulatsiooni, objekti identsuse, suhete, pärimise ja polümorfismi toetus.

- Põhilise elutsükli operatsioonide toetus

  Peab olema võimalus teostada objektide loomist, muutmist ja eemaldamist. Relatsiooniliste andmebaaside terminitega seda nimetatakse CRUD-operatsioonideks (*Create/Read/Update/Delete – Loomine-Lugemine-Muutmine-Kustutamine*).

- Pärimine ja tulemuste kättesaamine

  Peab olema võimalus pärida objekte ja töödelda tulemust. Tavaliselt selleks kasutatakse RDBMS keelt SQL-i või sarnased vahendid.

- Tegevused mitme objektida

  Võimalus teostada operatsioone mitmete objektidega korraga.

- Paralleeleelse juurdepääsu võimalus

  Peab olema võimalus töötada objektiga paralleelselt. Tavaliselt realiseeritud kasutades transaktsioone või olemite versioonide jälgimist.

- Keele kujutamised (*mappings*)

  Erinevate keeltesse kujutamise võimalus (Java, C++ jne).

Tarkvarasüsteemide arendamisel võib tekkida vajadus objekt-relatsioonilise seostamise järgi erinevatel juhtudel [M_NOGES]:

- Ülevalt alla

  Alguses on olemas objektimudel ning vastavalt sellele on vaja luua andmebaasi skeem ja vastavad seosed objektide ja andmebaasi tabelite vahel.

- Alt üles

  On olemas relatsiooniline andmebaas ja selle järgi on vaja luua objektimudel. Tavaliselt selline olukord tekib siis, kui on vaja täiendada olemasoleva infosüsteemi kus on juba olemas andmed.

- Keskelt välja

  Alguses on olemas objektimudeli metaandmete kirjeldus, näiteks disainidokument. Vastavalt sellele on vaja luua nii programmeerimiskeele objektid, kui ka andmebaasi struktuuri ning seostada neid.

- Väljast keskele

  On olemas andmebaas ja objektimudel. On vaja tekitada nendevahelised seosed, näiteks kui on vaja ühenduda olemasoleva infosüsteemi teise infosüsteemi andmebaasiga.

CORBA raames spetsifitseeritud PSS teenus võimaldab kasutada “Ülevalt alla” mudelit ja eeldab just sellist lähenemist [PSS_SPEC].

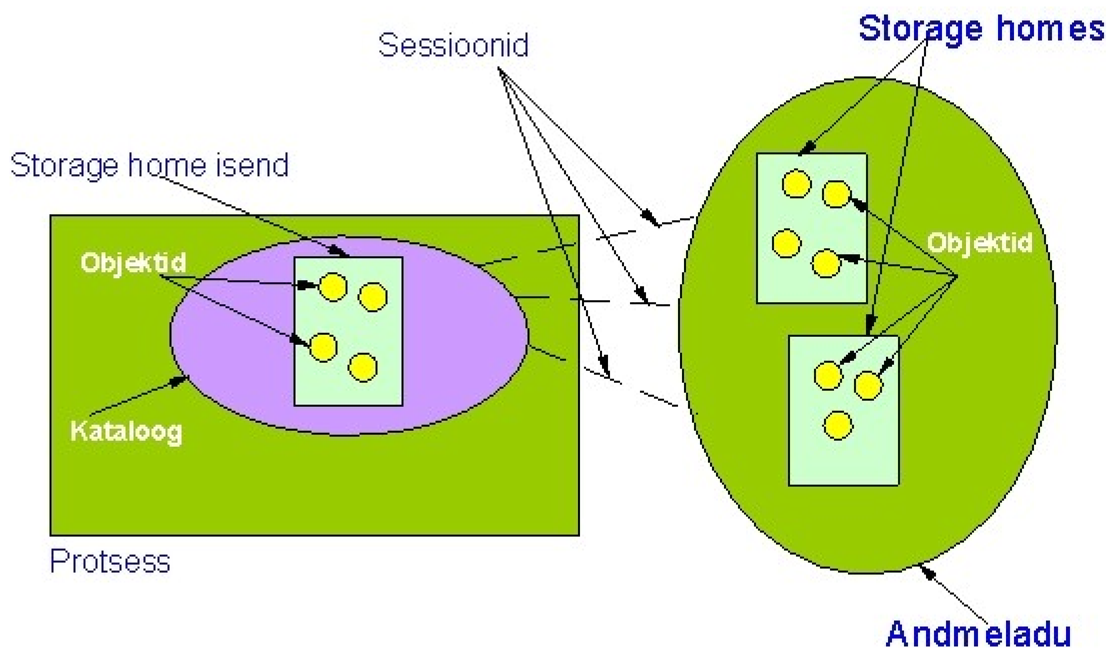
3 CORBA Persistent State Service

Pikaajaline objektide säilitamine on tähtis peaaegu igas infosüsteemis (kui tegemist ei ole rakendusega, mis ei säili oma oleku). CORBA korral säilitavate objektide kasutamise semantika peab olema sõltumatu konkreetsest programmeerimiskeelest ja platvormist.

CORBA tuuma funktsionaalsus küll lubab kasutada püsivaid POA objektiviiteid, kuid mitte püsiva olekuga objekte. Arvestades sellega, aastal 1994 OMG spetsifitseeris POS (*Persistent Object Sevice*) teenust, mis pidi lubama püsiva olekuga objekte. Kuid POS-i spetsifikatsioon ei sobinud hästi reaalmaailma. Kõigepealt teenuses ei õnnestunud hästi eraldada objekti oleku ja objekti esitamist andmehoidlas. Teenuse kasutajad nägid mis on tegelikult objekti taga ja lisaks sellele pidid ilmutatud kujul salvestama ja laadima objektide olekud. POS oli kaugel transparentse mehhanismist. Lisaks POS-i arhitektuur oli liiga keeruline ja raskesti realiseeritav. Sellepärast ei olnud ühtegi CORBA kommertsrealisatsiooni, kus oleks olemas POS-i spetsifikatsioonile vastav teenus [BOOK_CORBA].

Arvestades nendega probleemidega OMG kirjutas täielikult üle säilitavate objektide teenuse spetsifikatsiooni. Selle tulemusena aastal 1997 ilmus esimene versioon uuest teenusest – PSS (*Persistent State Service*). See spetsifikatsioon paremini vastas nõuete ja selle arhitektuur ei olnud nii keeruline. On teada vähemalt kümme CORBA realisatsiooni, kus see teenus on realiseeritud [*Lisa 3*].

3.1 PSS andmemudel



Joonis 3.1: PSS andmemudel

Säilitava olekuka objektide andmehoidla koosneb kolme tüüpi objektidest. Üldisem ühik on Andmeladu (*Datastore*). Selle esitusviis kliendirakenduses on Kataloog (*Catalog*). Andmeladu sees on Storage Home objektid, mis kujutavad endast konkreetse tüüpi objektide hoidlaid. Storage Home objekt hoiab mitu säilitava objekti. Iga säilitav objekt omab identifikaatori – *pid-i* (on olemas ka kohalik, konkreetse Storage Home ulatuses kehtiv, identifikaator *short-pid*). Andmeladu saab võrrelda andmebaasiga, Storage Home – tabeliga selles andmebaasis ning säilitava objekti – kirjega tabelis [CHUPP_PSS].

Rakenduses kasutatavate säilitava olekuga objektide defineerimiseks kasutatakse spetsiaalset keelt PSDL (*Persistent State Definition Language*), mis on IDL keele ülemhulk. Säilitavad objektid on kirjeldatavad abstraktsete tüüpide(S*toragetype*) abil. Objekti säilitavad atribuudid nimetatakse Oleku liikmeteks(S*tate members)*, lisaks on võimalik defineerida säilitava objekti operatsioonid (*Storagetype operations*). Storage Home on defineeritav abstraktse *Storage Home* abil, see defineerib ka võtme, mis on unikaalne selle Storage Home ulatuses (relatsioonilise andmebaasi tabeli primaarse võtme analoog). Võti võib olla ka kombineeritud. PSDL faili sisu näitena võib anda selles töös kasutatava CORBA realisatsiooni – OpenORB-i, näide.

```
abstract storagetype Person
```

```
{
    state string name; //  State members definition
    state string address;
    state long old;
    void print(); // Storagetype operations
    void birthday();
};
// PersonHome  is a simple container for persistent data.
abstract storagehome PersonHome of Person
{
    key name;
    factory create( name, address, old ); // creation operation
};
// Storage type provides an implementation for the abstract storage type 'Person'.
storagetype PersonBase implements Person
{
};
// This storage home implements the abstract storage home
// 'PersonHome' and manages a storage type 'PersonBase'.
storagehome PersonHomeBase of PersonBase implements PersonHome
{
};
catalog PersonCatalog {
      provides PersonHome aPersonHome;
}
```

Rakenduses Andmehoidlale vastab Kataloog, Storage Home-le vastab Storage Home isend (*instance*), Säilitavale objektile vastab Säilitava objekti isend (*Storage Object instance*). Kui säilitavate objektide isendid on ühendatud vastavate objektidega andmehoidlas, siis neid nimetatakse *Inkarnatsioonideks*.

3.2 PSDL keele kompilaator

PSDL keel oli loodud selleks, et PSS objekte saaks defineerida abstraktselt ja eraldi failis. CORBA PSS realisatsioonid pakuvad PSDL keele kompilaatori. Kompilaatori sisendiks on PSDL keel fail (failid) laiendusega *.psdl* . PSDL keel on samade leksikaalsete reeglitega nagu

IDL keel, PSDL keele grammatika laiendab IDL grammatikat. PSDL failis saab kasutada suvalisi IDL konstruktsioone (*struct, sequence, union*) ning lokaalselt ka PSDL keele konstruktsioone (näiteks jada säilitatavate objektide viidetest), pluss samas failis defineeritud PSS objekte *(storagetype* jne*)*. PSDL kompilaatori abil genereeritakse kujutamised konkreetsetesse programmeerimiskeeltesse (näiteks *.cpp* või *.java* faile). OMG on spetsifitseeritud PSDL kujutamised Java ja C++ keelte jaoks. See oli tehtud eesmärkiga, et saaks kasutada erinevate PSS teenuse realisatsioone (realiseeritud C++ keeles või Javas) ilma PSS kliendi koodi uuesti kompileerimata [PSS_SPEC].

3.3 PSS realisatsiooni arhitektuur

Keskne roll on PSS Ühendajal (*PSS Connector*), tema ühendab säilitavad objektid nende füüsilistega representatsioonidega. On olemas näiteks Faili Ühendaja – File Connector (andmeid hoitakse failis), Andmebaasi Ühendaja - Database Connector (andmeid hoitakse andmebaasis). Lisaks on võimalik teiste Ühendajate olemasolu. Selleks, et saaks dünaamiliselt kasutada ja pärida erinevaid Ühendajad, PSS teenus pakub kasutamiseks Ühendajate Registri (*Connector Registry*). Registris on registreeritud kõik olemasolevad PSS Ühendajad.

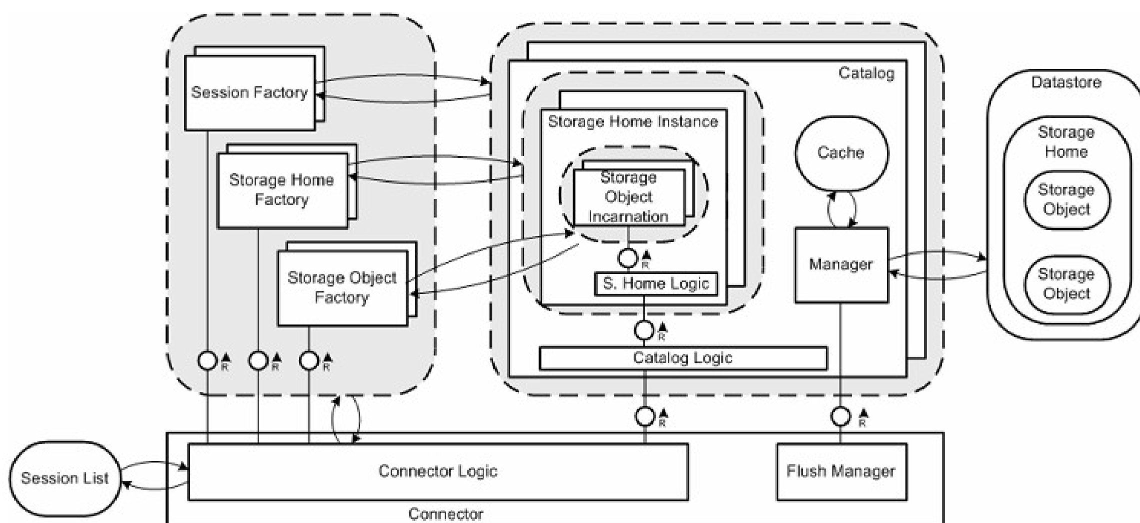
Samuti tähtis mõiste on Sessioon (*Session*) - kasutaja seanss Andmehoidlaga [SCHUPP_PSS].

Ühendaja

Ühendaja abil toimuvad uute PSS isendite tekitamine, selleks registreeritakse Varikud (*Factiries*) iga Storage Home, Kataloogi ja Säilitava Objekti jaoks. Kuna need objektid on erinevad (nende tüüp on defineeritud PSDL failides), siis on vajalikud ka erinevad vabrikud. Iga vabrik peab olema eelnevalt registreeritud, et seda saaks rakenduses kasutada. PSS Ühendaja "publitseerib" temale kuuluvate kataloogide nimekiri [SCHUPP_PSS].

OMG spetsifikatsioon määrab ainult üldist PSS käitumist, defineerib liidesed ja PSDL keele kujutamist [PSS_SPEC]. Sellepärast PSS realisatsioonid võivad erineda oma sisemise arhitektuuri ja lisafunktsionaalsuse poolest (komplitseeritud puhverdamine, erinevate Ühendajate olemasolu jne.).

Edasi on toodud näitena ühe konkreetse CORBA realisatsiooni – OpenORB-i, PSS Ühendaja realisatsiooni kirjeldatav loonis [SCHUPP_PSS].



Joonis 3.2: OpenORB Ühendaja realisatsioon [SCHUPP_PSS]

Realisatsioonis on olemas elemendid, mis ei ole OMG poolt täpselt spetsifitseeritud, kuid on vajalikud PSS efektiivseks toimimiseks. Nende hulka kuuluvad Flush Manager (), Kataloogi äriloogika, üldine Haldur (*Manager*), Ühendaja äriloogika. Haldur jälgib andmete vastavust säilitavate objektide (*Incarnations*) ja füüsiliste kirjete vahel. Kõik väärtuste muudatused, mis toimuvad isenditega opereerimise käigus, võetakse arvesse ning pannakse vahemällu. Mõne aja pärast mudatused salvestatakse füüsiliselt Andmehoidlasse. Puhverdamine märgatavalt parandab jõudlust. Ajavahemikke, mille järel toimub sünkroniseerimine, määrab Flush Manager. Samamoodi Halduri abil toimub isendite väärtuste värskendamine siis, kui toimus väline andmete muutmine Andmehoidlas [SCHUPP_PSS].

Sessioonide haldus

PSS spetsifikatsioon defineerib kahte võimalust sessioonide haldamiseks: ilmutatud kujul ja ilmutamata kujul. Programmi arendaja saab manipuleerida sessioonidega ilmutatud kujul, vastasel juhul sessioonide haldamine toimub automaatselt. On olemas kaks tüüpi sessioone: Baassessioon (*Basic Session*) ja Transaktsiooniline sessioon (*Transactional Session*) – neid saab kasutada ilmutatud kujul. Lisaks on olemas veel üks tüüp – sessioon sessioone sessioonide varust. Baassessioon sobib ainult siis, kui Andmehoidlaga töötab ainult üks kasutaja ja selle kliendi äriloogika arvestab paralleelse juurdepääsuga. Reaalsete rakenduste jaoks paremini sobib Transaktsioone sessioon. Selle tüüpi sessioon tagab, et konkureeritavad operatsioonid täidetakse isoleeritult arvestades ACID (A*tomicity, Consistency, Isolation, Durability*) printsiipe

[PSS_SPEC]. PSS Transaktsioonilise sessiooni realisatsioon baseerub OMG Transaction Service (OTS) peal, sellepärast Transaktsioonilise sessiooni kasutamiseks OTS peab olema aktiivne. OTS teenusest räägitakse järgmistes jaotustes. Seega nende sessioonide valikut määrav kriteerium on parallelne juurdepääs:

- Baassessioon – tavaline sessioon, ei paku paralleeleelse juurdepääsu turvalisust
- Transaktsioone sessioon – transaktsiooniline sessioon, mis lubab töötada säilitavate objektidega paralleelselt (näiteks mitmekasutaja infosüsteemides)

Sessioonide varu haldab sessioone taustal ja täiesti automaatselt (sessiooni loomine, ressursside registreerimine), sessioonide varu initsialiseerimisel määratakse, mis tüüpe sessioone saab sellest varust võtta (Baassessioon või Transaktsiooniline). Kasutamisel varust küsitakse jooksev sessioon ja seda kasutatakse nagu tavaliselt [SCHUPP_PSS].

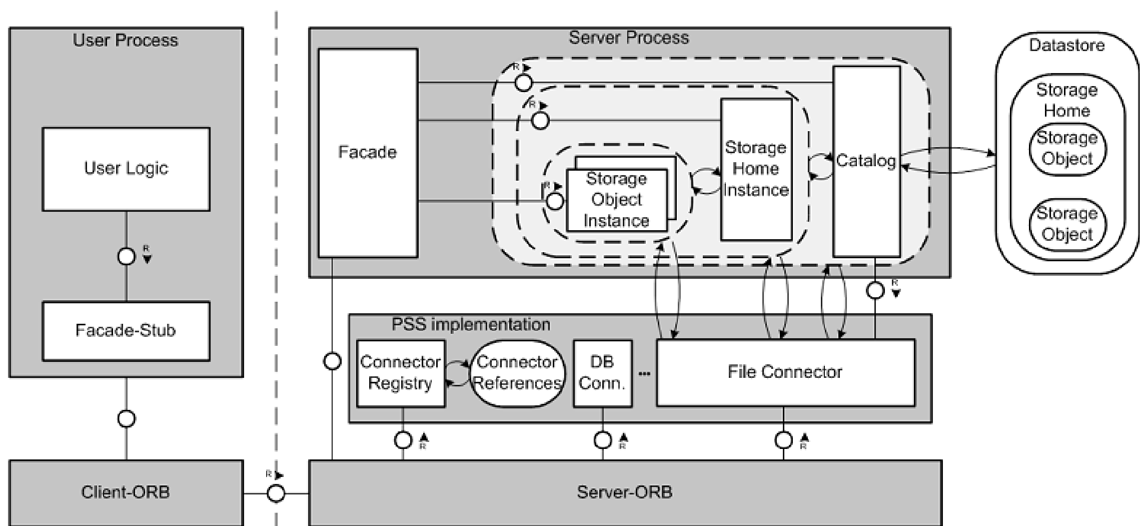
3.4 PSS kasutamise võimalused ja tüüpilised stsenaariumid

PSS-i poolt pakutavad võimalusi saab kasutada nii serveri, kui ka kliendi poolel – selleks on vaja vastavalt konfigureerida kasutatava CORBA realisatsiooni. Reegina PSS võimalusi kasutab serverirakendus. Transaktsioonilise sessiooni kasutamise puhul PSS-i kasutaja peab muretsema transaktsiooni alustamisest ning lõpetamisest (luues transaktsioone kas ilmutatud või ilmutamata kujul).

PSS kasutamisel serveri poolel tihti leiab kasutust nn. Fassaad-teener (*Facade servant*) , mis peidab kliendilt Storage Home ja Andmeladu objektide olemasolu. Säilitava olekuga objektide kasutamine kliendi poolel sel juhul ei erine tavaliste CORBA objektide kasutamisest [OPEN_ORB_DOC].

Server initsialiseerib PSS-i ja selle abil saab juurdepääsu PSS registrile ning säilitavate objektidele. Esimese sammuna päritakse viide Ühendajale (näiteks Andmebaasi Ühendajale). Ühendaja abil registreeritakse objektide vabrikut (säilitava objekti implementatsiooni) ning Storage Home vabrikut. Kui kasutatakse transaktsioonilisi sessioone, on soovitatav initsialiseerida sessioonide vabrikut, mitte initsialiseerida neid ilmutatult [PSS_SPEC]. Teise sammuna toimub sessiooni initsialiseerimine – kui kasutatakse tavalist sessiooni, siis initsialiseerimine toimub ilmutatud kujul, vastasel juhul sessioon võetakse sessioonide varust. Viimase sammuna päritakse Home objekti, mille abil saab otseselt luua ja muuta säilitavaid objekte. Viide Home objektidele tagastab sessiooni objekt – meetod *find_storage_home(storageHome_id)*. Joonisel on skemaatiliselt näidatud PSS kasutamise

stsenaarium.



Joonis 3. PSS kasutamise tüüpiline stsenaarium [SCHUPP_PSS]

Klient kutsub välja Fassaad-objekti (kaugobjekt) protseduure ja see omakorda haldab säilitavate objektide olekut. Protseduuride parameetrid antakse kas lihttüüpide abil või liitobjektina (IDL keele *struct*). Selle põhjuseks on see, et sest säilitavad objektid on defineeritud eraldi .PSDL failis ning neid ei saa kasutada parameetrina liideste defineerimisel teistes IDL-failides. Säilitava objekti realisatsioon ei laienda liidest *org.omg.CORBA.portable.IDLEntity,* seega ta ei ole tavaline CORBA objekt. PSS kasutamist on vaja ette planeerida disaini ajal, tavalist CORBA objekti ei saa programmselt muuta säilitavaks objektiks [PSS_SPEC].

PSS kasutamise näitena võib anda väikest koodilõiku, mis on võetud konkreetse OpenORB-i näidete hulgast. On näidatud tavalise andmebaasisessiooni kasutamist. Sessiooni konfigureerimine toimub parameetrite abil.

```
// PSS initsialiseerimine
org.omg.CORBA.Object obj = orb.resolve_initial_references( "PSS" );
org.omg.CosPersistentState.ConnectorRegistry registry = org.omg.CosPersistentState.ConnectorRegistryHelper.narrow( obj );
// Ühendaja pärimine
org.omg.CosPersistentState.Connector connector = registry.find_connector( "org.openorb.pss.Database" );
// Register the storage type factory
connector.register_storage_object_factory("PSDL:org/openorb/pss/examples/database/basic/first/PersonBase:1.0",
        Thread.currentThread().getContextClassLoader().loadClass("org.openorb.pss.exa
```

```
mples.database.basic.first.PersonBaseImpl" ) );
// Register the storage home factory
connector.register_storage_home_factory("PSDL:org/openorb/pss/examples/database/basic/
first/PersonHomeBase:1.0",
		Thread.currentThread().getContextClassLoader().loadClass("org.openorb.pss.exa
mples.database.basic.first.PersonHomeBase" ) );
...
// parameetritena antakse andmebaasi URL ja teised väärtused...
org.omg.CosPersistentState.Session mySession =
connector.create_basic_session(org.omg.CosPersistentState.READ_WRITE.value,"",
parameters );
PersonHomeBase home = ( PersonHome )
mySession.find_storage_home("PSDL:org/openorb/pss/examples/database/basic/first/Perso
nHomeBase:1.0" );
// nüüd saab töötada säilitavate obkektidega home abil
```

Kui on olemas Home objekt (ja selle kaudu säilitavatele objektidele), siis saab töötada säilitavate ojektidega. Home objekti tähtsamad meetodid on

- *create()* - uue säilitava objekti loomine
- *find_by_key(key)* - objekti pärimine võtme järgi
- *find_by_pid(pid)* (*find_by_short_pid(spid)*) – objekti pärimine unikaalse identifikaatori järgi

Säilitavate objektide realisatsioonid realiseerivad PSDL failis kirjeldatud meetodid, nende objektide muutujate seosed andmehoidlaga ei ole kasutamisel kajastatud. Muutujate väärtuste muutmised toimuvad vastavate *muutujaNimi(OmistatavVäärtus)* meetodite abil, väärtused saadakse vastavate meetodite *muutujaNimi()* abil. Väärtuste sünkroniseerimine andmehoidlaga toimub automaatselt ning üldiselt ei vaja programmeerija sekkumist. Püsiva olekuga objektide kasutamine on samasugune sõltumata kasutatavast Ühendajast.

Kui PSS-i soovitakse kasutada kliendi poolel, siis initsialiseerimise sammud on samad, ainult tuleb arvestada sellega, et kliendil peab olema algviide PSS teenusele. Home objektide kasutamine on samuti identne.

3.5 Päringute API

Kahjuks OMG PSS spetsifikatsioonis on mainimata sobiva päringukeele väljaarendamise võimalused [PSS_SPEC]. Objekte saab küll pärida võtme (võti saab olla ka kombineeritud) või

PID järgi, kuid tihti need võimalused ei lahenda kõike reaalelu probleeme. Suurimaks puuduseks on ilmselt see, et puudub võimalus pärida soobivalt filtreeritud olemite hulka nagu seda tehakse andmebaasides SQL keelt kasutades ja efektiivselt piirates tulemuste hulka mingi kriteeriumi järgi. See probleem on lahendatav teiste CORBA vahenditega, saab näiteks kasutada *Trading Service (Objektivahetuse teenus)* teenust ja registreerida iga säilitav objekt seal. Objektivahetuse teenus võimaldab kasutada SQL-sarnast keelt objektide pärimiseks. Samas selle teenuse kasutamine koos PSS teenusega teeb programmerimist keerulisemaks ja kindlasti negatiivselt mõjub üldisele jõudlusele, kuna päringud tehakse mitte andmebaasile kus säilitavad andmed asuvad, vaid eraldiseisvale väärtuste hoidlale mis ei tea midagi andmebaasist ning ei oska teostada otsingut nii optimaalselt kui seda teevad relatsioonilised andmebaasid. Lisaks tuleb arvestada sellega, et kõik väärtused, mille järgi soovitakse teha otsingut, tuleb sisuliselt kopeerida Objektivatuse teenuse repositooriumisse (või jooksvalt pärida). See on aga ressursside raiskamine.

Õnneks on olemas mõned CORBA realisatsioonid, kus taoline päringukeel on olemas. Näitena võib tuua IONA Technologies Orbix realisatsiooni Query API (OMG poolt spetsifitseerimata realisatsioonispetsiifiline funktsionaalsus). Orbix Query API kasutamine on sarnane *JDBC* kasutamisega. [IONA_PRGUIDE]

Näitena on toodud C++ koodilõik IONA Orbix vabalt kättesaadavast dokumentatsioonist [IONA_PRGUIDE].

```
IT_PSS::Statement_var statement =
association.get_session_nc()->it_create_statement();
  IT_PSS::ResultSet_var result_set = statement->execute_query(
    "select ref(h) from PSDL:BankDemoStore/Bank:1.0 h");
  BankDemoStore::AccountBaseRef     account_ref;
  CORBA::Any_var                    ref_as_any;
  while (result_set->next())
  {
    ref_as_any = result_set->get(1);
    CORBA::Boolean ok = (ref_as_any >>= account_ref);
    cout << "  " << account_ref->account_id()  << ", $" << account_ref->balance() << endl;
  }
  result_set->close();
```

Päringukeel tunduvalt lihtsustab PSS-i kasutamist ja ei lisa suurt jõudluse vähendamist. Põhjusteks, miks päringukeel ei olnud spetsifitseeritud OMG poolt, võivad olla erinevad, näiteks soov saavutada abstraktsust või keekukus päringukeele implementeerimisel erinevate Ühendajate jaoks. Kuid suure tõenäosusega saab väita, et päringukeele spetsifitseerimine ei olnud kõikide OMG liikmete huvides. OMG liikmeteks on ka suuremad CORBA realisatsioonide tootjad, kes ei ole huvitatud selles, et kõik vajalikud lisavõimalused oleks spetsifitseeritud. See teeb olemasoleva CORBA rakenduse migreerimist ühest CORBA platvormist teisele (näiteks vabavaralisele) eriti keeruliseks, mis vastab suurtootjate huvidele [CORBA_TRBL].

3.6 PSS Transparent Persistence

OMG PSS spetsifikatsioonis on mainitud transparentse objektide säilivuse võimalus, ehk võimalus defineerida säilitavad objektid mitte PSDL abil, vaid kasutatava programmeerimiskeele vahendite abil. Transparentse säilivuse mehhanism võimaldab teha säilivaks iga Java või C++ klassi (mõnede kitsendustega). Transparentne säilivus annaks võimalust kasutada objektide väljad otseselt, ilma standardsete (ja kohustuslikke) *muutujaNimi()* ja *muutujaNimi(OmistatavVäärtus)* meetodite kasutamata. OMG pakkus variandid, kuidas oleks võimalik realiseerida transparentsust, kuid see on rohkem soovitus ning ei ole spetsifitseeritud. Need soovitused on sellised [PSS_SPEC]:

Java keele jaoks:

- Kasutada Java failide preprotsessori, et lisada koodi mis värskendas väljade väärtusi andmehoidlast enne igat lugemist. Samas lisada koodi, mis sünkroniseeriks väärtusi andmehoidlas siis, kui säilitavate objektide väärtusi muudetakse programmi töötamise ajal. Tulemusena on modifitseeritud *.java* failid.
- Kasutada spetsiaalset Java kompilaatori, mis teeks need muudatused ilma *.java* failide tekitamiseta.
- Kasutada Java failide postprotsessori, mis teeks sama tööd baitkoodi tasemel (lähtekoodi kasutamata).
- Kasutada modifitseeritud Java virtuaalmasinat, mis teostaks objektide lugemist ja kirjutamist modifitseeritud kujul (kasutades andmehoidla väärtusi)

C++ keele jaoks:

- Kasutada C++ keele versiooni, mis vastaks *Object Data Management Group* (ODMG) C++ keele standardile. Selles standardis on spetsifitseeritud *smart pointer-i* (nutikas viide) kasutamise võimalused, mis võimaldavad jälgida objektide olekut ja sünkroniseerida neid automaatselt.

Transparentne säilitavus ei ole PSS spetsifikatsiooni mandatoorne osa, selle kasutamise võimalust on reeglina olemas ainult nendes CORBA realisatsioonides, mis realiseerivad CORBA Component Model spetsifikatsiooni (CORBA 3). Põhilisteks põhjusteks on realiseerimise keerukus ning üldise standardi puudumine. Samas on olemas palju Java ja C++ keelte raamistikuid, mis realiseerivad säilitatavuse võimalusi sarnasel moel (Java Persistance API , JDO, JPOX jne), kuid mitte CORBA tehnoloogia raames.

4 Teised O-R kujutamise raamistikud CORBA rakendustes

PSS, nagu paljud teised teenused, ei kuulu ORB põhifunktsionaalsuse juurde ja on spetsifitseeritud lisateenusena. See on põhjendatud, sest mitte iga CORBA rakendus vajab säilitavate objektide kasutamist ning teenuse realiseerimine ei ole lihtne. Paljud realisatsioonid on valdkonnaspetsiifilised, PSS lisateenuse olemasolu on pigem erand kui reegel ning tavaliselt on olemas nendes realisatsioonides, kus on juba realiseeritud OTS (*Object Transaction Service*) lisateenus [PUDER].

Samas eksisteerib palju võimalusi kasutada CORBA rakendustes üldisemad meetodeid (näiteks JDBC, ODBC jne.) ja raamistikke objektide oleku säilitamiseks. Mõned vahendid sobivad paremini, teised halvemini (POA eripärad, hajusad ja heterogeensed arhitektuurid).

Mõnikord puuduseks on see, et tehnoloogiad on ainult Java või C++ põhised. Tegelikult ei ole alati vajalik kasutada objektide säilitamist, tihti piisab mõnede väärtuste säilitamisest andmebaasis. See, kas objektide (või lihtsalt objektide väljade väärtuste) säilitamine on vajalik ja mis tingimustel (transaktsioonide realiseerimine jne.) otsustakse iga konkreetse projekti jaoks.

PSS pakub head abstraktsust ning lihtsa programmeerimismudelit. PSS samaväärtuslikuna asendusena saab vaadelda need tehnoloogiaid, mis on samamoodi lihtsalt kasutatavad ning annavad sarnast abstraktsiooni – võimalust töötada objektidega, mitte andmebaasi tabeli kirjetega. Üks selliseid tehnoloogiaid on Java Persistance API (JPA).

4.1 Java Persistence API

Java Persistence API (JPA) tegeleb relatsioonilise andmebaasi andmete kujutamisega Java objektidesse ("säilitavad olemid" - *persistent entities*), defineerib viisi säilitada need objektid andmebaasi tabelites eesmärkiga, et need objektid oleks kättesaadavad ka peale seda, kui neid loodud programm oma tööd lõpetanud. Java Persistence API on osa Java Enterprise Edition (*Java EE*) platvormist ja selle eesmärgiks on EJB 3.0 (*Enterprise Java Beans 3.0*) kasutamise lihtsustamine. Kuigi JPA on osa Java EE tehnoloogiast, saab seda kasutada ka tavalistes Java rakendustes (*Java SE - Standard Edition*).

Java Persistence API loomise põhjusteks olid eelmiste Java 2 EE raames standardiseeritud objektide säilitamise tehnoloogiate ebapopulaarsus, nimelt EJB 2.1 spetsifikatsiooni kuuluva Entity Beans spetsifikatsiooni (säilitava olekuga objektid) komplitseeritud arhitektuur, keeruline programmeerimismudel ja madal jõudlus. Tarkvara arenduse käigus EJB 2.1 oli enamasti

asendatud lihtsamate tehnoloogiatega (raamistikutega), mis ei olnud Java 2 EE raames standardiseeritud – näiteks Hibernate [HIBERNATE], Oracle TopLink [TOPLINK], Java Data Objects (JDO) [JDO] jne.

Java Persistence API väljaarendamisel olid arvestatud need ideed, mis olid kasutusel just populaarsust saavutatud tehnoloogiatel. Java Persistence API pakub märkamisväärsed muudatused võrreldes eelmistega EJB spetsifikatsioonidega, pakkudes võimalust kasutada seda API-d ka väljaspool Java EE rakendustest [JAVA_EE]:

- Vähem klasse ja liideseid
- Paigaldamise kirjeldusfailid on asendatud Java annotatsioonidega
- Lihtsam konfigureerimine – on olemas vaikeväärtused
- Lihtsam ja selgem objekt-relatsiooniline kujutamine
- On olemas standardiseeritud päringukeel - Java Persistence query language (EJB QL)
- Võimalused juhtida transaktsioone
- Säilitavad objektid on tavalised Java klassid - Plain Old Java Object (POJO), seega ei pea implementeerima eeldefineeritud liideseid
- On kasutatav väljaspool Java EE konteinerit – Java SE rakendustes
- Säilitavuse varustajad (*Persistence Providers*) võivad olla erinevad ja neid saab asendada (JPA spetsifikatsioonile vastavad varustajad)

Java Persistence API on loodud arvestades eelmiste spetsifikatsioonide probleeme, seega tulemuseks on parema jõudlusega ja mugavama programmeerimismudeliga API.

Java Persistence API kasutamine

Java Persistence API kasutab tavalisi Java objekte (POJO), säilitavuse märkimiseks ei ole vaja klassi laiendada või implementeerida liideseid – kasutatakse Java annotatsioone (Java keele osa alates versioonist 5). Annotatsioonide abil defineeritakse vajalikke metaandmeid otseselt Java lähtekoogi failis [JAVADOC_ANNOTATIONS]. Annotatsioonide abil defineeritakse klassi säilitavust, objekti võti (mis väli on objekti unikaalseks identifikaatoriks ning võti genereerimise strateegia), kujutamise parameetrid, objektivahelised seosed, tabelite ja veergude nimed jne. Mõned annotatsioonid on kohustuslikud, mõned mitte; enamus annotatsioonidest omavad vaikeväärtusi mida kasutatakse vastava annotatsiooni puudumisel. Tuleb mainida, et

neid samu väärtusi on võimalik defineerida annotatsioone kasutamata – XML failiga, kus on kirjeldatud samad metaandmed. JPA klassid ja liidesed on defineeritud paketis *javax.persistence*. Selleks, et Java klassis saaks kasutada JPA API-d, lähtekoodi failidesse on vaja importida vastav pakett (import *javax.persistence.**;). Lisaks klassis peab olema argumendita konstruktor ning säilitavad väljad peavad olema kas *private, package-private,* või *protected* juurdepääsuga ning omama *set* ja *get* avalikke meetodeid [JPA_INTR].

Kui klass on defineeritud, saab seda kasutada nagu tavalist klassi, kuigi programmi koodis on vaja öelda, et loodud objekti olekut on vaja säilitada. Selleks kasutakse eraldi liidest *javax.persistence.EntityManager*. *EntityManager* meetodite abil saab objekti olekut säilitada, leida objekti, kustutada jne. Ise säilitav objekt ei oma meetodeid enda oleku haldamiseks, kõik tegevused säilitava oleku muutmiseks teostatakse *EntityManager´i* kaudu. *EntityManager ´il* roll on sarnane CORBA PSS-i *StorageHome* liidese rolliga.

Lihtsaima säilitava klassi defineerimise näitena võib tuua sellist koodilõiku:

```
import java.io.Serializable;
import javax.persistence.*;
@Entity
public class JpaUser implements Serializable {
   @Id
   @GeneratedValue(strategy = GenerationType.AUTO)
   private Long id;
   @Column(name="name")
    private String username;
   /** Creates a new instance of JpaUser */
   public JpaUser() {
   }
   public Long getId() {
      return this.id;
   }
   public void setId(Long id) {
      this.id = id;
   }
   public String getUsername() {
      return username;
   }
```

```
    public void setUsername(String username) {
        this.username = username;
    }
}
```

Antud klassi defineerimine ei erine palju tavalise klassi defineerimisest. Antud klass laiendab liidest *java.io.Serializable*, kuid see on vajalik ainult siis, kui seda objekti kasutatakse kaugprotseduurite argumendina [J5EE_TUT]. Objektide säilitavus on defineeritud annotatsiooniga *@Entity* – see näitab, et selle klassi isendid on säilitavad olemid. Lisaks on defineeritud olemi võti (annotatsioon *@Id*) ja selle genereerimise strateegia – antud juhul on annoteeritud, et see on automaatselt genereeritav väärtus *@GeneratedValue(strategy = GenerationType.AUTO)*. Võtme defineerimist saab kontrollida, näiteks kasutades funktsooni. Samuti on konfigureeritavad ka kujutamised: *@Column(name="name")* ütleb, et vastavale väljale klassis vastab andmebaasi tabeli veerg nimega "name".

Säilitava klassi kasutamine vajab *EntityManager´i* aktiveerimist vastava vabriku abil. Näide JPA kasutamisest on toodud *Lisas 4*. Objekte luuakse tavalisel moel. Kui soovitakse olekut säilitada, kasutakse meetodit *persist(Object o)*, mille väljakutsumise tulemusena objekti olekut säilitakse andmebaasi tabelis. entityManager.*getTransaction()* abil on kättesaadav jooksev transaktsioon.

EntityManager´i tähtsamad meetodid on:

- *persist(Object o)* – objekti säilitamine
- *createQuery(String qlString)* – objektide pärimine andmebaasist päringukeele abil
- *find(Class<T> entityClass, Object primaryKey)* – objekti otsing võtme järgi
- *merge(Object o)* – objekti oleku uuendamine andmebaasis
- *remove(Object o)* – objekti kustutamine andmebaasist.

Täpsemat ja täielikumat ülevaadet Java Persistence API-st saab allikatest [JPA_INTR] ja [J5EE_TUT]

Java Persistence API Päringukeel

Java Persistence päringukeele abil saab koostada päringud, et pärida säilitavad objektid ja nende olekud. See keel on SQL-sarnane, kuid kasutab abstraktsed *schema-d* ja ei sõltu kasutavatest andmehoidlast [J5EE_TUT].

Kõige rohkem leiab kasutust SELECT lause, selle süntaks üldiselt jälgib SQL süntaksit (saab kasutada standardsed SELECT, FROM, WHERE, GROUP BY, HAVING, ORDER BY), kuid erinevalt SQL-st tulemusest saab kohe lugeda objekti. Lisaks on olemas UPDATE ja DELETE laused, millede abil saab objekte uuendada ja kustutada. Java Persistence päringukeel on oma funktsionaalsuse poolest standardse SQL-i tasemel ning sobiv objektidega töötamiseks [J5EE_TUT].

4.2 PSS ja JPA võrdlus

PSS ja JPA vahel on olemas nii erinevused, kui ka ühised osad. Kõige tähtsam erinevus on see, et PSS on loomulik CORRBA spetsifikatsiooni osa ja tänu sellele seda saab kasutada erinevate programmeerimiskeeltega ja seda saab kasutada ka klindirakendustes. JPA seevastu kasutatav ainult Java platvormil ja selle kasutusvaldkond piirdub serveripoolsete CORBA rakendustega (võimalikud heterogeensed kliendid) – loomulikult see kehtib ainult CORBA rakenduste jaoks, mitte tavaliste Java rakenduste jaoks.

Veel üks aspekt on transaktsioonide kasutamine. PSS on sujuvalt integreeritav CORBA transaktsioonide teenusega (OTS) ja vajab minimaalset CORBA realisatsiooni konfigureerimist. JPA juures OTS teenusega integreerimine on ka võimalik, kuid on komplitseeritum. Sellest räägitakse järgmistes jaotustes.

Ühine PSS ja JPA vahel on see, et need mõlemad võimaldavad saavutada abstraktsust ja peidavad andmehoidla omapärad (PSS juures see tase on kõrgem). Lisaks mõlemal juhul säilitavust defineeritakse deklaratiivselt – PSDL failis ja Java annotatsioonidega. Programmeerimismudelid on arusaadavadja lihtsasti õpitavad mõlemal juhul.

5 Transaktsioonid hajussüsteemides

Transaktsioonid omavad väga suurt rolli suurte hajussüsteemide loomisel. Transaktsiooni definitsioon on järgmine: transaktsioon on operatsioonide hulk mis jälgib ACID (A*tomicity, Consistency, Isolation, Durability*) printsiipe ning viib süsteemi ühest mittevastuolulisest olekust teisse mittevastuolulisse olekusse. ACID printsiibid on järmised [BOOK_CORBA]:

Atomic (Atomaarsus): Kui transaktsioon on katkestatud veaolukorra tõttu, siis kõik selle transaktsiooni tegevused võetakse tagasi (*rollback*). Rakendustransaktsiooni näitena võib tuua raha ülekandmist, kui raha võetakse ühelt arvelt ja kantakse üle teisele arvele. Selleks, et tagada korrektsust, mõlemad tegevused peavad olema tehtud ühe operatsioonina: kas kõik tegevused õnnestuvad või mitte ühtegi.

Consistent (Terviklikkus): Transaktsiooni efektid ei tohi olla poolikud. Rakendustransaktsiooni näitena võib tuua raamatupidamise süsteemi – kõikide arvete summa selles süsteemis peab olema null. Kui võtta raha ühelt arvelt ja mitte lisada seda teisele arvele, siis summa ei ole enam null ja kogu süsteemi terviklikkus on rikutud.

Isolated (Isoleeritus): Transaktsiooni vahepealsed muudatused ei ole nähtavad väljaspool transaktsiooni kuni transaktsiooni lõppu. Rakendustransaktsiooni näitena võib tuua raha ülekandmist ühelt arvelt teisele – mingil hetkel, kui raha on võetud ühelt arvelt ja ei ole veel juurde lisatud teisele arvele, tekib olukord, et väliste kontrollide jaoks ülekandmise summa ei ole null. Selline olukord on võimalik ainult siis, kui transaktsiooni sisu ei ole isoleeritud. Isoleerimine lahendab sellist probleemi.

Durable (Püsivus): Lõppenud transaktsiooni mõju on püsiv ja ei kao, välja arvatud olukorras, kus toimub kriitiline süsteemiviga. Tavaliselt see tähendab, et õnnestunud transaktsiooni sisu peab olema füüsiliselt salvestatud andmekandjale.

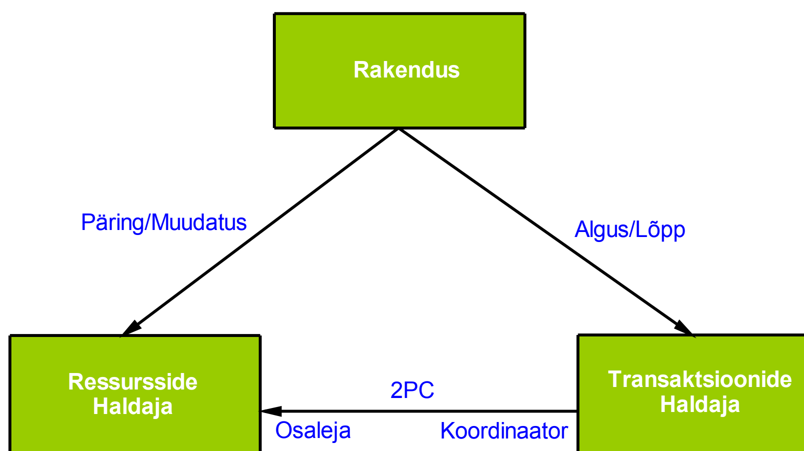
ACID transaktsioon võib olla lõpetatud ainult kahel viisil: kas kogu transaktsiooni sisu on kinnitatud (*commited*), või võetud tagasi (*rolled back*). Kui transaktsioon on kinnitatud, siis selle sisu muutub püsivaks, vastasel juhul (*rollback*) kogu sisu jääb tegemata .

5.1 Transaktsioonide juhtimine

Tavaliselt transaktsioonidega assotsieeruvad andmebaaside transaktsioonid ja neid realiseeritakse konkreetese RDBMS vahendite abil – kasutades SQL-i (või JDBC vahendeid). Kuid hajussüsteemides võivad korraga esineda mitu erilist tüüpi (erinevad produktid erinevatest

tootjatest) andmebaase, lisaks on võimalikud teised ressurssid ja transaktsioonilised failisüsteemid. Hajussüsteemide transaktsioonid võivad toimuda IPC (*Inter-Process Communication*) protokolli kaudu ja ühes transaktsioonis võivad korraga osaleda mitu protsessi. Selleks, et kindlustada transaktsioonide korrektset toimumist taolistest süsteemidest, kasutatakse transaktsioonide haldurit (*Transaction Manager*), mis mängib keskset rolli transaktsioonide läbiviimisel [BOOK_CORBA].

Iga transaktsioonide töötlemise süsteemi (*TP-system – transaction processig system*) kuluvad kolm elemendi: rakendus, ressursside haldaja, transaktsioonide haldaja.



Joonis 5.1: Hajusate transaktsioonide töötlemise süsteemi komponendid [BOOK_CORBA]

Joonisel on näidatud nende komponentide koostööd. Rakendus kasutab transaktsioonide haldajat transaktsiooni alustamiseks ja lõpetamiseks. Ressursside haldaja pakub liideseid rakenduse jaoks, mille abil saab teostada transaktsiooni sisu – näiteks andmete pärimine ja muutmise jaoks. Peale seda, kui rakendus lõpetab transaktsiooni, transaktsioonide haldaja teeb koostööd erinevate ressursside haldajatega (kasutades 2PC – *2 Phase Commit Protcol*) ja lõpetab transaktsiooni sellisel moel, et tagada selle atomaarsust.

Kui transaktsiooni tulemusena andmed muutuvad kahes (või rohkem) ressursside haldajas, siis atomaarsuse saavutamiseks on vaja kasutada lisavahendeid, sest iga haldaja võib tekitada vea mis ei luba lõpetada transaktsiooni edukalt. Kui esimene ressursside haldaja lõpetab oma transaktsiooni edukalt, kuid teine haldaja ei saa kinnitada muudatusi, siis kogu hajusa transaktsiooni atomaarsus rikub, sest transaktsioon õnnestub ainult poolikult. Selle probleemi lahendamiseks kasutatakse kahefaasilist kehtestamise protokolli (2PC - *2 Phase Commit Protcol*) [BOOK_CORBA].

Esimeses kehtestamise faasis transaktsioonide haldaja teostab “valimiste” protseduuri. Selle

protseduuri osalejateks on ressursside haldajad. Iga osaleja omab vetoõigust – õigust katkestada kogu transaktsiooni. Kui ressursside haldaja teatab lõpetamisest, siis see tähendab, et vajadusel ta on valmis lõpetada oma osa transaktsioonist. Sellega esimene, ettevalmistamise, faas lõpeb.

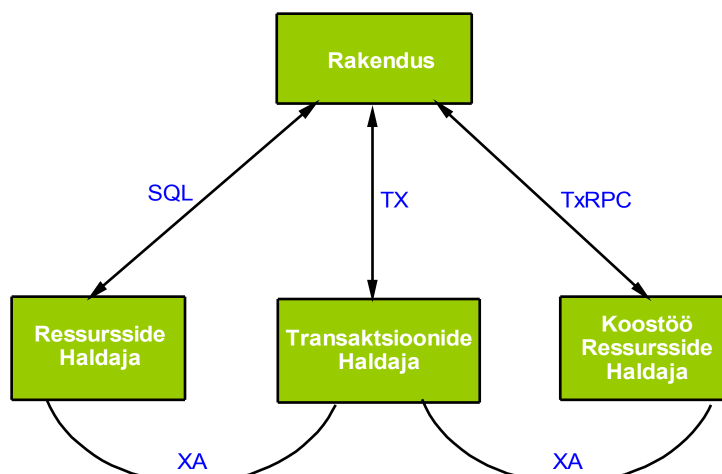
Sõltuvalt esimese faasi tulemusest (kas kõik saadud vastused on positiivsed või mitte), transaktsioonide haldaja saadab kõikidele ressursside haldajate kas kinnitamise (*commit*) või katkestamise (*rollback*) teade. Kõik osalejad saavad sama teade. Teatades kinnitamisest, iga ressursside haldaja teatab, et tema on valmis lõplikult viia muudatused oma ressurssidesse (või katkestamise korral taastada esialgne seis). See tähendab, et muudatused kirjutatakse füüsilisele andmekandjale lõplikult. Sel sammul ressursside haldaja kirjutab oma logisse muudatuste sisu, et hiljem vajadusel ikkagi saaks muudatusi tagasi võtta. Kui ressursside haldajal ei õnnestunud täita transaktsioonide haldaja juhised, siis tema katkestab suhtlemist ning kogu transaktsioon ebaõnnestub. Selline mudel töötab korrektselt ainult siis, kui kõik osapooled järgivad 2PC protokolli reegleid. [2PC]

On olemas kaks standardi, mis määravad üldiseid liideseid komponentide koostööks transaktsioonide toimumisel hajusates süsteemides: X/Open DTP ja CORBA hajusate transaktsioonide teenus – OTS [BOOK_CORBA].

5.2 Transaktsioonide juhtimise mudelid

X/Open DTP

X/Open *Reference Model for Distributed Transactions Processing* (Hajusate transaktsioonide töötlemise baasmudel) määrab liidesed standardsete TP-süsteemi komponentide vahel.



Joonis . X/Open DTP mudel [BOOK_CORBA]

Esialgne versioon ilmus aastal 1991 ja praeguseks hetkeks X/Open DTP on saanud üldtunnustatud standardiks. Joonisel on näidatud komponendid ja nendevahelised suhtlemisprotokollid.

- Transaktsioonide haldaja

  Võimaldab klientidel alustada ja lõpetada transaktsioone ning vastutab kahefaasilise kehtestamise protokolli toimimise eest.

- Ressursside haldaja

  Juhib transaktsioonilise ressursse. Tavaliselt ressursside haldaja rollis on relatsioonilise andmebaasi haldur või transaktsioonilise failisüsteemi haldur.

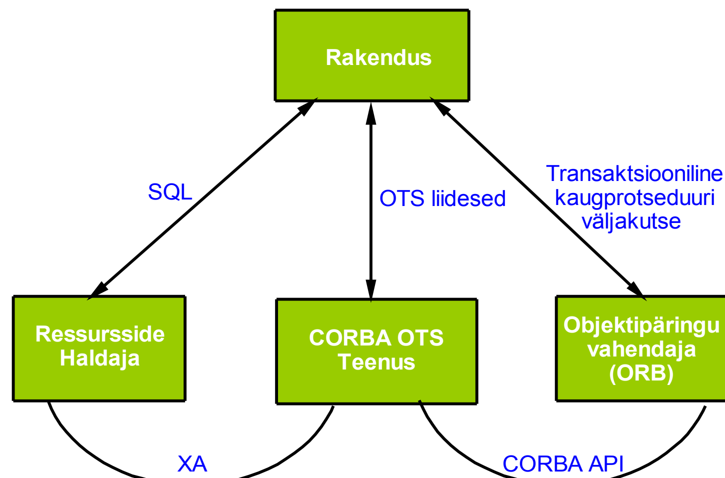
- Koostöö Ressursside haldaja (CRM – *communication resource manager*)

  Võimaldab klientidele luua transaktsioonilisi kaugühendusi. Baseerub kaugprotseduuride liidese peal (kasutades näiteks RPC – *Remote Procedure Call*).

Rakendused suhtlevad transaktsioonide haldajaga kasutades TX liidest. Rakenduse ja ressursside haldaja vaheline protokoll ei ole standardiseeritud – kasutatakse konkreetse RDBMS poolt pakutava protokolli. Rakendus võib luua transaktsioonilisi ühendusi koostöö ressursside haldaja abil. Transaktsiooni lõppemisel transaktsioonide haldaja käitub vastavalt 2PC protokollile ja ühendab osalenud ressursside haldurid. Transaktsioonide ja ressursside haldajate vaheline suhtlemine toimub XA liidese kaudu [BOOK_CORBA].

CORBA Transaktsioonide teenus

Kui X/Open DTP mudel määrab protseduuride liidesed, siis CORBA transaktsioonide teenus OTS (*Object Transaction Service*) määrab hajusad, objektorienteeritud liidesed TP-komponentidele. OTS spetsifikatsioon oli välja arendatud eesmärgiga olla täielikult ühendav X/Open DTP mudeliga, sellepärast OTS näeb välja kui X/Open DTP edasiarendus, mitte täielik asendus [BOOK_CORBA].



Joonis 5.2: CORBA OTS mudel [BOOK_CORBA]

CORBA OTS kasutab X/Open DTP mudeliga ühenduvad ressursside haldajad. Hajusate transaktsioonide läbiviimiseks rakendused saavad kasutada spetsifitseeritud objektorienteeritud CORBA liidesed (mitte X/Open liidesed), samas OTS teenus suhtleb ressursside haldajatega standardse XA liideste abil. See tähendab, et igat ressursside haldajat, mis toimib X/Open mudeli järgi, saab kasutada koos OTS teenusega [BOOK_CORBA].

Rakendus saab iseseisvalt suhelda ressursside haldajaga näiteks RDBMS liidese kaudu. Samas CORBA spetsifikatsioon pakub võimalust täielikult abstraheeruda ressurssidest ja välistada kliendi otsest suhtlemist ressursside haldajaga, kasutades PSS teenust CORBA liideste kaudu. PSS kasutamine OTS-ga ei ole kohustuslik ja sõltub süsteemi arendajate otsustest.

OTS teenuse spetsifikatsioon määrab kahte erinevat viisi transaktsioonide loomiseks:

- Loomine ilmutatud kujul

  Sellise viisi kasutamisel uute transaktsioonide loomiseks kasutatakse transaktsioonide vabrikut. Transaktsioonid on esitatud CORBA objektidena. Rakendusel on juurdepääs transaktsiooni juhtimisele.

- Ilmutamata kujul transaktsioonid

  Rakendusel ei ole juurdepääsu transaktsiooni loomisele. See mudel on kõige levinum meetod OTS kasutamiseks.

OTS spetsifikatsioon määrab *Current* liidest, millele vastab jooksev OTS-i lõim. Seda liidest kasutatakse transaktsioonide kasutamisel ilmutamata kujul. Rakendus kutsub välja *Current* liidese meetodit *begin()* ja selle peale OTS seostab jooksva transaktsiooni lõime kliendi lõimega

(tekib seos *transaktsioon-lõim*, seda saab luua tänu sellele, et kliendilt transparentselt saadetakse transaktsiooni konteks). OTS jälgib transaktsiooni täitmist just selle seose kontekstis. [BOOK_CORBA]. Näitena transaktsiooni kasutamisest ilmutamata kujul võib tuua sellise Java koodilõiku:

```
org.omg.CORBA.Object tr_obj = orb.resolve_initial_references( "TransactionCurrent" );
org.omg.CosTransactions.Current current =
        org.omg.CosTransactions.CurrentHelper.narrow( tr_obj );
current.begin();
        corba_obj.createItem("Item1");
current.commit();
```

6 Näidisrakendus

6.1 Näidisrakenduse eesmärk

Selle töö praktiline pool oli pühendatud sellele, et katsetada CORBA PSS ja OTS teenuste tööd reaalses rakenduses. Selleks luua näidisrakendus, mis kajastaks reaalelu rakenduse funktsionaalsed nõuded, kuid oleks piisavalt triviaalne. Tähelepanu oli suunatud just säilivate objektide hoidmise võimaluste kasutamisele – eriti programmeerija vaatenurgast.

Teine eesmärk oli proovida kasutada CORBA rakenduses PSS asendusena üldisema, mitte CORBA poolt spetsifitseeritud, raamistiku objektide säilitavuse saavutamiseks – nimelt Java Perrsistance API, mis oli eelmistes peatukkides juba kirjeldatud. Huvi pakkus JPA integreerimise võimalused arvestades CORBA hajusate rakenduste eripärad – nimelt võimalused peita kliendilt objektide säilitavuse implementatsiooni, hajusate transaktsioonide läbiviimine OTS teenuse abil jne. JPA ja PSS teoreetiline võrdlus oli tehtud peatükis *4*.

Lisaks sellele oli planeeritud teha jõudluse katsed mõlema variandi jaoks (PSS ja JPA), et hinnata jõudluse näitajad. Olgu kohe öeldud, et kvantitatiivsete hinnangute (protsessori aeg, latents jne.) kasutamine ei ole päris korrektne. Selleks on mitu põhjust. Kõigepealt, tuleb arvestada, et on olemas terve hulk CORBA realisatsioone – iga neist pakub oma jõudluse taset (mõned on realiseeritud C/C++ , teised Javas või isegi Pythonis), mõned realisatsioonid sobivad kriitiliste ülesannete täitmiseks (reaalaja realisatsioonid), teised mitte jne. [WIKI_CORBA]. See kehtib ka JPA kohta – jõudlus võib sõltuda näiteks virtuaalmasina versioonist. Teiseks tuleb arvestada sellega, et PSS oli spetsifitseeritud spetsiaalselt CORBA jaoks, mis võib põhjustada mõnede JPA-s puuduvate iseärasuste olemasolu (funktsionaalsus on defineeritud IDL liideste abil jne.). Seega analüüsiks oli püütud enamasti kasutada kvalitatiivseid tunnuseid.

6.2 Tehnoloogiliste aspektide analüüs

Rakenduse loomisel tulid esile üldisemad ning konkreetsemad CORBA, PSS, JPA ja teiste tehnoloogiate omapärad, probleemid ning tugevad küljed. Loodud näidisrakenduse baasil on võimalik hinnata mõned aspektid. Mõnedest neist täpsemalt.

Transaktsioonid

Rakenduses oli püütud kasutada ilmutamata kujul transaktsioone. See on soovitatav lähenemine nii PSS, kui ka JPA puhul. Hajusate transaktsioonide korral PSS ei kasuta metaandmeid

transaktsioonide defineerimiseks (sessionne initsialiseeritakse programmselt), JPA aga kasutab (annotatsioonid ja lisakonfiguratsioonifailid). PSS kasutamine tänu sellele tundub olema selgem.

Kasutamise mugavus

Üldiselt saab öelda, et valitud tehnoloogiatega oli võimalik realiseerida vajalikku funktsionaalsust. Samas JPA pakub tunduvalt rohkem standardseid võimalusi ja programmeerijal on valik lahendada ülesanded erineval viisil. Saab valida ka abstraktsuse taset – kui on soov rohkem kontrollida andmebaasi tabelid (defineerida veerude atribuudid, kasutada väärtuste generaatorid jne.). PSS juures tekitab palju ebamugavusi asjaolu, et puudub võimalus automaatselt genereerida uued võtmed lisatavate kirjete jaoks. Üldiselt võib öelda et nende jaoks, kes on tutvunud SQL-ga, JPA on arusaadavam.

6.3 Kasutatava PSS realisatsiooni eripärad

Kasutatud CORBA realisatsioon OpenORB pakub PSS kasutamise võimalust (vastab OMG PSS spetsifikatsioonile 2.0). PSS realisatsioon ei sõltu konkreetsest andmebaasist ja peaks töötama enamusega nendest. Testrakenduses oli kasutatud MySql andmebaas, mis on küllaltki populaarne [MYSQL]. Selgus, et OpenORB-i PSS ei ole päris ühenduv MySql andmebaasiga (indeksite loomisel tabelite peal tekkis SQL viga) ning et konfiguratsiooni muutmisega ei saa kõik probleemid ära lahendada. Lahendusena oli muudetud PSS lähtekood. Paindlikum PSS konfiguratsiooni süsteem võiks olla selle alternatiiviks. Selliste ebamugavuste olemasolu viitab mitte ainult konkreetse realisatsiooni probleemidele, vaid ka selle, et tänapäeval ei ole lootust luua rakendust ilma erinevate andmebaasisüsteemide omapärade arvestamata.

Nõrgaks kohaks võib nimetada päringukeele puudumist (mis ei ole OMG poolt spetsifitseeritud). Selle olemasolu lihtsustaks PSS kasutamist märgatavalt. Antud teema on juba töös eelnevalt arutletud.

6.4 Testrakenduse valdkond

Testrakenduse teemaks on totalisaator – sündmuste registreerimise ja panuste tegemise süsteem. Taolised süsteemid on kasutusel näiteks spordiürituste tulemuste ennustamisel. Kasutajad registreerivad ennast ning seejärel saavad teha panused olemasolevatele sündmustele. Igal sündmusel on olemas staatus ja lõpptulemus. Kasutaja saab vaadata oma panuste tulemusi. Kuna tegemist on testrakendusega, oli realiseeritud ainult põhifunktsionaalsus – sündmuste

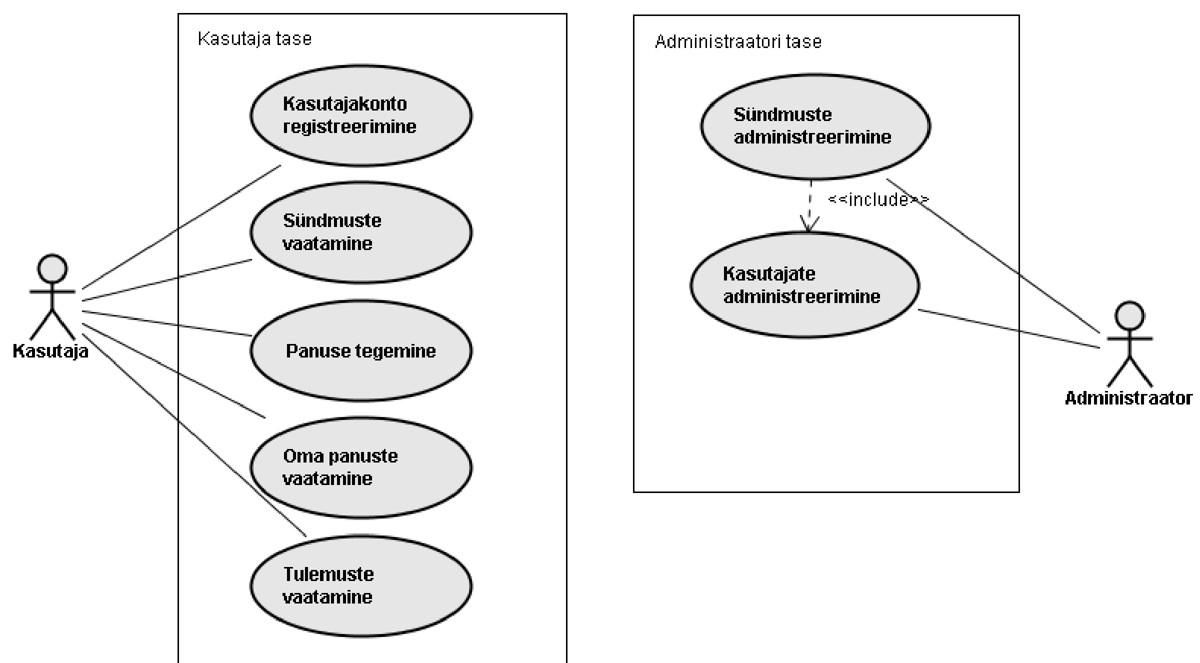
registreerimine, kasutajate registreerimine, panuste tegemine, tulemuste vaatamine. On olemas veebipõhine kasutajaliides.

Sellise valdkonna rakendus sobis testrakenduseks sellepärast, et peaaegu kõik selle süsteemi tähtsamaid olemid saab esitada säilivatena objektidena. Nende hulka kuuluvad: Kasutaja *(User)*, Sündmus *(Event)*, Panus (*Bet*). Need olemid peavad olema säilitavad, et rakenduse uuesti käivitamise järel andmeid jääks püsima.

Testrakenduse lähtekood koos käivitatava veebirakendusega on lisatud tööle CD-na. Kasutajajuhend koos käivitamise õpetusega on antud lisana (*Lisa 1*).

6.5 Kasutuslood

Süsteemiskasutajad on eraldatud kahte gruppi: tavakasutajad ja administraatorid. Süsteemil nendel on erinevad rollid. Tüüpilised kasutuslood on näidatud UML *Use Case* diagrammil [UML].



Joonis 6.1.: Kasutuslood

Tavakasutaja rollid on:

- Kasutajakonto registreerimine – uus kasutaja määrab kasutajanimi ja salasõna. Süsteemis tekib uus *User* objekt.
- Sündmuste vaatamine – olemasolevate sündmuste vaatamine. Nimekiri *Event* objektidest.

- Panuse tegemine – kasutaja määrab panust kindlale sündmusele, tekib uus *Bet* objekt.
- Panuste vaatamine – kasutaja enda panuste vaatamine, nimekiri *Bet* objektidest.
- Tulemuste vaatamine – kasutaja panuste ja tulemuste vastavus.

Administraatori rollid on:

- Sündmuste administreerimine – uute sündmuste tekitamine, olemasolevate sündmuste muutmine (sh. kustutamine). Tegevused *Event* objektidega.

6.6 Kasutajaliides

Loodud veebiliides on kättesaadav veebilehitsejat kasutades ja võimaldab katsetada rakenduse tööd: luua uued objektid, pärida, muuta ja kustutada neid. On võimalik jooksvalt valida PSS ja JPA vahel. Samuti liidese abil on kättesaadav testimise funktsionaalsus:

- *Event* objektide loomine – nende arvu määramine
- *Event* objektide kustutamine
- *Bet* objektide loomine – nende arvu määramine
- *Event* objektide kustutamine

Create Events OK Time:
Create Bets OK Time:

Return Delete All Events Delete All Bets Delete Tests data...

Tests					
Objects	Event	Time	Persistance	Run date	Comment
No items found.					

Joonis 6.2.: Testimise veebileht

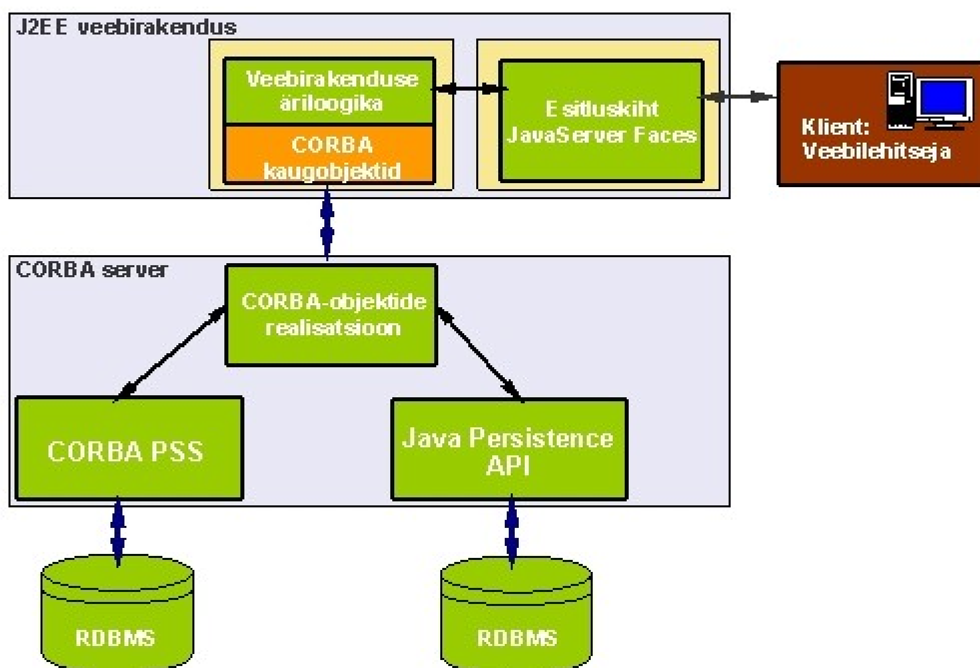
Iga testkatse järel ilmub testile kulunud aeg ja see tulemus salvestatakse tabelisse. Saab näha iga katse statistikat. Tabeli sisu säilib ka peale rakenduse väljalülitamist.

Kasutajaliidese keeleks on inglise keel. Kasutajajuhend on antud lisana (*Lisa 1*).

6.7 Arhitektuur

Testrakenduse kavandamisel ei olnud eesmärki kasutada ainult CORBA-spetsiifilisi vahendeid.

CORBA abil saab ehitada suvalise suurusega rakendusi ning integreerida CORBA poolt pakutava funktsionaalsust teiste tehnoloogiatega. CORBA funktsionaalsus võib olla kasutatud suuremal või väiksemal määral. Suurim osa CORBA funktsionaalsusest kasutatakse serveri poolel, kliendi poolel CORBA-spetsiifilist koodi on tavaliselt palju vähem.



Joonis 6.3.: Näidisrakenduse arhitektuur

Joonisel on näidatud loodud rakenduse arhitektuur. Rakendust saab loogiliselt jagada kaheks osaks:

- CORBA server – serveris on realiseeritud suurim osa rakenduse äriloogikast, selle funktsionaalsuse abil toimub kasutatavate rakenduslikke objektide haldamine. Serveris aktiveeritakse teenreid, nende liidesed on spetsifitseeritud IDL keele abil ning on kättesaadavad ORB-i kaudu kliendi poolel. Kõik tegevused säilitavate andmetega teostatakse serverirakenduse sees.
- J2EE veebirakendus - kujutab endast Java EE *(Java Enterprise Edition)* veebirakendust. Oli kasutatud J2EE, mitte J5EE tehnoloogia, sest ei olnud vajadust kasutada uuemat spetsifikatsiooni. Veebirakendus on ühest küljest CORBA serveri klient (rakenduse laadimisel aktiveeritakse ORB-i ja kaugobjektid), teisest küljest veebiserver, mis pakub graafilise liidese kasutamise võimalust. Suurim osa veebirakenduse äriloogikast tegeleb kasutajaliidese poolt tulevate sündmuste töötlemisega ja CORBA serveriga suhtlemisega. Veebirakendus ise ei suhtle

andmehoidlaga.

Selline arhitektuur võimaldab eraldada CORBA serveri ja eraldiseisva veebirakenduse funktsionaalsust. Serveri funktsionaalsuse realiseerimisel ei pea arvestama kasutajaliidese omapäradega, tuleb kokku leppida ainult liidesed ja andmetüübid – need on spetsifitseeritud vastavas IDL failis. Sellisel juhul on võimalik, et serveri funktsionaalsust hakkab kasutama ka mingi teine klient, näiteks eraldiseisev Java või C++ rakendus.

CORBA server peidab klientidelt andmehoidla olemasolu ja selle kasutamise eripärad. Testrakenduse CORBA server oli realiseeritud niimoodi, et seda saaks käivitada kahe erineva säilitavate objektide varustajaga – CORBA PSS-ga ja Java Persistance API-ga (JPA). Kahjuks PSS ja JPA poolt tekitavad andmebaasimudelid ei ole identsed ja selle tõttu ei saa kasutada ainult üht andmebaasi. Seega PSS või JPA korral on kasutusel erinevad andmebaasid (ja vastavalt erinevad andmed). Selline olukord ei ole aktsepteeritav reaalelu rakendustes, kuid tegemist on testrakendusega ja selle kasutamise eesmärk on uurimuslik. Erandiks on kasutajakontode ja testtulemuste info, mida hoitakse ainult ühes andmehoidlas.

Täiendavat infot Java EE tehnoloogiast ja veebirakendustest saab allikatest [J5EE_TUT] ja [JAVA_EE].

6.8 Kasutatavad tehnoloogiad ja vahendid

Kuigi antud töö teema on seotud CORBA-ga, testrakenduse valmimisel olid kasutatud erinevad tehnoloogiad.

CORBA serveri juures vajavad mainimist:

- Kasutatav CORBA realisatsioon – oli kasutatud avaliku lähtekoodiga realisatsioon OpenORB 1.4 [OPEN_ORB]. OpenORB implementeerib täielikult CORBA tuuma spetsifikatsiooni 2.4.2 ning pakub kõikide (mis on märkimisväärne) COS lisateenuste realisatsioonid. OpenORB põhilised featuurid on [OPEN_ORB_DOC]:
    - Realiseeritud Javas
    - Multilõimeline realisatsioon
    - Modulaarne ja paindlikult konfigureeritav
    - DynAny (dünaamiline määramata tüüp), DII (*Dynamic Invocation Interface*), DSI (*Dynamic Skeleton Interface*) kasutamise võimalused
    - Arendusvahendite olemasolu

- Kasutatav PSS realisatsioon – OpenORB 1.4 Persistent State Service, vastab OMG PSS spetsifikatsioonile 2.0 , “ptc/01-12-02”. OpenORB põhilised featuurid on [OPEN_ORB_DOC]:
  - PSDL kompilaator
  - Andmebaasi, Faili, Mälu Ühendajad
  - Baas - ja Transaktsioonilised sessioonid, sessioonide varu
- Kasutatav Java Persistence API realisatsioon - Java 5 EE soovitusrealisatsioon (*reference implementation)*, säilitatavuse varustajaks on Oracle TopLink Essentials. Oracle TopLink Essentials on limiteeritud versioon Oracle TopLink vahendist, mis vastab JPA spetsifikatsioonile [JAVA_EE],[TOPLINK].
- Kasutatav andmebaas – MySql 5.0 koos selle JDBC draiveriga (JConnector). MySql andmebaas toetab transaktsioonide isoleerimist kõikidel spetsifitseeritud tasemetel, täielikult vastab X/Open XA mudeli nõuetele. [MYSQL]

J2EE veebirakenduse juures vajavad mainimist:

- Kasutatav CORBA realisatsioon – sama, mis CORBA serveris (OpenORB 1.4)
- Kasutatav J2EE veebikonteiner – Apache Tomcat 5.5.17. Vabavaraline avaliku lähtekoodiga veebikonteiner [TOMCAT].
- JSF (*JavaServer Faces*) – serveripoolne kasutajaliidese komponentide raamistik veebirakenduste jaoks [JSF].

Kasutatavad CORBA teenused : Nimeteenus *(Naming Service),* PSS (*Persistent State Service*), OTS (*Object Transaction Service*).

Tarkvaraarendamise vahenditest arendamise ja testimise käigus olid kasutatud: NetBeans 5.5 (vabavaraline avaliku lähtekoodiga Java IDE koos visuaalse veebirakenduste ehitamise võimalustega *(Visual Web Pack))* [NETBEANS], NetBeans Profiler (rakenduste profileerimise vahend) [NETBEANS], Apache Ant (projektiehituse vahend) [ANT], OpenORB arendusvahendid (kompilaator, administreerimisliidesed jne.) [OPEN_ORB].

6.9 Rakenduse realisatsioon

CORBA serveri funktsionaalsus on realiseeritud kaugobjektidena. Nagu on tavaks CORBA rakenduste puhul, alguses oli määratud vajalik funktsionaalsus ning defineeritud IDL liidesed.

Kõik need liidesed on defineeritud ühes IDL failis (*Lisa 1*). Need liidesed mängivad Fassaad-teenri rolli ja nende kasutaja ei pea muretsema sellest, kuidas realiseeritud Fassaad-objekti meetodid. Sellise disaini plussid olid juba kirjeldatud peatükis *3.4*.

- IEventManager – liides, mille abil saab töötada sündmustega *(Events)*. Protseduurid, mis pakub see liides on: *createEvent(), removeEvent(), updateEvent(), getEvent(), getEvents()* jne.
- IBetManager – liides, mille abil saab töötada panustega (*Bets*). Protseduurid: *createBet(), removeBet(), updateBet(), getBet(), getBets()* jne.
- IUserAdmin – liides, mille abil saab töötada kasutajakontodega (*Users*). Protseduurid: *createUser(), updateUserInfo(), removeUser(), getUser(), checkPassword()* jne.

IDL liideste järgi olid kirjutatud liideseid implementeeritavad klassid. Objekte aktiveeritakse POA abil ning registreeritakse Nimeteenuse repositooriumis (*bind*- protsess). CORBA kliendid kasutavad nimeteenust, et saada serveriteenrite objektviited nime järgi (*resolve*-protsess) [BOOK_CORBA_PROGR].

Selleks, et CORBA protseduuride parameetriteks saaks kasutada liitobjekte, IDL pakub sobiva struktuuri – *struct,* Java kujutamises (*mapping*) sellele vastab avalik lõplik klass avalikku juurdepääsuga väljadega mis on mugavalt kasutatav [BOOK_CORBA_PROGR]. Selleks, et protseduuride parameetrite tüübid ei sõltuks kasutavast säilitavuse varustajast, olidki kasutusele võetud IDL *struct*-na defineeritud klassid, mis täitsid säilitavate objektide ja veebirakenduse objektide vahendaja rolli.

Objekte, mida oli vaja säilitada, oli süsteemis kolm – Sündmus (*Event*), Panus (*Bet*) ja Kasutaja (*User*). Nende olemite säilitavust käsitlevad meetodid olid realiseeritud sellisel moel, et neid saaks kasutada nii PSS-i, kui ka JPA kasutamisel. Jooksev säilitavuse varustaja on määratud iga liidest implementeeritava klassi privaatse muutujaga (*this.persistenceName*) mida algväärtustatakse teenri aktiveerimisel ja mida on võimalik muuta rakenduse töö jooksul kasutajaliidese abil (*Lisa 1*). Ühe sellise meetodi näide:

```
// Event objekti eemaldamine
public void removeEvent(String name) {
        switch (this.persistenceName){
                case PSS :
                        IEvent e = home().find_by_name( name );
                        e.destroy_object();
```

```
                    break;
            case  JPA:
                    destroyJpaEvent(getJpaEventByName(name));
                    break;
   }
}
```

Säilitavad PSS objektid on defineeritud PSDL failis (*Lisa 2*), säilitavad JPA objektid *(Entities)* on defineeritud annotatsioonide abil lähtekoodi sees, need failid asuvad eraldi paketis "*jpa_entities"* ja klasside nimed algavad sufiksiga *"Jpa" (Lisa 5).* PSDL faili sisu on kompileeritav PSDL kompilaatoriga – kompileerimise tulemusena tekivad uued *.java* failid. Annotatsioonide sisu loetakse *.java* failide kompileerimise ajal.

Veebirakendus on loodud kasutades NetBeans IDE (*Integrated Development Environment*). IDE kasutamisega on tunduvalt lihtsustatud rakenduse arendamine, kompileerimine, silumine ja testimine. Veebirakenduse liidese loomine oli tehtud kasutades visuaalsed vahendid (*Visual Web Pack*) ning pakutava komponentide paletti [NETBEANS].

Äriloogika CORBA serveriga suhtlemiseks on realiseeritud *SessionBean* (Java objekt, mis säilitab oma olekut veebilehitseja seanssi jooksul) komponendi sees. Iga algava sessiooni jaoks initsialiseeritakse ORB-i ja leitakse Nimeteenuse abil vajaklikud kaugobjektid. HTTP päringute vahel *SessionBean* objektide olek säilib. Igale konkreetsele veebilehele vastab Java klass, see omakorda kasutab *SessionBean*-i meetodid lehe äriloogika teostamiseks [J5EE_TUT].

Transaktsioonide juhtimine toimub veebirakenduse poolel (ehk CORBA kliendi juures), mitte serveri objektides, mis tegelevad operatsioonidega säilitavatel objektidel. Selline lähenemine on tüüpiline transaktsioonide läbiviimisel hajusas süsteemis, kuna transaktsioonis võivad osaleda mitu erinevat ressurssi. Transaktsioonide juhtimiseks kasutatakse OTS teenuse abil saadava liidest *Current*. Operatsiooni edukal sooritamisel kogu transaktsiooni sisu kehtestatakse, vea tekkimisel muudatused võetakse tagasi. Näitena toodud koodijupis transaktsioonis osaleb ainult üks objekt ja üks ressurss, tegelikult nende arv võib olla suurem, kuid semantika ei muutu.

```
// veebirakenduse SessionBean komponendi meetod
 public void addEvent(EventDataModel em){
        try {
           org.omg.CORBA.Object tr_obj = orb.resolve_initial_references("TransactionCurrent");
           org.omg.CosTransactions.Current current =
                   org.omg.CosTransactions.CurrentHelper.narrow( tr_obj );
```

```
        current.begin();
                // kaubobjekti metodi väljakutse
                manager.createEvent(name, description, eventDate,
                                        beginDate, endDate, status, result);
        current.commit(false);
    } catch (java.lang.Exception ex){
        ex.printStackTrace();
    }
}
```

Siin tuleb suunata tähelepanu sellele, et programmeerija jaoks hajusate transaktsioonide kasutamine on triviaalne, kogu keerukus on peidetud OTS teenuse taga. Märkida tuleb ka seda, et vea tekkimisel OST teenus automaatselt võtab kõik muudatused tagasi ning tänu sellele *try-catch* plokkis puudub *rollback* direktiiv. Täpsemalt hajusatest transaktsioonidest oli räägitud peatükis *5*.

OTS teenus oli täitelikul moel kasutatud ainult PSS juures. JPA korral transaktsioonide juhtimine toimus CORBA serveri juures, ignoreerides OTS teenust (kuigi OTS meetodite väljakutsumine samuti toimus). JPA-ga saab kasutada ainult JTA-t (*Java Transaction API*), mis pakub CORBA OTS-ga analoogseid liideseid ja hajusate transaktsioonide läbiviimise funktsionaalsust. JTA baseerub OMG OTS spetsifikatsioonil ning *Java EE* raames realiseeritud teenus Java Transaction Service (JTS) on sisuliselt OTS liideste kujutamine *Java* keele ja *Java EE* tehnoloogia tarbeks. Tänu sellele enamus OTS realisatsioonidest pakuvad ka JTA toetust, selleks kasutatakse mähisklasse *(wrappers)*, mis transleerivad ja suunavad JTA väljakutsed OTS teenusele ja tagasi [JTA]. OpenORB OTS teenuse realisatsioon pakub JTA toetust (pakett *org.openorb.ots.jta*), kuid probleemiks sai asjaolu, et OpenORB dokumentatsioonis ei olnud midagi OTS konfigureerimisest JTA pakkujana. Samuti ei õnnestunud kiiresti leida juhiseid JPA kasutamisest OTS teenusega (*DataSource* objektide konfigureerimine jne.). Taoline dokumentatsioon (mõne teise CORBA realisatsiooni oma) ja juhised on kindlasti olemas, kuid nende ülesleidmine osutas liiga aeganõudvaks ja lõpuks oli otsustatud sellest loobuda. JPA korral ei kasutud hajusaid transaktsioone, kuid see ei olnud takistuseks, sest vajati ainult üht ressurssi korraga ja lokaalne transaktsioonide juhtime oli aktsepteeritav (reaalelu rakendustes aga ilmselt mitte).

6.10 Tulemused ja järeldused

Testrakendusega oli katsetatud PSS ja JPA tööd ja proovitud realiseerida sama funktsionaalsus

erinevate vahenditega. Suurem tähelepanu oli suunatud PSS teenusele.

CORBA pakub kõik võimalused hajusa arhitektuuriga rakenduse loomiseks. Selles töös oli vaadeldud ainult väike osa OMG poolt spetsifitseeritud võimalustest ja lisateenustest.

CORBA tehnoloogiat saab edukalt integreerida J2EE veebirakendusega. Samas siin on näha CORBA tehnoloogilist puudujääki – puuduvad võimalused luua veebirakendusi natiivsete vahenditega [CORBA_RISE_FALL].

Säilitavuse kasutamine oli peidetud vahekihtiga ja tänu sellele kliendiprogrammi on õnnestunud teha sõltumatuna konkreetsest säilitavuse varustajast. Rakenduses oli kasutatud ainult üks ressurss ja selle tõttu kõiki OTS teenuse võimalused ei õnnestunud vaadelda. Kuid sellist eesmärki ei olnudki. Samas ei olnud täielikul moel katsetatud JPA kasutamise aspektid CORBA rakendustes – näiteks POA aktiveerimise/deaktiveerimise mõjud. Selles võivad peita mõned probleemid.

Suur abi oli ka OpenORB-i lähtekoodist, mis aitas paremini mõista PSS tööd.

Võib nõustuda ka CORBA kriitikutega, kes väidavad, et suurimateks CORBA probleemideks on tänapäeval [CORBA_RISE_FALL], [WIKI_CORBA]:

- Keeruline ja läbipaistmatu spetsifitseerimisprotsess
- Soovituslikku realisatsiooni (*referential implementation*) puudus
- CORBA realisatsioonide mahajätmine OMG spetsifikatsioonidest
- Liiga keeruline komponenttehnoloogia (CORBA 3)
- Keeruline realisatsioonide valiku tegemine

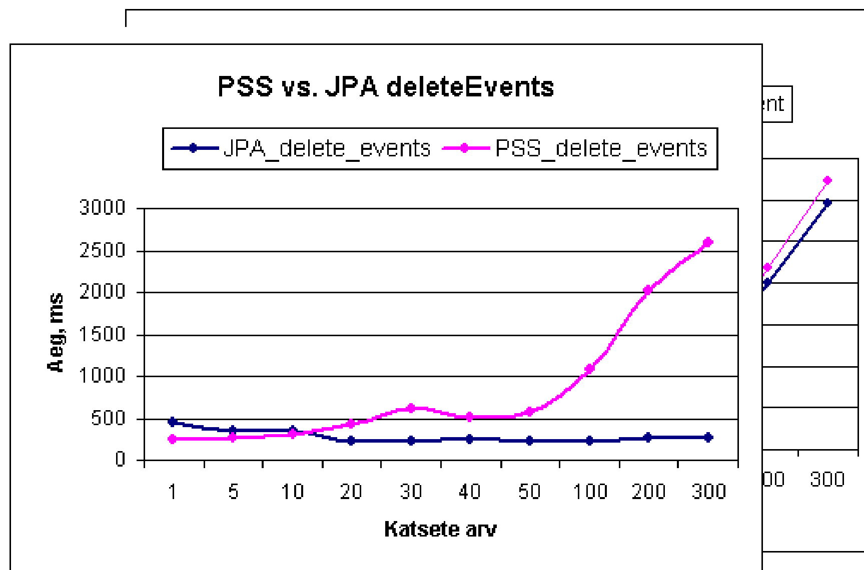
Jõudluse hindamine

Jõudluse hindamiseks olid tehtud testid erinevate katsete arvuga (*Event* ja *Bet* objektide loomine ja kustutamine, kasutades kasutajaliidest). Vahemikuks oli 1- 300 objekti (katset). Katsete arv 30 tähendab, et selle testi ajal tehti sama tegevust 30 objektiga. Algväärtuse (1) ja lõppväärtuse (300) ajakulud erinevad väga palju (umbes 200 korda). Ajakulu on mõõdetud millisekundites.

Objektide loomisel järjest lisatakse uusi objekte – meetodiga *create()*. Kustutamisel kasutatakse operatsiooni *deleteAll()*, mis kustutab kõik olemasolevad objektid (Sündmused või Panused) ühe operatsiooniga. Kasutajaliidese kirjeldus oli juba eelnevalt tekstis andud, lühike kasutajajuhend on antud lisana (*Lisa 1*).

Teste tehti konfiguratsiooniga, kus klient ja server töötasid samal masinal, kasutades võrgusuhtlemiseks *loopback* võrguliidest. Seega võrgu latentsust võib mitte arvestada.

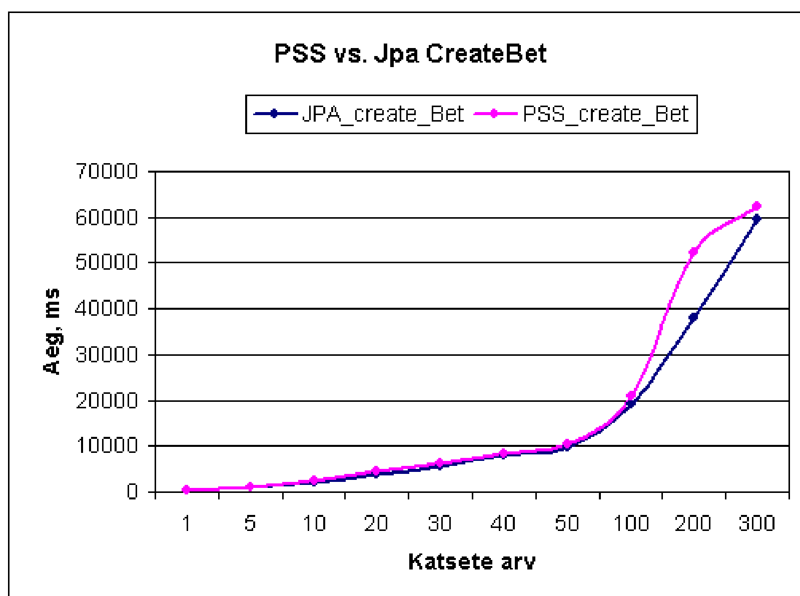
Testide tulemuste võrdemist on võimalik teha graafikute abil. Ühel graafikul on korraga PSS ja vastavate JPA katsete väärtused.



Joonis 6.5.: PSS vs. JPA - DeleteEvents

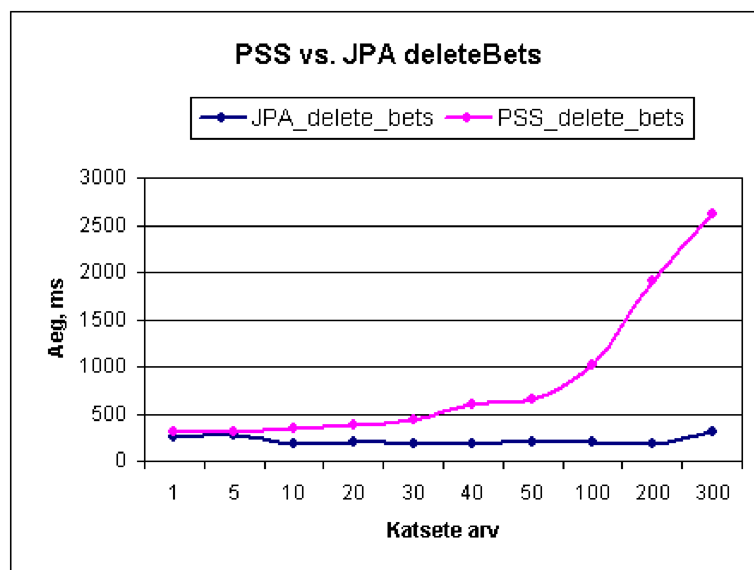
Esimesel graafikul on näidatud sõltuvus katsete arvust ja kulutatud aja vahel *Event* objekti loomisel. On näha, et PSS kasutamisel ajakulu on mõnevõrra suurem, kuid üldiselt see erinevus ei ole suur (10-15%).

Teisel graafikul on näidatud sõltuvus katsete arvust ja kulutatud aja vahel *Event* objektide kustutamisel. On näha, et JPA kasutamisel ajakulud ei sõltu objektide arvust, PSS kasutamisel aga sõltub. Selle põhjuseks on see, et JPA lubab kasutada SQL-ga sarnast operatsiooni *DELETE*, mille abil saab kustutada objekte, teadmata nende võtmete või identifikaatorite (PSS korral *pid*) väärtused. OMG spetsifikatsioon ning samuti kasutatud PSS-i realisatsioon sellist võimalust ei paku, seega iga objekt on kustutatud eraldi, mis mõjub jõudlusele. Sellele probleemile oli juba viidatud peatükis *3*.



Joonis 6.6.: PSS vs. JPA - CreateBet

Bet objektide loomisel on jälgitav sarnane pilt. Üldiselt suurt erinevust ei ole. Suurem erinevus on vahemikul 100-250, kuid see erinevus on suure tõenäosusega tingitud hetkeks tekkinud süsteemi lisakoormusega (näiteks opsüsteemi ülesanne vms.), sest teste tehti järjest (1, 5, 10, ... 300 PSS jaoks ja siis sama jada JPA jaoks).



Joonis 6.7.: PSS vs. JPA - DeleteBets

Event ja *Bet* objektide kustutamisel kehtivad samad sõltuvused.

Tuleb mainida, et objektide kustutamine toimub tunduvalt kiirem kui loomine ka PSS-i korral, siis kui kustutatakse iga objekt eraldi, kuid ühe kaugoperatsiooniga. 300 objekti juures nende objektide kustutamine võtab 25 korda vähem aega, kui nende objektide loomine kasutades 300

kaugoperatsiooni. JPA juures see erinevus on veelgi suurem – 266 korda, mis näitab, et väljakutsete arvu ja kuulutatud aja vahel sõltuvust ei ole.

Sellised tulemused selgelt viitavad sellele, et suurim osa ajast kulutatakse CORBA kliendi ja serverivahelisele suhtluse peale. Oli katsetatud variant, mille korral serveri poolel ei tehta midagi – lihtsalt kutsutakse välja kaugprotseduuri. Niimoodi saab hinnata aega, mis kulub suhtlemisele – valemiga *Suhtlemise aeg* = (*Tühi väljakutsumine*/*Väljakutsumine objektide loomisega*) Selgus, et sellisel juhul

- 200 objekti loomisel – suhtluse osakaal oli 86% (PSS)
- 100 objekti loomisel – suhtluse osakaal oli 77% (PSS)
- 50 objekti loomisel – suhtluse osakaal oli 72% (PSS)

Ainult ühe objekti loomisel suhtlemise osakaal oli keskmiselt 56% (PSS) ja 68% (JPA). Ühe objekti loomisel PSS ajakulu oli umbes 15% võrra suurem kui JPA-d kasutades. Testimise ajal klient ja server asusid samal masinal ja võrgusuhtlemise latentsust ei olnud. Kui kliendirakendus töötaks võrguga eraldatud arvutil, siis suhtlemise osakaal oleks veelgi suurem.

Seega suurimaks jõudluse probleemiks on intensiivne CORBA suhtlemine, jõudluse parandamiseks objektide loomisel oleks mõistlikum kutsuda välja üht protseduuri ja anda parameetrina massiiv andmetest. IDL liideste optimeerimisest oli räägitud esimeses peatükis. Samas selle konkreetse rakenduse tavalise kasutamise korral selline olukord, kus tekitakse korraga mitu objekti, on vähetõenäoline.

PSS-i kasutades 300 objekti kustutamine võttis 25 korda vähem aega kui loomine, kuigi serveri poolel tehti samuti 300 tegevust. Sellest tuleneb, et PSS sisemised meetodid võtavad mitu korda vähem aega kui kaugobjektide meetodid kliendi poolel. Seega PSS sisemised operatsioonid on palju optimaalsemad, kui testrakenduse omad. See on seletatav ka sellega, et PSS spetsifikatsiooni IDL liidesed ei kasuta parameetriteks keerulisi struktuure ja on üldiselt väga optimaalselt disainitud [PSS_SPEC]. Samas nagu oli juba mainitud peatükis *3*, PSS funktsionaalsust on võimalik kasutada ka CORBA kliendi poolel.

Järeldusena võib väita, et taolistes rakendustes, kus kaugprotseduuride parameetriteks on küllaltki suured struktuurid (umbes 10 välja), PSS teenuse kasutamine ei mängi väga kriitilist rolli jõudluse degradeerimisel. Sellisel juhul PSS-i meetodite peale kulub mitu korda vähem aega, kui rakenduse põhiliste kaugprotseduuride väljakutsumisele. JPA kasutamine peab olema üldiselt parema jõudlusega (vähem vahekihte), eriti operatsioonidel paljude objektidega, kui on

võimalik minimiseerida kasutatud meetodite arvu. Samas JPA kasutamine võib olla piiratud sellega, et soovitakse kasutada hajusaid transaktsioone, kuid valitud OTS teenuse realisatsioon ei paku JTA (*Java Transaction API*) liideste kasutamise võimalust. Säilitavuse kasutamine võib tugevalt mõjutada jõudlust siis, kui rakenduse IDL liidesed on juba piisavalt hästi optimeeritud.

Kokkuvõte

CORBA rakendustes tihti tekib vajadus kasutada säilitava olekuga objekte. Selline võimalus on olemas ja on spetsifitseeritud OMG poolt – see on *Persistant State Service* (PSS). Säilitavuse kasutamisel sellistes hajusas süsteemides, nagu CORBA rakendused, peab arvestama olemasolevate omapäradega.

Käesolev töö annab ülevaate CORBA tehnoloogiast, PSS teenusest ja selle kasutamisest. Antakse ka ülevaadet transaktsioonidest hajusates süsteemides ja vastavast OMG teenusest (OTS).

Võimalikku alternatiivina PSS-le on esitatud Java Persistance API tehnoloogia. Olid arutletud CORBA rakendustega integreerimise aspektid.

Töö praktiliseks osaks oli testrakenduse loomine. Rakendus on realiseeritud mitmekihilisena, kasutades erinevad tehnoloogiad, mitte ainult CORBA-t. Testrakenduse baasil oli läbiviidud mõningane CORBA tehnoloogia, PSS ja JPA analüüs. Järeldused baseeruvad ka teoreetilistel väidetel mis on esitatud töö esimeses osas.

Abstract

CORBA PSS

CORBA is a widely accepted middleware technology for building heterogeneous distributed software solutions. Many aspects of such systems are covered in OMG (Object Management Group) specifications. This document focuses on CORBA PSS (Persistent State Service). Firstly, an introductory overview of CORBA technology is given, explaining its purposes. Then the need for persistent objects in CORBA applications is discussed.

The main part of this thesis is dedicated to Persistent State Service (PSS). PSS architecture, PSDL persistence definition language are described. Some examples and typical usage of PSS are given, which are based on particular CORBA realization – OpenORB. Also the need for Query API is discussed, which is not part of OMG specification but found as proprietary extensions in some realizations. As topics on which PSS depends, CORBA Object Transaction Service (OTS) and distributed transactions are also discussed.

As a possible alternative for PSS another object-relational mapping technology is described – Java Persistence API (JPA). Some issues of using it with CORBA are given.

Practical part of this thesis consisted of developing a demo application, which uses both PSS and JPA functionality to provide object persistence. This application was implemented as J2EE Web application that uses remote CORBA server business logic. Based on this application, comparisons were made between PSS and JPA and some issues were pointed out, such as performance and usability. Both demo application and its source code are located on enclosed CD.

Mõisted, millele võiks leida sobiv eesti keele vaste.

Flush Manager -

Cache Manager -

featuurid-

Allikad

Internetti allikad on seisuga 25.05.2007

1) [PSS_SPEC] “Persistent State Service Specification”, Version 2.0. Object Management Group, September 2002
2) [OTS_SPEC] – “Transaction Service Specification”, Version 1.4. Object Management Group September 2003
3) [CCM_SPEC] – “CORBA Component Model Specification”, Version 4.0. Object Management Group, April 2006
4) [BOOK_CORBA] – Дирк Слама, Джейсон Гарбис, Перри Рассел “Корпоративные системы на основе CORBA”, издательский дом "Вильямс”, 2000. Tõlke
5) [BOOK_CORBA_PROGR] – Gerald Brose, Andreas Vogel, Keith Duddy "Java Programming with CORBA. Advanced Techniques for Building Distributed Applications", Third Edition, Wiley Computer Publishing, 2001
6) [JH_BK] – Jüri Harju, "Ülevaade CORBA serverist ja lisateenustest, Bakalaureusetöö", Tartu 2002
7) [JH_MAG] – Jüri Harju, "Turvaline CORBA nimeteenus, Magistritöö", Tartu 2005
8) [OPEN_ORB] – “The Community OpenORB”, Version 1.4 http://openorb.sourceforge.net/
9) [OPEN_ORB_DOC] – OpenORB 1.4 Dokumentatsioon. http://openorb.sourceforge.net/documentation1_4.html
10) [WIKI_CORBA] – Wikipedia artikkel “Common Object Request Broker Architecture”, http://en.wikipedia.org/wiki/Corba

11) [WIKI_REL] – Wikipedia artikkel "Relational model" http://en.wikipedia.org/wiki/Relational_model

12) [IONA_PRGUIDE] – "CORBA Programmer's Guide, C++", IONA Technologies PLC, http://www.iona.com/

13) [REL_OO] – Wikipedia artikkel "Object database", http://en.wikipedia.org/wiki/OODB

14) [PUDER] – "CORBA Product Profiles" http://www.puder.org/corba/matrix/

15) [SCHUPP_PSS] – Robert Schuppenies, "CORBA Persistent State Service" , 2004, Hasso-Plattner-Institute for Software Systems Engineering

16) [J2EE BOOK] – Елена Иванова, Максим Вершинин, "Java 2 Enterprise Edition, технологии проектирования и разработки", "БХВ-Петербург", 2003

17) [HIBERNATE] – Hibernate, http://www.hibernate.org/

18) [TOPLINK] – Oracle TopLink, http://www.oracle.com/technology/products/ias/toplink/index.html

19) [JDO] – Java Data Objects, http://java.sun.com/products/jdo/

20) [JAVADOC_ANNOTATIONS] – Java Annotatios, http://java.sun.com/j2se/1.5.0/docs/guide/language/annotations.html

21) [JAVA_EE] – Java EE, http://java.sun.com/javaee/

22) [J5EE_TUT] – Java EE Tutorial, "Introduction to the Java Persistence API", http://java.sun.com/javaee/5/docs/tutorial/

23) [JPA_INTR] - "The Java Persistence API - A Simpler Programming Model for Entity Persistence", http://java.sun.com/developer/technicalArticles/J2EE/jpa/

24) [2PC] – Wikipedia artikkel "Two-phase commit protocol", http://en.wikipedia.org/wiki/2PC

25) [CORBA_TRBL] – David Chappell, "The Trouble With CORBA", http://www.davidchappell.com/articles/article_Trouble_CORBA.html

26) [CORBA_RISE_FALL] – Michi Henning, "The Rise and Fall of CORBA" http://acmqueue.com/modules.php?name=Content&pa=showpage&pid=396

27) [RISE_FALL_RESPONSE] – "Response to 'The Rise and Fall of CORBA" http://www.orbzone.org/?p=121

28) [JTA] - “Transaction Services with JTA and JTS”, By Sams Publishing, http://www.developer.com/java/ent/article.php/10933_2224921_1

29) [UML] – Unified Modeling Language, http://www.uml.org/

30) [MYSQL] – MySql RDBMS, http://www.mysql.com/

31) [TOMCAT] - Apache Tomcat, http://tomcat.apache.org/

32) [JSF] - JavaServer Faces Technology, http://java.sun.com/javaee/javaserverfaces/

33) [ANT] – Apache Ant, http://ant.apache.org/

34) [NETBEANS] – NetBeans IDE, http://www.netbeans.org/

Lisad

Lisa 1: Kasutajajuhend

Rakenduse käivitamine

Nõuded keskkonnale: rakenduse käivitamine eeldab, et arvutis on olemas Java Runtime Envinronment 1.5 või uuem (arendusvahendite olemasolu ei ole ole vajalik). Lisaks on vaja, et *localhost* peal töötaks MySql andmebaasiserver. Täpsem info MySql keskkonda konfigureerimisest (kasutaja ja schema loomine) on olemas CD peal. Kõige mugavam rakenduse käivitamise viis on selle käivitamine pakutud skriptide abil. Teine võimalus on NetBeans IDE kasutamine – rakenduse realisatsioon on selle IDE projekti kujul (on vajalik *Visual Web Pack* lisapaketti olemasolu). Veebiserveri tarkvara (Apache Tomcat 5) on olemas samal kettal, samuti projektiehitamise vahend Apache Ant. Kõigepealt CD sisu tuleb kopeerida arvutisse (on vajalik kirjutamisõigus), sihtkohaks sobib suvaline kaust (NB! Kogu sisu maht on umbes 100MB).

Rakendus koosneb kahest osast – CORBA serverist ning J2EE veebirakendusest. Veebirakenduse käivitamise eelduseks on CORBA serveri valmisolek.

Üldine konfiguratsioon.

Failis "set_env.bat" (Windows) omistada muutujale JAVA_HOME korrektset väärtust (JRE asukoht). Luua MySql kasutajad – selleks kasutada skripti "create_users_and_schemas.sql".

Andmebaas

Tuleb käivitada andmebaasi serverit.

Server.

Käivitada skript "run_CRB_srv.bat" (Windows).

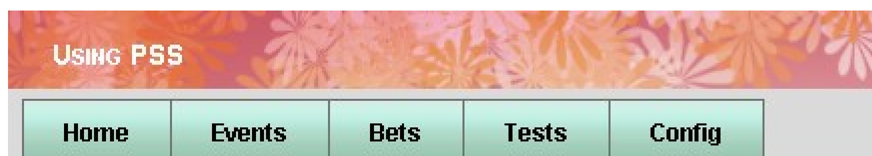
Veebirakendus.

Käivitada skript "run_tomcat.bat" (Windows). Testimiseks veebilehitsejas sisestada aadress

http://localhost:8080/PSS_totalisator/ .

Rakenduse kasutamine

Peale käivitamist rakendus asub aadressil http://localhost:8080/PSS_totalisator/ . Navigeerimiseks saab kasutada navigatsiooniriba. Selle kaudu on kättesaadavad tähtsamad veebilehed. Navigeerimisriba üleval on kirjas, mis säilitavuse allikas on parajasti kasutusel (PSS või JPA). Selle muutmiseks kasutage tabi “Config”.



Joonis 1.: Navigatsiooniriba

Sündmuste vaatamiseks kasutage tabi “Events”, nende loomiseks tabi “Home” > “Administrate Events”. Panuse loomiseks valige alguses tabil “Events” mõni konkreetne Sündmus (nupuga “Details”). Siis sellele Sündmusele saab omistada Panust (*Bet*).

Testimise funktsionaalsus on kättesaadav tabil “Tests”.

Mõned tegevused vajavad sisselogimist – selleks paremas ülemises nurgas on olemas vastav link “Login”. Sisselogimiseks kasutage eeldefineeritud kasutajat “*user*”, salasõnaks on samuti “*user*”.

Lisa 2: Testrakenduse IDL ja PSDL failide sisu

Asub CD peal – kaustas “PSS_totalisator/src/java”. Failide nimedeks on vastavalt “Intefaces.idl” ja “Persistance.psdl”.

Lisa 3: Mõnede CORBA realisatsioonide lisateenused

Abbrev.	Kirjeldus
AMI	Asynchronous Messaging Interface
CCM	CORBA Component Model
QoS	Quality of Service
FT	Fault tolerance
RT	Real time
PSS	Persistent State Service
CONC	Concurrency Service
PROP	Property Service
EVNT	Event Service
RLSH	Relationship Service
NAM	Naming Service
SEC	Security Service
TIME	Time Service
TRAD	Trading Service
NOTF	Notification Service
TRANS	Transaction Service

Profile	Core							Services									
	Min	AMI	CCM	QoS	FT	RT	PSS	CONC	PROP	EVNT	RLSH	NAM	SEC	TIME	TRAD	NOTF	TRANS
Borland Enterprise Server Visibroker Ed.	-	-	-	Y	Y	-	-	-	-	Y	-	Y	Y	-	-	Y	Y
BusinessWare	?	Y	?	?	?	?	?	?	?	?	?	Y	?	?	?	?	?
DSTC CORBA Services	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	Y	-
e*ORB C++ Edition	Y	Y	Y	?	Y	Y	Y	-	-	Y	-	Y	Y	Y	-	Y	?
e*ORB Java Edition	Y	-	-	-	P	-	?	?	?	?	?	Y	?	?	?	?	?
Egypt/Taiwan	Y	-	-	-	-	Y	Y	Y	Y	Y	-	Y	Y	Y	Y	Y	Y
EJCCM	-	-	Y	-	-	-	-	-	-	-	-	-	-	-	-	-	-
Eorb	Y	-	-	-	Y	Y	-	-	-	-	-	Y	-	-	-	Y	-
Gibraltars	Y	-	-	-	-	-	-	-	Y	Y	-	Y	Y	Y	Y	Y	-
GNU Classpath	-	-	-	P	P	P	-	-	-	-	-	Y	-	-	-	-	-
HP NonStop CORBA	-	P	-	Y	Y	-	-	-	P	Y	-	Y	P	P	P	P	Y
JacORB	-	Y	-	-	-	-	-	Y	-	Y	-	Y	P	-	Y	Y	Y
Java 2 Standard Edition 1.4.1	?	?	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	Y
Jonathan	-	Y	P	-	-	-	P	-	-	Y	-	Y	-	-	-	-	P
Mexico/Jordan	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y
MICO	Y	P	Y	-	P	-	-	-	Y	Y	Y	Y	Y	Y	Y	P	-
mico/E	-	-	P	-	-	-	P	-	P	Y	P	Y	-	-	Y	P	-
microORB	Y	Y	-	Y	-	Y	-	-	-	-	-	Y	-	-	Y	P	-
MiddCor	-	-	-	-	-	-	?	?	?	Y	?	Y	?	?	?	Y	?
Omniorb	-	P	-	-	-	-	-	-	-	Y	-	Y	-	-	-	Y	-
omniORB 3	-	-	-	-	-	-	-	-	-	Y	-	Y	-	-	-	Y	-
omniORB 4.0 preview	-	P	-	-	-	-	-	-	-	Y	-	Y	-	-	-	Y	-
Openfusion TCS (Total CORBA Solution)	Y	Y	P	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	P
OpenORB	-	-	-	-	P	-	Y	Y	Y	Y	-	Y	-	Y	Y	Y	Y

Orb2	P	P	-	P	P	-	-	-	-	Y	-	Y	Y	-	Y	-	-
Orbacus	-	Y	-	Y	-	-	-	-	-	Y	-	Y	-	-	-	Y	-
ORBacus 4	-	P	-	-	-	-	-	-	Y	Y	-	Y	-	Y	Y	Y	Y
ORBacus/E 1.1	-	-	-	-	-	-	-	-	Y	Y	-	Y	-	Y	-	-	-
Orbexpress GT for C++	Y	Y	P	Y	-	-	-	-	-	Y	-	Y	P	P	-	P	-
Orbexpress RT for Ada	Y	Y	-	Y	P	Y	-	-	-	Y	-	Y	P	P	-	-	-
Orbexpress ST for Ada	Y	Y	-	Y	-	-	-	-	-	Y	-	Y	P	P	-	-	-
Orbexpress ST for C++	Y	Y	P	Y	-	-	-	-	-	Y	-	Y	P	P	-	P	-
Orbit2 for Red Hat Linux 8.0	?	Y	-	-	?	-	-	P	-	P	-	Y	-	-	-	-	P
Orbix 2000	-	Y	-	Y	Y	-	Y	-	-	Y	-	Y	-	-	Y	Y	Y
Orbix E2A Application Server Platform	-	Y	-	Y	Y	-	Y	-	-	Y	-	Y	Y	-	Y	Y	Y
Orbix/E 2.0	-	-	-	-	-	-	-	-	P	P	-	Y	-	-	-	-	-
Orbriver	P	-	P	-	P	-	-	-	-	Y	-	Y	P	P	-	P	-
Orbriver RT	P	P	P	P	P	P	-	-	-	Y	-	Y	P	P	-	P	-
PolyORB	-	-	-	P	P	Y	-	-	-	Y	-	Y	P	Y	-	Y	-
Sankhya Varadhi	Y	P	-	P	Y	P	P	-	-	Y	-	Y	Y	-	P	P	-
SmalltalkBroker	-	Y	-	-	-	-	-	Y	-	Y	-	Y	-	-	-	-	Y
Tao	Y	Y	Y	Y	Y	Y	Y	Y	Y	Y	-	Y	Y	Y	Y	Y	Y
Tao (The ACE ORB)	Y	Y	P	Y	P	Y	P	Y	Y	Y	-	Y	Y	Y	Y	Y	P
The ACE ORB (TAO) 1.1a	Y	Y	P	Y	P	P	-	Y	Y	Y	-	Y	P	Y	Y	Y	-
The Community Openorb	-	P	-	P	P	-	Y	Y	Y	Y	-	Y	-	Y	Y	Y	Y
Tuxedo 8.0	-	Y	-	Y	Y	-	-	Y	-	Y	-	Y	Y	-	-	Y	Y
VBOrb	-	-	-	-	-	-	-	-	-	-	-	Y	-	-	-	-	-
VisiBroker	-	-	Y	Y	Y	-	Y	-	-	Y	-	Y	Y	-	-	Y	Y
Visibroker RT (RealTime)	Y	Y	-	-	-	Y	-	-	?	Y	-	Y	-	-	P	-	-

Lisa 2: Erinevate CORBA realisatsioonide lisateenused (NB! andmed seisuga 15-05-2004)

Lisa 4: JPA kasutamise näide

```
import javax.persistence.*;
public class JpaTest{
	public static void main(String [] args){
		// Vabriku loomine, JpaTestPU on eelnevalt defineeritud Persistence Unit
		// (failis persistance.xml)
		// Persistence Unit´i defineerimisel konfigureeritakse andmebaasi URL jne.
		EntityManagerFactory emf =
Persistence.createEntityManagerFactory("JpaTestPU");
		EntityManager em = emf.createEntityManager();
		// objekti loomine
		JpaUser u = new JpaUser();
		u.setName("User");
		try {
			em.getTransaction().begin();
			// objekti säilitamine
			em.persist(u);
			em.getTransaction().commit();
		} catch (Exception ex) {
			em.getTransaction().rollback();
		}
	}
}
```

Lisa 5: Rakenduse lähtekood

Asub CD peal – kaustas “PSS_totalisator/src/java”.