Markus Lappi

J2EE CONNECTOR ARCHITECTURE YRITYKSEN SOVELLUSTEN INTEGROINNISSA

Tietojärjestelmätieteen
Pro gradu
21.12.2002

Jyväskylän yliopisto
Tietojenkäsittelytieteiden laitos
Jyväskylä

TIIVISTELMÄ

Lappi, Markus Tuomas
J2EE Connector Architecture yrityksen sovellusten integroinnissa / Lappi
Jyväskylä: Jyväskylän yliopisto, 2002.
134 s.
Pro gradu -tutkielma

Tässä tutkielmassa tarkastellaan yritysten sovellusten integrointiin soveltuvia arkkitehtuureita, teknologioita ja selvitetään millä eri teknisillä tasoilla yrityksen sovelluksia voidaan integroida. Teknologioista tarkempaan tarkasteluun on valittu Java 2 Enterprise Edition -ohjelmointialustan sisältämä J2EE Connector Architecture (JCA). JCA:n käyttöä on havainnollistettu tutkielmaa varten ohjelmoidulla esimerkkitoteutuksella.

Yrityksen sovellusten integroinnin perimmäinen ongelma on yrityksissä olevien järjestelmien välinen kommunikointikyvyn puute. Tarvetta sovellusten väliselle kommunikoinnille on aiheuttanut liiketoiminnalliset muutokset, siirtyminen elektroniseen liiketoimintaan ja yrityksen arvoketjuun muutokset kilpailukyvyn säilyttämiseksi tai saavuttamiseksi. J2EE- ja Java-sovelluksien osalta integroinnin ongelmaan helpottaa JCA-määrittely, joka tarjoaa standardin käytännön sovelluksien väliselle kommunikoinnille.

AVAINSANAT: EAI, väliohjelmistot, yritysten sovellusten integrointi, Java, J2EE, J2EE Connector Architecture, JCA

SISÄLLYSLUETTELO

1 JOHDANTO 5
1.1 Tutkimuksen tausta 5
1.2 Tutkimusongelma ja rajaus 6
1.3 Tutkimuksen rakenne ja kulku 7

2 YRITYKSEN SOVELLUSTEN INTEGROINTI 8
2.1 Yleistä sovellusten integroinnista 9
2.2. Integroinnin tasot 11
2.2.1 Datan Integrointi 12
2.2.2 Sovellusliittymätason integrointi 15
2.2.3 Metoditason integrointi 17
2.2.4 Käyttöliittymätason integrointi 19
2.2.5 Sovellusten integraation tasot liiketoiminnan näkökulmasta 20
2.3 Yhteenveto integroinnista 23

3. EAI JA VÄLIOHJELMISTOT 25
3.1. Väliohjelmistoarkkitehtuurit 26
3.1.1 Pisteestä pisteeseen -arkkitehtuuri 26
3.1.2 Monesta moneen -arkkitehtuuri 27
3.1.3 Sovittimet sovellusten integraatiossa 29
3.2 Väliohjelmistotekniikoita 30
3.2.1 Etäkutsut 30
3.2.2 Sanomapohjainen väliohjelmisto 31
3.2.3 Sanomanvälittäjät 35
3.2.4 Tapahtumapohjaiset väliohjelmistot 36
3.2.5 Hajautetut oliot 38
3.2.5 Tietokantaorientoituneet väliohjelmistot 40
3.3 Yhteenveto väliohjelmistoista 42

4. JAVA 2 ENTERPRISE EDITION 44
4.1 Java 44
4.2 Yleistä J2EE-ohjelmointialustasta 46
4.3 Javan nimi- ja hakemistoliittymät 51
4.4 Javan tietokantaliittymä 53
4.5 Javan tapahtumanhallinnan liittymät 55
4.6 Enterprise JavaBeans -komponenttimalli 57
4.7 Yhteenveto J2EE-alustasta 60

5. J2EE CONNECTOR ARCHITECTURE 62
5.1 JCA yleisesti 63
5.1 Roolit JCA-pohjaisessa sovelluskehityksessä 65
5.2 Järjestelmä- ja sovellustason sopimukset 66

5.5 Tapahtumien hallinta ... 73
5.6 Tietoturvan hallinta ... 76
5.7 Yleinen asiakasliittymä ... 78
5.8 Yhteenveto J2EE Connector Architecturesta ... 79

6. RESURSSISOVITTIMEN RAKENTAMINEN ... 81
6.1 Resurssisovittimen liittymät ... 81
6.2 Konstruktion ympäristö ja toiminnallisuus ... 85
6.3 Resurssisovittimen konstruointi ... 87
6.4 Yhteenveto resurssisovittimen rakentamisesta ... 89

7. POHDINTA ... 90
7.1 Miksi JCA? ... 90
7.2 JCA:n sijoittuminen integrointiarkkitehtuureihin ja -malleihin ... 91
7.3 JCA:n tavoitteet ... 92
7.4 JCA:n heikkouksia ... 94
7.5 Muita huomioita ... 95

8. YHTEENVETO ... 96

LÄHDELUETTELO ... 100

LIITE 1 ... 104

1 JOHDANTO

1.1 Tutkimuksen tausta

Tällä hetkellä IT-alalla vallitsee kolme suurta suuntausta: Internet, ohjelmistopaketit ja sovellusten integraatio. Internet-teknologiat tarjoavat mahdollisuuden yhdistää yrityksen asiakkaat, alihankkijat, liikekumppanit ja sisäiset käyttäjät. Eräät ohjelmistopaketit, kuten eri Enterprise Resource Planning (ERP) -järjestelmät (esimerkiksi SAP R/3) tarjoavat integroitavaa ympäristöä tukemaan liiketoimintaprosesseja. ERP-järjestelmät tukevat taustajärjestelmiä (back office), kun taas toiset ohjelmistopaketit tukevat edustajärjestelmiä (front office), kuten esimerkiksi asiakaspalvelua. Sovellusintegrointia tarvitaan: yhdistämään tausta- ja edustajärjestelmät, siirtämään liiketoimintaprosessia verkkoon, sekä laajentamaan toimitusketjua kattamaan asiakkaat, toimittajat ja kumppanit. Sovellusintegraatiota tarvitaan myös tuomaan tietoa vanhoista operationaalisista järjestelmistä (legacy systems)' uuteen ympäristöön. (Johannesson & Persons, 2000, 212)

Organisaatioihin on kehitetty moniin eri tarkoituksiin hyvin erilaisia sovelluksia, joiden välinen kommunikointi ei ole ollut alun perin tarpeellista. Esimerkiksi organisaatiossa saattaa olla kymmeniä erityyppisiä avoimia ja sovelluskohtaisia järjestelmiä, jotka on kehitetty eri alustoille ja eri teknologioita käyttäen. Myöhemmin yrityksien liiketoimintaprosessien muutokset ja tehokkuuteen tähtäävät vaatimukset ovat asettaneet myös uusia vaatimuksia sovelluksille. Kaikkea ei kannata kehittää uudestaan, joten yksi vastaus näille vaatimuksille on sovellusten välinen kommunikointi, ja juuri tähän tarvitaan sovellusten integraatiota.

Sun Microsystems julkisti vuoden 1999 lopussa uuden Java 2 -määrityksen. Sun on ottanut käyttöön uuden kolmitasoisen jaon eri ohjelmointialustoilleen. Aikaisemmin nimellä JDK (Java Developmet Kit) tunnettu Java-kehitysympäristö on nykyään Java 2 Standard Edition (J2SE), joka sisältää kaikki tavallisten Java-ohjelmien ja -sovelmien toteuttamiseen tarvittavat sovellusliittymät. Java 2 Micro Edition (J2ME) on vastaavasti erilaisiin mobileihin ja pieniin laitteisiin, esimerkiksi kämmenmikroihin tarkoitettu Java-ohjelmointialustan määritys, joka sisältää vain erittäin rajoitetun osajoukon kaikista Javan ominaisuuksista. EAI:n kannalta mielenkiintoisin näistä ohjelmointialustoista on Java 2 Enterprise Edition (J2EE), joka sisältää kaikki hajautettujen liiketoimintaa tukevien järjestelmien toteuttamiseen tarvittavat sovellusliittymät. J2EE:n tarkoituksena on nimensä mukaisesti pyrkiä vastaamaan yritystason järjestelmien asettamiin vaatimuksiin.

Syksyllä 2001 Sun Microsystems Inc. julkaisi uuden 1.3 version J2EE:stä, joka tuo mukanaan uuden JCA-määrittelyn (J2EE Connector Architecture). JCA mahdollistaa J2EE-sovellusten yhteyden ottamisen sovittimien avulla olemassa oleviin tietojärjestelmiin (esimerkiksi SAP R/3 tai IBM CICS).

Evan Data Corporation tutkimusyhtiön mukaan Java-kehittäjiä on vuonna 2002 enemmän kuin C/C++-kehittäjiä, joten Javan saama suuri suosio jatkuu edelleen. Yhä useampi sovellus kehitetään Javalla ja J2EE-alustalle: samalla kasvaa tarve saada uudet Java-sovellukset toimimaan yhteistyössä yrityksen vanhojen sovellusten kanssa. JCA-määrittely tarjoaa vastauksia juuri tällaisiin integraatio-ongelmiin.

1.2 Tutkimusongelma ja rajaus

Tässä tutkimuksessa tarkastellaan yrityksen sovellusten integrointia ja erityisesti J2EE Connector Architecturen -määrityksen käyttöä sovellusten integroinnissa.

Jatkossa tästä määrityksestä käytetään lyhennettä JCA. Integroinnin voidaan katsoa jakautuvan kolmelle käsitteelliselle tasolle: tekniselle (toteutukset), loogiselle (integraatioarkkitehtuurit) ja liiketoiminnan tasolle. Lähtökohtana integroinnille on usein yrityksen liiketoiminnan muutoksien tuomat odotukset tietojärjestelmille ja niiden mallintaminen, mutta tässä tutkielmassa pääpaino on integroinnin loogisella ja teknisellä tasolla.

Tutkimusongelmat:

- Mitä mahdollisuuksia JCA tarjoaa yritysten sovellusten integraatioon?
- Mitä menetelmiä ja tekniikoita on esitetty ratkaisuksi sovellusintegraatiolle?

Tutkielman tavoitteena on kuvata kirjallisuudessa esitettyjä sovellusintegrointiin soveltuvia arkkitehtuurimalleja ja menetelmiä sekä selvittää, miten JCA:ta voidaan käyttää yrityksen sovellusten integroinnissa.

Tämä tutkielma on tyypiltään käsitteellis-konstruktiivinen.

1.3 Tutkimuksen rakenne ja kulku

Tutkielman toisessa luvussa selvitetään yrityksen sovellusten integrointiin liittyviä käsitteitä sekä integroinnin eri tasoja teknisestä ja liiketoiminnan näkökulmasta. Kolmannessa luvussa selvitetään yrityksen sovellusten integrointiin liittyviä arkkitehtuureja ja tekniikoita. Neljännessä luvussa esitellään J2EE-sovellusalustaa yleisesti. Viidennessä luvussa perehdytään tarkemmin JCA-määrittelyyn. Kuudennessa luvussa kuvataan konstruktion avulla JCA:n käyttöä sovellusten integroinnissa. Seitsemännessä luvussa analysoidaan JCA:ta ja tutkielman päättää kahdeksas luku, joka sisältää yhteenvedon.

2 YRITYKSEN SOVELLUSTEN INTEGROINTI

Tämä luvun tarkoituksena on selvittää yrityksen sovellusten integroinnin (Enterprise Application Integration eli EAI) keskeisiä käsitteitä, kuvata EAI:n ongelmaa yleisesti ja esitellä EAI:n eri tasoja.

Koska EAI:ssa on kyse sovellusten integroinnista, on myös hyvin oleellista määritellä ohjelmiston (software) ja yrityksen sovelluksen käsitteet. Pressman (1997, s. 10) on esittänyt seuraavan määritelmän ohjelmistolle: "Ohjelmistot ovat (1) käskyjä (tietokone ohjelmia), jotka suoritettaessa tuottavat halutut toiminnot halutulla suoritusteholla, (2) tietorakenteita, jotka mahdollistavat ohjelmien manipuloida informaatiota toivotulla tavalla, ja (3) dokumentteja, jotka kuvaavat ohjelmien toiminnot ja käyttön". Sovelluksella (application) ja ohjelmistolla (software) voidaan tarkoittaa lähes samaa asiaa, mutta Pressmanin (1997) mukaan sovelluksen luonnetta määriteltäessä informaation sisältö on merkitsevämpi tekijä. Tässä informaation sisällöllä tarkoitetaan sovelluksen syöte- ja tulosinformaation muotoa. Pressmanin (1997) esittämä käsite liiketoiminnan ohjelmistosta (Business Software) vastaa hyvin tässä tutkielmassa käsiteltävää yrityksen sovelluksen käsitettä. Liiketoiminnan ohjelmistolla tarkoitetaan yrityksien erillisiä järjestelmiä (esimerkiksi varastonhallinnan ja palkanlaskennan sovellukset) tai näistä kehittyneitä laajoja johdon tietojärjestelmiä, jotka käyttävät useita laajoja liiketoimintaan liittyvää tietoa sisältäviä tietokantoja. Tällaiset sovellukset käsittelevät olemassa olevaa dataa siten, että siitä muodostuu liiketoiminnan toimintoja ja johdon päätöksiä helpottavaa informaatiota.

EAI:n tarkoituksena on yhdistää yrityksen sisäiset sovellukset ja tietovarastot liiketoimintaprosessien ja tiedon jakamiseksi. Tästä tulisi suoriutua ilman suurempia ohjelmistojen ja tietorakenteiden muutoksia. (Linthicum 2000, 1-3)

EAI:n yhteydessä puhutaan usein järjestelmä- tai sovellusintegraatiosta, mutta näillä käsitteillä voidaan myös tarkoittaa yrityksien välisten sovellusten ja jakeluketjujen integrointia. EAI:ssa ja toimitusketjujen sovellusten integroinnissa käytetään samoja arkkitehtuureja ja teknologioita, mutta eroina ovat jakeluketjujen sovellusten teknologioiden suuremmat eroavaisuudet sekä osapuolten erilaiset toimintatavat ja liiketoimintatavoitteet. Stavridounia (1999, 91) ja Kuhnia (1990) mukaillen järjestelmäintegroinnilla voidaan tarkoittaa organisaation alijärjestelmien (ohjelmistojen ja/tai laitteistojen) yhdistämistä siten, että saadaan tyydytettyä sellaisia organisaation tarpeita, joihin yksitäiset järjestelmät eivät pysty. Ruh, Maginnis ja Brown (2001, 2) määrittelevät EAI:n seuraavasti: ”EAI on uusien strategisten liiketoimintaratkaisujen luomista yhdistelemällä toiminnallisuuksia yrityksen olemassa olevien sovellusten, kaupallisten paketoitujen sovellusten ja uuden koodin välillä käyttämällä yleistä väliohjelmistoa”.

EAI-käsite keskittyy siis yhteen yritykseen ja sen sisäisten sovellusten integrointiin. Järjestelmäintegraatio-termi puolestaan voi kattaa myös eri yrityksien välisten sovellusten integroinnin. EAI ottaa huomioon yrityksen liiketoimintaprosessit, tiedon, näiden kahden välisen suhteen ja sen, kuinka nämä ilmenevät yrityksessä. Väliohjelmistoarkkitehtuurit ja -tekniikat ovat myös keskeinen osa EAI:ta. Vaikka EAI on monille tuntematon käsite, se ei ole silti pelkkä uusi kolmikirjaiminen lyhenne IT-maailmassa; organisaatiot ovat etsineet tapoja integroida sovelluksia jo vuosikausia (Vander Hey 2000, 38).

2.1 Yleistä sovellusten integroinnista

Suurin osa organisaatioista käyttää toiminnassaan useampia tietojärjestelmiä. Tietojärjestelmien yhteistoiminta on kuitenkin edelleen puutteellista. Vasta viime vuosina on kiinnitetty enemmän huomiota siihen, miten yrityksen toimintaa voidaan kehittää parantamalla tietojärjestelmien välistä yhteistyötä. Usein

systeemit yhdistetään vain tietojen vaihdolla erikoisratkaisuin. Kuitenkin asiaan liittyy paljon muutakin kuin tietojen vaihtoa. Integraation merkitys lisääntyy, kun yhä kriittisempiä järjestelmiä yhdistetään toimimaan keskenään. EAI:n perusongelma on, miten nämä järjestelmät saadaan integroitua.

Yritykset ovat usein perinteisesti jakautuneet toiminnallisiin osastoihin, kuten tuotanto, markkinointi, asiakashallinta ja palkanlaskenta. Myös yrityksien tarvitsemat tietojärjestelmät ja sovellukset on suunniteltu ja kehitetty vastaamaan vain näiden erillisten funktionaalisten osastojen tarpeita. Myöhemmin yritykset ovat kuitenkin pyrkineet suoraviivaistamaan liiketoimintaprosessejaan siirtymällä funktionaalisesta mallista prosessimalliin sekä rikkomaan osastojen välisiä rajoja. Tällainen liiketoiminnallinen muutos tarvitsee myös vahvan tuen tietojärjestelmiltä. Yrityksien järjestelmät on saatettu kehittää eri vuosikymmenillä, joten on luonnollista, että on käytetty eri teknologioita, ohjelmointikieliä, protokollia ja alustoja. Järjestelmät on myös tarkoitettu tukemaan liiketoimintaa vain sillä funktionaalisella alueella, johon järjestelmät on alun perin suunniteltu. Juuri nämä asiat tekevät EAI:n tarpeelliseksi, mutta samalla haastavaksi ja vaikeaksi. (Wangler & Paheerathan 1990, 79-80)

Tarve EAI:lle on kasvanut monista eri syistä. Eräs merkittävä syy on yrityksien siirtyminen käyttämään hyväksi elektronista liiketoimintaa. Porter (1985, 51-60) on esittänyt mallin yrityksen arvoketjusta. Porter on jakanut yrityksen toiminnot kahteen osaan: tukitoiminnot ja perustoiminnot. Tukitoiminnot kattavat yrityksen infrastruktuurin, inhimillisten voimavarojen hallinnan, tekniikan kehityksen ja hankinnan. Perustoimintoihin kuuluu tulologistiikka, tuotanto-operaatiot, lähtölogistiikka, myynti ja markkinointi sekä huolto. Näiden eri osien kulujen ja tuottojen erotuksesta saadaan yrityksen kate. Luonnollisesti yritykset yrittävät vähentää kuluja, maksimoida tuottoa ja parantaa kilpailukykyä, mikä johtaa muutoksiin myös arvoketjussa. Arvoketjun optimoinnissa voidaan hyödyntää elektronista liiketoimintaa. Esimerkiksi tuotteita voidaan markkinoida ja myydä

Internetissä suoraan loppukäyttäjille, jolloin säästytään toimitusketjussa välikäsien aiheuttamilta kustannuksilta. Yritysten ja sen kumppaneiden arvoketjua voidaan myös ajatella integroituna järjestelmänä, jossa raaka-ainetoimittajan lähtölogistiikka on yrityksen tulologistiikka ja yrityksen lähtölogistiikka on asiakkaan tulologistiikka. Yrityksellä ja sen kumppaneilla ei aina ole yhteistä tietojärjestelmää, mutta silti tästä logistisesta ketjusta pitäisi saada toimiva, joten tarvitaan eri tietojärjestelmien yhteistoimintaa eli järjestelmien integrointia.

2.2. Integroinnin tasot

Yrityksen sovellusten integrointia voidaan tehdä monella eri tasolla. Linthicum (2000) on jakanut EAI:n neljään eri tasoon: data-, sovellusliittymä-, metodi- ja käyttöliittymätaso. Näitä tasoja tarkastellaan tarkemmin seuraavissa kappaleissa.

Wangler ja Paheerathan (2000, 81-82) ovat puolestaan määritelleet sovellusten integroinnille kuusi eri lähestymistapaa markkinoilla olevien ohjelmistotyökalujen ja -teknologioiden perusteella:

1. Alustan integrointi käyttäen eri väliohjelmistoteknologioita, kuten etäkutsuja (Remote Procedure Call eli RCP), sanomanvälittäjiä (Message Brokers), CORBA-olioita (Common Object Request Broker Architecture). Väliohjelmistotekniikkoja käsitellään tarkemmin omassa luvussaan.
2. Datan integrointi käyttäen eri tietokannoissa käytettävää yleistä tietokantojen SQL-kyselykieltä (Structured Query Language) tai tiedon integrointiin valmistettuja työkaluja.
3. ERP-järjestelmien ja perinnejärjestelmien (legacy systems) komponenttien integrointi käyttäen hyväksi sovelluspalvelimien tarjoamia palveluja, kuten tapahtumienhallintaa, tietokantayhteyksien varastoa (connection pool) ja istunnon hallintaa.

4. Sovellusten integrointi sovittimilla (adapters). Käytetään esimerkiksi ERP-järjestelmiin rakennettuja sovittimia, jotka tarjoavat ERP-järjestelmän sisältämiä palveluja sovellusliittymätasolla.
5. Liiketoimintaprosessien integrointi. Tämä on abstrakti ja korkean tason lähestymistapa, jossa on keskeisessä osassa liiketoimintaprosessien mallintaminen sellaisilla väliohjelmistotuotteilla, jotka tukevat tällaisia toimintoja.
6. Yrityksien välinen (business to business) integrointi. Yrityksien välinen integrointi tukee laajentunutta yrityskonseptia mahdollistamalla sovellusten integroinnin yli organisaatiorajojen. Esimerkiksi yhdistetään alihankkijoiden ja asiakkaiden järjestelmät yrityksen omiin järjestelmiin.

2.2.1 Datan Integrointi

Sovellusten integroinnissa (etenkin datatasolla) tiedon ja informaation hankkiminen ovat juuri se asia, miksi integraatiota tehdään, joten mielestäni on oleellista selvittää datan käsite.

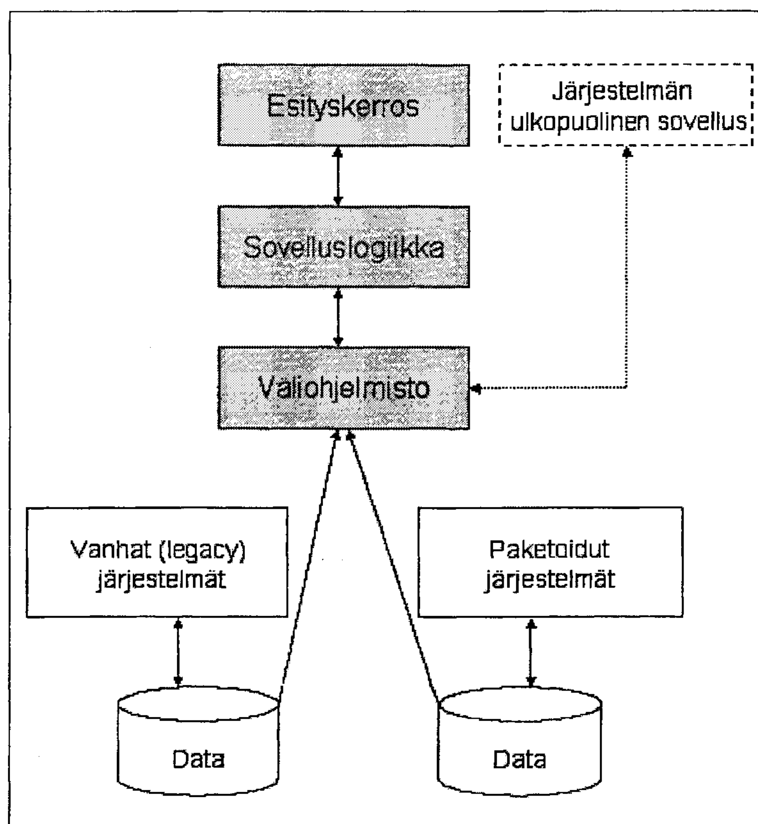
Poutsaari ja Holopainen (1996, 14) ovat määritelleet datan seuraavasti: "Data on tiedon säännönmukainen esitys säilytettävässä, viestitettävässä tai käsittelykelpoisessa muodossa". Hirschheim, Klein ja Lyytinen (1995, 12) määrittelevät datan seuraavasti: "*Data* on muuttumaton suure, joka sisältää mahdollisen merkityksen ihmiselle hänen tulkintansa kautta". Tässä tutkielmassa datalla käsitetään ainoastaan tietokoneellisesti käsiteltävää tietoa.

Datatason EAI on kokonainen prosessi (ja tekniikat sekä teknologiat) datan siirtämiseen tietovarastojen välillä. Yksinkertaistetusti dataa siis haetaan yhdestä tietokannasta, tarvittaessa muokataan ja tallennetaan toiseen tietokantaan. Suurin etu tässä on se, että olemassa olevia sovelluksia ei tarvitse muokata, vaan riittää,

että saadaan yhteys tietokantoihin. (Johannesson, Jayaweera ja Wangler, 2000, 160)

Datatason lähestyminen toimii parhaiten silloin, kun integroinnin kohteena on yksinkertainen datan ja informaation jakaminen, esimerkiksi jos järjestelmästä tarvitsee saada haettua vain asiakas- tai myyntitietoja. Datatason integraatio on kypsynein integraatiomenetelmä, mikä näkyy myös sitä tukevien markkinoilla olevien tuotteiden saatavuudessa. Datatason integroinnin tueksi löytyy nykyään melko kattavasti ohjelmistoja. Esimerkiksi sellaiset suuret tietokantatoimittajat kuten Informix/Ardent, Oracle/Carleton, Smart DB Crop. ovat esitelleet tuotteita, jotka tukevat datatason integrointia. Näillä tuotteilla voidaan hakea, muuntaa ja siirtää tietoa eri tietokantojen välillä. Muihin integrointitasojen ratkaisuihin nähden datatason integrointi on edullista, suhteellisen riskitöntä ja käytännöllistä. Datatason ratkaisut ovat hyviä silloin kun integroinnin tarve on hyvin kevyt, mutta monimutkaisemman informaation tai liiketoimintaprosessien jakamiseen tarvitaan muita ratkaisuja. (Vander Hey 2000, 38)

Ruh, Maginnis, Brown (2001 17-27) ovat esitelleet datatason integrointimallin, joka perustuu suoraan yhteyteen integroitavien sovellusten käyttämiin tietokantoihin. Tässä mallissa saadaan väliohjelmistolta (middleware) suora yhteys sovellusten tietokantoihin ja tietorakenteisiin ohittaen esitys- ja sovelluslogiikkakerrokset (KUVIO 1).



KUVIO 1. Datatason integrointimalli (Ruh ym. 2001, 24)

Datatason integrointimallia voidaan käyttää, esimerkiksi jos integroidaan datatasolla kolme eri sovellusta: asiakkaan tilausjärjestelmä, asiakkaan laskutusjärjestelmä ja tiedonlouhintasovellus (data mining application), jolla kerätään esimerkiksi tietoa asiakkaiden kulutustottumuksista. Tässä mallissa tiedonlouhintasovellus käyttää väliohjelmistoa, joka tarjoaa suoran yhteyden myynti- ja laskutusjärjestelmien tietokantoihin. Väliohjelmisto tekee myynti- ja laskutusjärjestelmistä "läpinäkyviä" tiedonlouhintasovellukselle eli tiedonlouhintasovelluksen ei tarvitse olla tietoinen muiden sovellusten sovelluslogiikasta tai edes siitä, minkä toimittajan tietokantoja ne käyttävät. Tiedonlouhintasovellukselle riittävät väliohjelmiston tarjoamat palvelut tiedon operointiin. (Ruh ym. 2001, 19)

Vaikka datatason integrointi on mallina yksinkertainen, sen toteutus ei ole välttämättä helppoa. Järjestelmien kehittäjien tulee tuntea yhtälailla niin

tietokantojen monimutkainen maailman kuin tietovirtojen kulku yrityksessä. Yrityksissä saattaa olla satoja tai jopa tuhansia tietokantoja, jotka on toteutettu eri alustoille ja eri teknologioilla. Tällaisten monimutkaiset laajat heterogeeniset ympäristöt tekevät integroinnista vaikeata yksikertaisilla malleillakin. (Linthicum 2000, 23-24)

2.2.2 Sovellusliittymätason integrointi

Sovellusliittymätason EAI:ssa käytetään sovellusten tarjoamia liittymiä integroinnin saavuttamiseksi. Sovellusliittymätason integrointiin liittyy oleellisesti sovellusliittymän (application programming interface eli API) käsite.

Sovellusliittymät ovat hyvin määriteltyjä mekanismeja, joilla saadaan yhteys jonkinlaiseen resurssiin, kuten tietokantaan, sovelluspalvelimeen tai väliohjelmistokerrokseen. Sovellusliittymä mahdollistaa sovelluskehittäjien kutsua liittymien avulla edellä mainittujen resurssien tarjoamia palveluita. Esimerkiksi haettaessa informaatiota tietokannasta voidaan joutua käyttämään väliohjelmiston tarjoamaa sovellusliittymää, jotta saadaan luotua yhteys asiakastietotietokantaan. (Linthicum 2000, 39)

Sovellusliittymätason integroinnissa integroinnin kohteena on usein jokin paketoitu sovellus. Paketoidulla sovelluksella tarkoitetaan jotain kaupallista sovellusta, joka on mieluummin ostettu kuin itse rakennettu. Nämä sovellukset sisältävät uudelleenkäytettäviä liiketoimintaprosesseja, jotka edustavat parhaiksi todettuja liiketoimintamalleja. Tällaisten sovellusten hyödyt ovat selvät: miksi kehittää esimerkiksi uusi varastonhallintasovellus, jos lähes täydellisiä on markkinoilla tarjolla. Täysimittainen sovelluskehitys aina suunnittelusta käyttöönottoon on usein kalliimpaa kuin valmiin ostaminen. Paketoitujen sovellusten markkinajohtajana ovat eri toiminnanohjausjärjestelmät (ERP-järjestelmät), kuten SAP, Baan tai PeopleSoft. (Linthicum 2000, 13)

Monet paketoidut sovellukset ovat julkaisseet sovellusliittymiä, jotka mahdollistavat muiden sovelluksien pääsyn kapseloituihin palveluihin ja dataan. Sovellusliittymien tarjoamat palvelut voidaan jakaa Linthicumin (2000, 48) mukaan kolmeen eri kategoriaan: liiketoimintapalvelut, datapalvelut ja oliot.

Liiketoimintapalveluilla tarkoitetaan paketoidun sovelluksen sisältämiä liiketoimintaloogisia osia, jotka on tuotu esille sovellusliittymään. Esimerkiksi jos halutaan lisätä paketoidun sovelluksen asiakastietokantaan jonkin toisen sovelluksen uusi asiakas, kutsutaan asiakkaan päivitykseen tarkoitettua palvelua paketoidun sovelluksen tarjoamasta sovellusliittymästä. Paketoitu sovellus lisää asiakkaan tietokantaan aivan kuten palvelua olisi käytetty paketoidun sovelluksen käyttöliittymästä. Tällä tasolla toimittaessa tarvitaan perehtyneisyyttä paketoituun sovellukseen. Yhdessä sovelluksessa saattaa olla jopa 10 000 eri liiketoimintapalvelua, mutta silti EAI-arkkitehdin tulee tietää, mitä käytettävä palvelu tekee, mitä informaatiota palvelu tarvitsee ja mikä on palvelun odotettu tulos. (Linthicum 2000, 48-50)

Datapalveluilla tarkoitetaan sovellusliittymän tarjoamia palveluita, joilla käsitellään tietokantojen dataa. Nämä palvelut eivät kuitenkaan ole perinteisiä datatason integroinnin työkaluja vaan paketoidun sovelluksen toimittajan tarjoamia sovellusliittymiä datan käsittelyyn. Sovellusliittymän tarjoamat datapalvelut tarjoavat usein palveluita vain datan lukemiseen tietokannasta, koska datan päivittämiseen ja lisäämiseen liittyy riski rikkoa datan eheys. Tämän vuoksi enemmän logiikkaa tarvitsevat datan päivitykset ja lisäykset suoritetaan usein liiketoimintapalveluiden kautta. Dataa voi myös päivittää väliohjelmistojen tarjoamilla palveluilla suoraan ohi sovelluksen ”turvallisten” sovellusliittymien, mutta tämä ei ole suositeltavaa edellä mainitusta syystä. (Linthicum 2000, 50-52)

Sovellusliittymätason kolmas kategoria on oliot. Oliot käärivät sisäänsä liiketoiminta- ja datapalveluita. Aivan kuten olio-orientoituneessa ohjelmistokehityksessä nämä oliot kapseloivat dataa ja metodeita, joilla käsitellään dataa. Selkeä hyöty olioista on se, että olioita käyttämällä ei voida ohittaa paketoidun sovelluksen asettamia tiedon eheyden tarkistuksia, koska dataa ei voida käsitellä ilman olioiden metodeita. Olioissa on myös huonoja puolia. Nämä oliot eivät ole standardeja hajautettuja olioita, vaan ne ovat usein sovelluskohtaisia ja paketoidun sovelluksen toimittajan määrittelemiä, joten ne eivät sovellu jokaiseen kehitysympäristöön. Tämän vuoksi osa paketoitujen sovellusten toimittajista käyttää standardeja sovellusliittymiä oliopalveluille, kuten CORBA:n (Common Object Request Broker Architecture) tai Javan tarjoamia sovellusliittymiä. (Linthicum 2000, 52-53)

2.2.3 Metoditason integrointi

Metoditason EAI:ssa on tarkoituksena yhdistää kaksi tai useampi sovellus siten, että ne voivat käyttää toistensa liiketoimintalogiikka ja dataa metoditasolla. Metoditason EAI:ssa sovellukset on integroitu käyttäen jaettuja joukkoja yleisiä metodeja. Jaetut metodit voivat olla varastoituna keskitetysti jaetulle palvelimelle tai vaihtoehtoisesti voidaan käyttää hajautettujen olioiden teknologioita, jolloin sovellusten jaettuja metodeja käytetään verkon yli. Metoditason EAI liittyy läheisesti ohjelmistojen uudelleenkäyttöön. Kun sovelluksesta julkaistaan jaettuja metodeita, voivat eri sovellukset uudelleenkäyttää niitä. (Johannesson ym. 2000, 161)

Metoditason integroinnin selventämiseksi on parasta esittää hieman pelkistetty esimerkki. Oletetaan, että halutaan integroida kaksi sovellusta. Ensimmäinen on C++-pohjainen Linux-ympäristössä ajettava sovellus ja toinen on NT-pohjainen asiakas/palvelin-arkkitehtuuriin perustuva Java-sovellus, jonka taustalla on Sybasen tietokanta. Tarkoituksena on yhdistää sovellukset käyttäen metoditason

EAI teknologioita ja tekniikoita, jotta ne voivat jakaa liiketoimintalogiikkaa keskenään ja mahdollisesti muille sovelluksille tulevaisuuden tarpeita varten. Toisin kuin muilla EAI:n tasoilla, metoditasolla ei ole juuri muita vaihtoehtoja kuin rakentaa sovellukset uudestaan siten, että ne tukevat EAI:ta metoditasolla. Tähän on periaatteessa kaksi vaihtoehtoa: Ensimmäinen vaihtoehto on siirtää molempien sovelluksien liiketoimintalogiikka jaetulle sovelluspalvelimelle ja käyttää liiketoimintalogiikkaa keskitetysti. Toinen vaihtoehto on rakentaa molemmat sovellukset uudestaan siten, että ne tukevat mekanismeja, joilla voidaan jakaa metodeita. Toteutusvaihtoehtoina tälle menetelmälle on käyttää hajautettujen olioiden teknologioita, kuten CORBA tai DCOM (Distributed Common Object Model). Tämä tarkoittaa kuitenkin sitä, että molemmat sovellukset tulee kirjoittaa uudestaan. Onneksi tätä varten on olemassa työkaluja useille eri ympäristöille, mutta työkaluista huolimatta menetelmä on työläs. Hajautettuja olioita käsitellään myöhemmin väliohjelmistojen yhteydessä. (Lithicum 2000, 63-64)

Linthicum (2000) on esittänyt, että metoditason integroinnissa kannattaa käyttää hyödyksi sovelluskehyksiä (frameworks). Freedman (1993, 242) on esittänyt integrointiin ja uudelleenkäyttöön liittyen sopivan määritelmän sovelluskehykselle: ”Sovelluskehykset koostuvat abstrakteista luokista, niiden yhteistyöstä ja konkreeteista luokista. Samalla kun olio-orientoitunut ohjelmointi tukee ohjelmiston uudelleenkäyttöä, sovelluskehykset tukevat *suunnittelun* uudelleenkäyttöä.” Sovelluskehykset tarjoavat valmiiksi suunniteltuja ja testattuja sovellusarkkitehtuureja. Sovelluskehykset piilottavat sovelluskehityksen vaikeat yksityiskohdat ja antavat sovelluskehittäjien keskittyä sovelluksen toiminnallisuuksiin. Sovelluskehyksiä voidaan jakaa eri tyyppeihin. Oliosovelluskehykset ovat abstrakteja ja konkreetteja luokkia, joita kehittäjät voivat periyttää tai käyttää suoraan hyväksi. Monet oliokielet ja työkalut tarjoavat juuri näitä ominaisuuksia. Palvelusovelluskehykset ovat vastakohtana oliosovelluskehyksille; ne eivät tarjoa perinnän kautta toiminnallisuutta

sovelluksille vaan hyviä käytäntöjä ja näkökulmia. Hajautettujen olioiden sovelluskehys on yksi esimerkki palvelusovelluskehyksistä. Proseduraaliset sovelluskehykset tarjoavat hyviä lähestymistapoja metoditason EAI:lle. Proseduraaliset sovelluskehykset edustavat ”black box” -lähestymistapaa, koska niiden tarjoamia peruspalveluita (esimerkiksi tapahtumamonitorit) ei voi laajentaa tai muokata. Nopeimmin kasvava sovelluskehystyyppi on komponenttisovelluskehys, johtuen komponenttiteknologioiden (esimerkiksi JavaBeans) saamasta suuresta suosiosta.

2.2.4 Käyttöliittymätason integrointi

Käyttöliittymätason integrointi on alkeellisin EAI:n taso. Sovellukset yhdistetään toisiinsa käyttöliittymien integroinnilla. Vaikka tämä lähestymistapa on ehkä vähiten ”miellyttävä”, se on usein ainoa mahdollinen tapa toteuttaa integrointi. Etuna tällä tasolla on se, että se ei aiheuta muutoksia integroitaviin sovelluksiin. (Linthicum 2000, 79-81)

Käyttöliittymätason integrointia saatetaan käyttää seuraavista syistä: halutaan säästää integrointikustannuksista, käyttöliittymätasolla EAI-projekti sisältää vähemmän riskejä tai käyttöliittymätason integrointi on ainoa mahdollinen vaihtoehto. Integroitava sovellus on mahdollisesti tehty niin vanhalla teknologialla, että yrityksestä ei löydy ketään, jolla olisi riittävää tietotaitoa tehdä tarvittavia muutoksia sovellukseen. Sovelluksen dokumentaatio saattaa olla myös hyvin puutteellista tai sitä ei ole lainkaan.

Tiedon hakeminen käyttöliittymästä on prosessi, jossa on tärkeätä määritellä tarkoituksenmukaiset näytöt (screens), mistä tietoa haetaan, määritellä haettavan tiedon sijainti näytöllä, selvittää miten tieto luetaan näytöltä ja miten luettu tieto käsitellään. Käytännössä joudutaan luomaan automatisoitu ohjelma, joka simuloi todellista käyttäjää, navigoi käyttöliittymän näytöillä, emuloi näppäimistön

käyttöä ja lukee näytöt muistiin, jossa haluttu tieto jäsennetään, muokataan ja lähetetään väliohjelmistolle, joka lopulta lähettää tiedon vastaanottavalle järjestelmälle (Linthicum 2000, 81).

Käyttöliittymästä tiedon keräämiseen on kaksi eri lähestymistapaa: käytetään näyttöjä kuten raakaa dataa tai käytetään näyttöjä kuten olioita. Molemmissa tapauksissa tieto luetaan virtuaalisiin liittymiin käyttämällä käyttöliittymää ohjelmallisesti todellista käyttäjää jäljitellen. Datalähestymistavassa näytöt luetaan tekstitietovirraksi ja tästä tekstitietovirrasta data jäsennetään, tunnistetaan, konvertoidaan ohjelmalla (esimerkiksi sanomanvälittäjä ohjelmisto tai adapteri), joka on vastuussa käyttöliittymän informaation prosessoinnista. Oliolähestymistapa on hienostuneempi tapa käsitellä informaatiota käyttöliittymästä kuin datalähestymistapa. Näyttöjen käsittely olioina tarvitsee tiedon muuntamista sovelluksen olioiksi (esimerkiksi Java- tai CORBA-olioiksi). (Linthicum 2000, 85-87)

2.2.5 Sovellusten integraation tasot liiketoiminnan näkökulmasta

Integroinnin tarpeet lähtevät liiketoiminnan muutoksista tai kasvavista informaatioteknisistä vaatimuksista kilpailukyvyn säilyttämiseksi. Eräs merkittävä syy muutoksiin on yritysten siirtyminen funktionaalisesta liiketoimintojen jaottelusta prosessimaiseen ajattelutapaan. Perinteisesti yritykset on jaettu funktionaalisiin osastoihin, ja jokaisen osaston toimintojen tukena on itsenäisiä järjestelmiä. Jotta yritys säilyttäisi kilpailukykynsä, järjestelmien täytyy tukea liiketoimintaprosesseja yli funktionaalisten rajojen. Erityisesti elektroninen liiketoiminta asettaa tällaisia vaatimuksia yrityksen järjestelmille. Vaatimuksien täyttämiseksi funktionaalisesti orientoituneiden järjestelmien täytyisi kommunikoida keskenään, ja tämän toteuttamiseen tarvitaan sovellusten integrointia. (Wangler & Paheerathan 2000, 81-82)

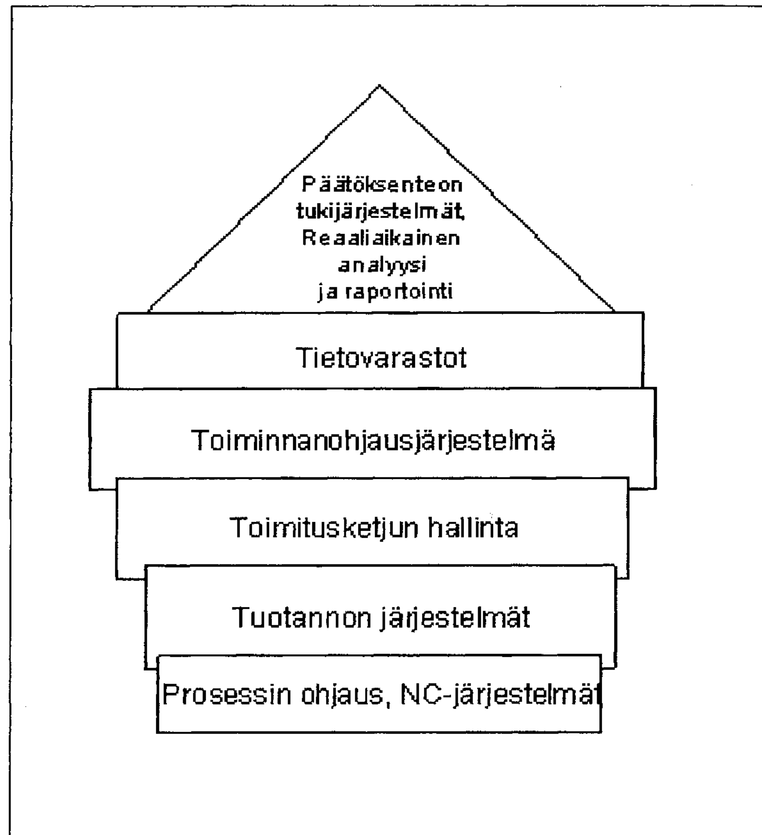
Liiketoiminnallisesta näkökulmasta katsottuna voidaan erottaa sovellusten integroinnin kolme eri tasoa: yrityksen sisäinen horisontaalinen integrointi, yrityksen sisäinen vertikaalinen integrointi ja yrityksien välinen sovellusten integrointi. Horisontaalisella integroinnilla tarkoitetaan yrityksen eri funktionaalisia alueilla toimivien sovellusten integrointia. Vertikaalisella integraatiolla tarkoitetaan yrityksen järjestelmien integrointia eri hallinnollisten ja liikkeenjohdollisten tasojen välillä. (Wangler ym. 2000, 81-83)

Tyypillinen esimerkki horisontaalisesta integroinnista on toimitusketjun hallinta (Supply Chain Management), jossa yrityksen tavoitteena on optimoida koko toimitusketjun toiminnot aina ostotapahtumasta tuotantoon ja toimitukseen. Tavoitteena on siis minimoida toimitusketjuun käytettävä aika ja kustannukset sekä maksimoida asiakkaan saama hyöty. (Wangler ym. 2000, 83.) Toimitusketjun optimoinnin mahdollistamiseksi toimitusketjua täytyy ajatella kokonaisena prosessina, joka rikkoo eri funktionaalisten alueiden rajoja sekä liiketoiminnan että järjestelmien näkökulmasta.

Vertikaalisen integraation tavoitteena on integroida järjestelmiä yrityksen eri hallinnollisilla tasoilla. Esimerkiksi tyypillinen toiminto useissa yrityksissä on tuotanto. Tehdasteollisuudessa tätä toimintoa saattaa kontrolloida alimmalla tasolla prosessinhallintajärjestelmä ja tietokoneistetut numeeriset ohjausjärjestelmät (NC machinery). Nämä järjestelmät käyttävät sovelluskohtaisia datan esitysmuotoja ja viestejä. Hyvin usein nämä järjestelmät toimivat eri käyttöjärjestelmillä ja käyttävät eri verkkoteknologioita. Nämä ohjausjärjestelmät tarvitsevat syötteenä dataa ylemmän tason suunnittelu- ja aikataulutusjärjestelmiltä samalla, kun ylemmän tason järjestelmät keräävät dataa alemman tason järjestelmiltä. (Wangler ym. 2000, 83-83)

Tiedon kulku on siis kaksisuuntaista. Esimerkiksi samalla kun nämä alemman tason järjestelmät saavat syötteenä tietovirtaa ylemmältä tasolta

toiminnanohjausjärjestelmältä, alemman tason järjestelmät keräävät informaatiota tuloksista ja lähettävät niitä päätöksenteon tukijärjestelmille ylemmälle hallinnolliselle tasolle (KUVIO 2). Vaikka osa järjestelmistä korvattaisiin yhdellä toiminnanohjausjärjestelmällä (ERP), tarvitaan silti olemassa olevia järjestelmiä ja niiden yhteistoimintaa (integrointia). Markkinoilla ei ole sellaista ERP-järjestelmää, joka korvaisi nämä järjestelmät, eikä sellaisen hankkiminen olisi taloudellisesti ja liiketoiminnallisesti kannattavaa. Liiketoiminta on koko ajan vaarassa muuttua, ja silloin aikaa vievä järjestelmien vaihdos on monissa tapauksissa kannattamatonta. (Wangler ym. 2000, 83-83)



KUVIO 2. Vertikaalisen integroinnin sovelluskerroksia (Wangler ym. 2000, 86)

Wanglerin ym. (2000, 86-87) mukaan liiketoiminnan kannalta sovellusten integroinnin kolmas taso on yritysten välinen integrointi. Yritysten välisessä

sovellusintegraatiossa käytetään samoja teknologioita ja tekniikoita kuin EAI:ssa, mutta EAI:n määritelmään kuuluu vain yrityksen sisäisten sovellusten integrointi, joten tässä ei käsitellä sitä enempää.

2.3 Yhteenveto integroinnista

EAI:lla tarkoitetaan yhden yrityksen ja sen sisäisten sovellusten integrointia. Syitä yrityksen sovellusten integroinnille ovat useimmiten organisaationaliset muutokset yrityksen sisällä ja liiketoiminnan tehostaminen, jonka tueksi tarvitaan uusien sovelluksien ja olemassa olevien sovelluksien liittämistä tarvittavalla tasolla. Keskeisin EAI:n ongelma on olemassa olevat sovellukset, jotka on suunniteltu vastaamaan vain jotain tiettyä toimintoa ilman tietoa siitä, että niiden tulisi pystyä kommunikoimaan joidenkin toisten sovelluksien kanssa.

Wanglerin ym. (2000) esittämät kuusi integroinnin tasoa ovat osittain samat kuin Linthicumin (2000), mutta edellisillä on ollut lähtökohtana markkinoilla olevat tuotteet. Heidän määrittelyssään on myös mukana liiketoimintaprosessien integrointi, kun Linthicumin (2000) tasot on määritelty teknisestä näkökulmasta. Wangler ym. (2000) ovat käsitelleet myös yrityksien välistä sovellusintegrointia, jonka voidaan katsoa kuuluvan EAI:n käsitteen ulkopuolelle. EAI:lla siis tarkoitetaan yrityksen sisällä tapahtuvaa sovellusten integrointia.

Sovellusten integrointi datatasolla on yleensä yksinkertaisin ja edullisin toteuttaa, eikä se aiheuta muutoksia integroitaviin sovelluksiin, mutta datatason tarjoamat mahdollisuudet eivät usein riitä. Sovellusliittymätason integrointi on riippuvainen paketoidun sovelluksen toimittamista sovellusliittymistä, ja aina niitä ei ole tarjolla ollenkaan. Metoditason integrointi tarvitsee käytännössä paljon muutoksia olemassa oleviin sovelluksiin ja on työläs verrattuna muihin tasoihin. Metoditason ratkaisut ovat kuitenkin parhaiten skaalautuvia ja hyvin tehtyinä tarjoavat tukea myös tulevaisuuden tarpeisiin. Käyttöliittymätason

integrointi on yleensä viimeisin vaihtoehto, jos muut menetelmät eivät sovellu integroinnin ongelman ratkaisuksi. Käyttöliittymätasolla toteutetut integroinnit ovat hankalia ylläpitää ja muuttaa, mutta eduksi voi lukea sen, että integroitaviin sovelluksiin ei tarvitse tehdä muutoksia, koska sovelluksia käytetään käyttöliittymän kautta aivan kuten todellisetkin käyttäjät sen tekisivät.

3. EAI JA VÄLIOHJELMISTOT

Tässä luvussa perehdytään yleisimpiin integraatioarkkitehtuureihin ja tekniikoihin. Kappaleessa 3.1 esitellään tunnetuimmat arkkitehtuurimallit ja kappaleessa 3.2 perehdytään erityyppisiin väliohjelmistotekniikoihin.

Väliohjelmistot (middleware) ovat yrityksen sovellusten integroinnissa hyvin keskeisessä osassa. Lähes kaikki EAI-ratkaisut perustuvat johonkin väliohjelmistotekniikkaan riippumatta siitä, millä tasolla EAI toteutetaan. Ruhia ym. (2001, 52) mukaillen väliohjelmiston voi määritellä seuraavasti: Väliohjelmisto on ohjelmistotyyppi, joka edistää kahden tai useamman ohjelmiston välistä kommunikointia käyttäen määriteltyjä sovellusliittymiä tai sanomia. Lisäksi väliohjelmistoon kuuluu ajonaikainen ympäristö, jolla voi hallita palvelupyyntöjä ohjelmistojen välillä.

Yksinkertaisimmillaan väliohjelmisto voi olla pelkkä kommunikointiputki kahden sovelluksen välillä (esimerkiksi Java:n RMI, Remote Method Invocation) tai palvelu, jolla saadaan yhteys tietokantaan (JDBC tai ODBC).

Yrityksen sovellusten integrointi väliohjelmistolla voidaan suorittaa kahden eri periaatteen mukaisesti: *yhteenkytkemisen* (coupling) tai *yhteenkuuluvuuden* (cohesion) periaatteen mukaisesti. Tässä yhteydessä yhteenkytkemisellä tarkoitetaan joko datan ja logiikan yhdistämistä, logiikan ja logiikan yhdistämistä ja/tai datan ja datan yhdistämistä. Yhteenkytkemisen periaatteessa sovellukset ja tietokannat ovat tiukasti sidoksissa toisiinsa. Jos yhteenkin sovellukseen tai tietokantaan tehdään muutoksia, täytyy muuttaa myös kaikkia siihen sidoksissa olevia sovelluksia tai tietokantoja. Vaikka yhteenkytkemisessä on selviä haittoja, sillä on myös etunsa. Yhteenkytkemisellä sovellukset saadaan yhdistetyksi kiinteästi toisiinsa ja sovelluslogiikkaa voidaan paremmin jakaa ja käyttää

uudelleen. Yhteenkuuluvuuden periaatteella saadaan enemmän joustavuutta integrointiin. Tämän periaatteen mukaan sovellukset ja tietokannat ovat itsenäisiä, eivätkä muutokset yhteen sovellukseen aiheuta muutoksia toisiin. Se kumpi näistä menetelmistä on parempi, riippuu hyvin paljon integroinnin tarpeista ja integroitavista sovelluksista. Kummallakin periaatteella on omat etunsa. (Linthicum 2000, 25)

3.1. Väliohjelmistoarkkitehtuurit

Väliohjelmistot perustuvat kahteen eri korkean tason arkkitehtuurimäärittelyyn: pisteestä pisteeseen -arkkitehtuuri (point-to-point) tai monesta moneen -arkkitehtuuri (many-to-many). Seuraavaksi käsitellään perusteet kummastakin arkkitehtuurista.

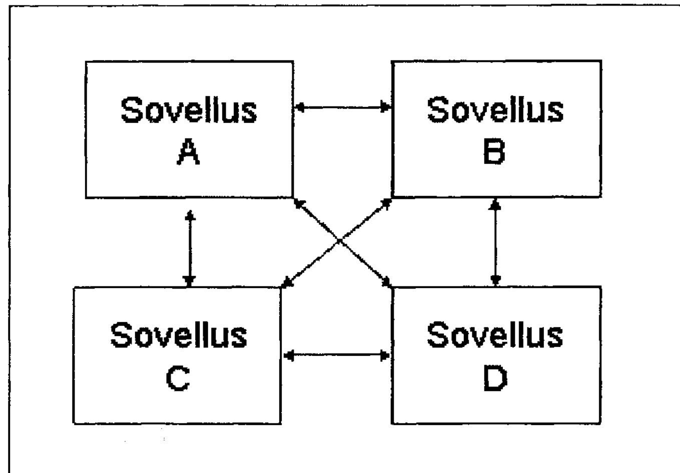
3.1.1 Pisteestä pisteeseen -arkkitehtuuri

Pisteestä pisteeseen -arkkitehtuuri (point-to-point) on sovellusten integrointiarkkitehtuuri, jossa jokainen sovellus yhdistetään suoraan toiseen sovelluksen (KUVIO 3). Tämä integrointiratkaisu saattaa toimia, jos integroitavia sovelluksia on vain muutama, mutta jos sovelluksien lukumäärä kasvaa suureksi, yhteyksien lukumäärä kasvaa mahdottomaksi. (Johannesson & Perjons 2000)

Tässä arkkitehtuurissa joudutaan rakentamaan erikseen jokainen yhteys kahden eri sovelluksen välille (nuolet kuviossa 3), jotta saadaan sovellukset kommunikoimaan keskenään. Yhteydet voivat olla yhden- tai kahdensuuntaisia riippuen tarpeesta. (Linthicum 2000, 133-134)

Pisteestä pisteeseen -arkkitehtuuri on korkean tason arkkitehtuuri, ja siinä kuvataan vain sovelluksien välisiä yhteyksiä. Arkkitehtuuri ei ota kantaa, miten

sovellusten välinen datan vaihto suoritetaan. Sovellus voi esimerkiksi saada suoran yhteyden toisen sovelluksen tietokantaan väliohjelmiston kautta tai sovellus voi kutsua jonkin tietyn liittymän kautta toisen sovelluksen palveluita.



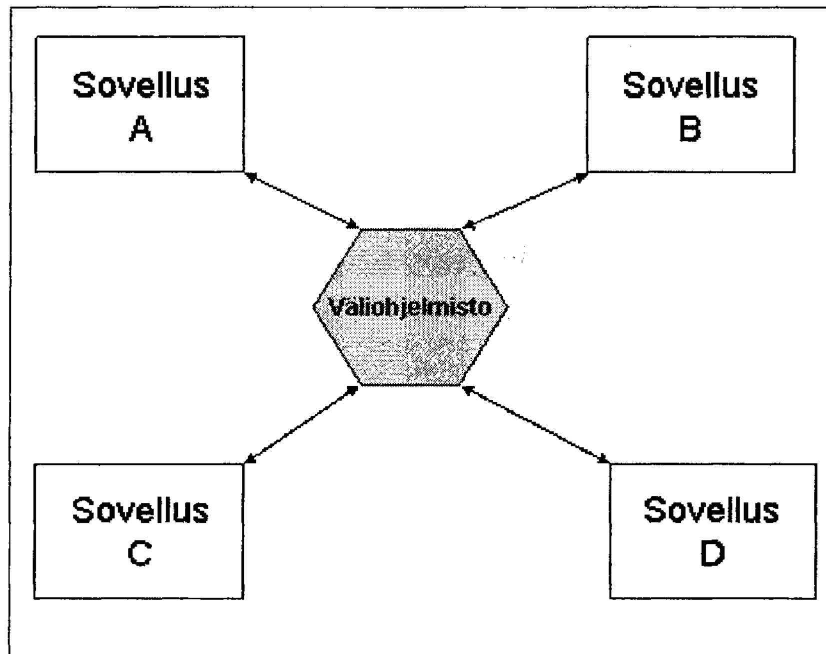
KUVIO 3. Pisteestä pisteeseen -malli (Johannesson & Perjons 2000)

Pisteestä pisteeseen -arkkitehtuurin etuna on sen yksinkertaisuus. Se vapauttaa EAI-arkkitehdit ja -kehittäjät monimutkaiselta suunnittelu- ja kehitystyöltä. Yrityksen sovellusten integroinnissa tämä arkkitehtuuri ei ole aina käyttökelpoisin, koska EAI-ympäristössä on usein tarve integroida yhteen useita eri sovelluksia ja yrityksen liiketoimintaprosesseja. Pisteestä pisteeseen -arkkitehtuuria käytetään kuitenkin edelleen monien integroinnin kannalta tärkeiden väliohjelmistotuotteiden pohjana. Esimerkiksi etäkutsut ja eräät sanomapohjaiset väliohjelmistotuotteet pohjautuvat pisteestä pisteeseen -arkkitehtuuriin. (Linthicum 2000, 133-134)

3.1.2 Monesta moneen -arkkitehtuuri

Kuten nimikin kertoo monesta moneen -arkkitehtuuri yhdistää monta sovellusta moneen sovellukseen. Monesta moneen -arkkitehtuuri on EAI:n kannalta tarkoituksenmukaisempi kuin pisteestä pisteeseen -arkkitehtuuri, koska se on

mukautuvampi ja joustavampi sekä vähentää pisteestä pisteeseen -arkkitehtuurin kompleksisuutta. Periaatteena tässä ratkaisussa on käyttää monesta moneen arkkitehtuuria tukevaa väliohjelmistoa integroitavien sovellusten välillä. Integroitavaan sovellukseen tarvitsee rakentaa vain yksi yhteys: yhteys väliohjelmiston ja sovelluksen välille (KUVIO 4). (Johannesson & Perjons 2000, 214)



KUVIO 4. Monesta moneen -malli (Johannesson & Perjons 2000, 214)

Monesta moneen -mallin mukaiseen EAI-ratkaisuun on helpompi tehdä muutoksia ja lisätä uusia integroitavia sovelluksia. Monesta moneen -arkkitehtuurin mukaisia ratkaisuja on useita, kuten sanomanvälittäjät, tapahtumamonitorit ja hajautetut oliot. (Linthicum 2000, 134-135; Johannesson & Perjons 1999)

3.1.3 Sovittimet sovellusten integraatiossa

Koska tämän tutkielman konstruktiivinen osuus keskittyy sovittimen rakentamiseen, on oleellista määritellä myös sovittimen käsite.

Iyerin mukaan (1999) sovitin (adapter) liittää valmissovellukset, tietokannat, räätälöidyt sovellukset ja perinteiset sovellukset integroituun ympäristöön. Niitä voidaan kutsua myös yhdistäjiksi tai yhteyskirjastoiksi. Adapterit tarjoavat myös tavan ymmärtää integroitavien sovellusten tapahtumia ja tietomalleja. Tämä mahdollistaa erilaisten tietomallien yhteensovittamisen.

Sovittimia tarvitaan yrityksen sovellusten integroinnissa, riippumatta käytetyistä arkkitehtuureista tai menetelmistä. Edellä mainitussa pisteestä pisteeseen -arkkitehtuurin mukaisessa toteutuksessa tarvitaan sovitin jokaisen sovelluksen välille. Sovitin mahdollistaa sovelluksien välisten yhteyksien muodostamisen. Samoin monesta moneen -arkkitehtuurin mukaisessa ratkaisussa tarvitaan joko sovelluskohtainen sovitin tai liittymä integroitavan sovelluksen ja väliohjelmiston välille.

Beverigde (2000) on määritellyt perinnejärjestelmiin kehitettäville sovittimille muutamia perusvaatimuksia. Sovittimen tulee olla suorituskykyinen ja tukea monisäikeistä suoritusta. Sovittimen toimintojen tulee olla oikeanlaisia, eikä niiden välillä saa olla ristiriitoja. Sovittimen tulee olla helposti mukautettavissa integroivan järjestelmän muutoksiin. Sovittimen tulee olla laajennettava. Sovittimen pitää selvitä hallitusti poikkeustilanteista ja tukea tapahtumanhallintaa.

3.2 Väliohjelmistotekniikoita

Linthicum (2000, 120-122) on jaotellut väliohjelmistoja eri tyyppeihin. Näitä ovat tapahtumapohjaiset väliohjelmistot (tapahtumamonitorit, sovelluspalvelimet), sanomanvälittäjät, sanomapohjaiset väliohjelmistot, etäkutsut (Remote Procedure Call), hajautetut oliot ja tietokantapohjaiset väliohjelmistot. Väliohjelmistoja on siis hyvin monenlaisia ja moneen eri tarkoitukseen. Seuraavissa kappaleissa käydään läpi perusperiaatteet EAI:n kannalta tärkeimmistä väliohjelmistotyypeistä.

Väliohjelmistot käyttävät kahdentyyppisiä kommunikointimekanismeja: asynkronisia ja synkronisia. Kommunikointi on *synkronista*, kun esimerkiksi sanoma tai etäkutsu lähetetään vain siinä tilanteessa, kun vastapuoli on valmiina ottamaan sen vastaan ja käsittelemään sen. Synkroninen väliohjelmisto on tiukasti sidottu (yhteenkytkemisen periaate) sovellukseen. Lähettävän sovelluksen toiminta keskeytyy, kunnes vastaanottava sovellus vastaa sanomaan tai on suorittanut etäkutsun. Tällaista väliohjelmistoa kutsutaan estäväksi väliohjelmistoksi (blocking middleware). *Asynkroninen* kommunikointi on käytännössä mahdollista vain sanomapohjaisilla ratkaisuilla. Sanomien jakelu on asynkronista silloin, kun sanoma voidaan lähettää riippumatta siitä, onko vastapuoli valmis käsittelemään sen vai ei. Asynkronisessa kommunikoinnissa molemmat sovellukset voivat suorittaa oman tehtävänsä riippumatta toisen sovelluksen valmiudesta. Asynkronista väliohjelmistoa ei ole tiukasti sidottu mihinkään sovellukseen, eikä sanoman lähetys keskeytä lähettävän sovelluksen toimintaa. (Linthicum 2000, 135-137; Luoma, Muhonen & Huomo 1999, 161)

3.2.1 Etäkutsut

Etäkutsut (Remote Procedure Call eli RPC) ovat tavanomaisen aliohjelmakutsun verkkolaajennus. RPC-väliohjelmisto toteuttaa synkronisen

kommunikointimallin, jossa kutsuva ohjelma odottaa aliohjelmakutsun palaamista. Etäkutsussa aliohjelma voi sijaita toisella laitteella verkkoyhteyden takana, jolloin etäkutsumekanismin on ajonaikaisesti huolehdittava monista asioista, jotka ohjelmoijan itsensä toteuttamana olisivat hyvin vaikeita, työläitä ja yleensä myös laiteriippuvaisia. Tyypillisesti ajonaikainen RPC-mekanismi sisältää muun muassa seuraavia rutiineja: eri laitteilla toimivien ohjelmistojen osien suorituksien synkronisointi, suurten parametrijoukkojen välittämisen puskurointi, tarvittavien tiedon esitystapamuunnosten tekeminen, tietoturva, virhetilanteista toipuminen, nimipalvelut. Etäkutsut eivät ole mikään uusi asia. Esimerkiksi IBM käytti niitä jo 70-luvulla sisäisesti omissa järjestelmissään. Etäkutsut ovat tulleet kuitenkin uudelleen esille, kun hajautettuihin järjestelmiin on haettu tehokasta ja tietoliikenteen peittävää standardoitua mallia. (Luoma ym. 1999, 155-156)

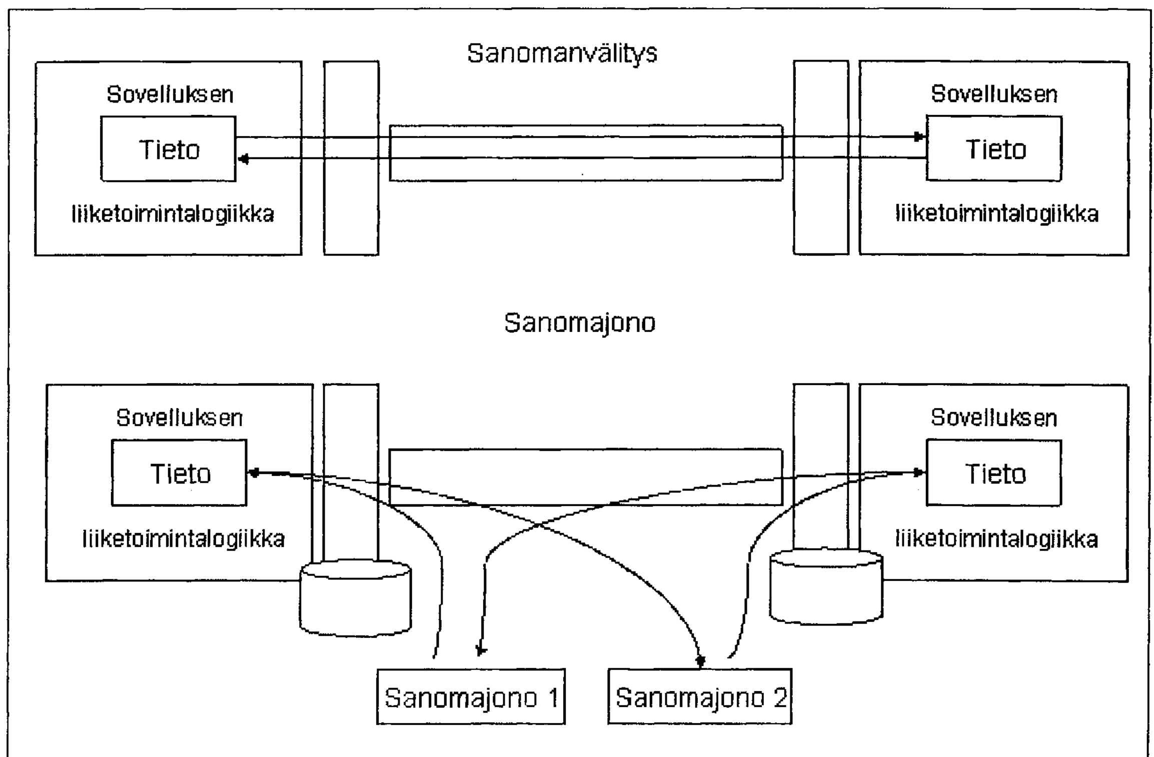
3.2.2 Sanomapohjainen väliohjelmisto

Sanomaratkaisussa sovellusten välillä siirretään viestejä. Ohjelman suorituksen kontrolli ei siirry samalla tavalla vastaanottajalle kuin etäkutsuissa eli sanomapohjainen kommunikointi on asynkronista. Hajautetussa ympäristössä prosessien välinen kommunikointi voidaan toteuttaa käyttäen *sanomanvälitystä* tai *sanomajonoja.* Molemmat kommunikoivat prosessit ovat näissä malleissa varsin itsenäisessä roolissa (päinvastoin kuin etäkutsuissa). Yhdessä näitä tekniikoita voidaan kutsua sanomapohjaiseksi väliohjelmistoksi (Message Oriented Middleware eli MOM). Tällä tarkoitetaan liiketoimintasovellusten ja tietoliikenneohjelmistojen välissä toimivaa ohjelmistoa, joka mahdollistaa mittavienkin hajautettujen sovellusten tehokkaan kommunikoinnin myös heterogeenisissä ympäristöissä. (Luoma ym. 1999, 159-160)

Luoma ym. (1999, 159-163) määrittelevät sanoman (message) seuraavasti: ”sanoma on vaihtelevan pituinen tavusekvenssi, joka yleensä muodostaa

merkityksellisen tietokokonaisuuden sekä lähettävälle että vastaanottavalle sovellukselle". Nykyiset sanomapohjaiset väliohjelmistot sallivat sanoman olla lähes minkä kokoista ja muotoista dataa tahansa. Sanomat voivat sisältää esimerkiksi merkkijonoja, binäärimuotoista dataa tai olioita. Sanomanvälittäjät osaavat pilkkoa tilaa vievän sanoman tarvittaessa pienempiin osiin.

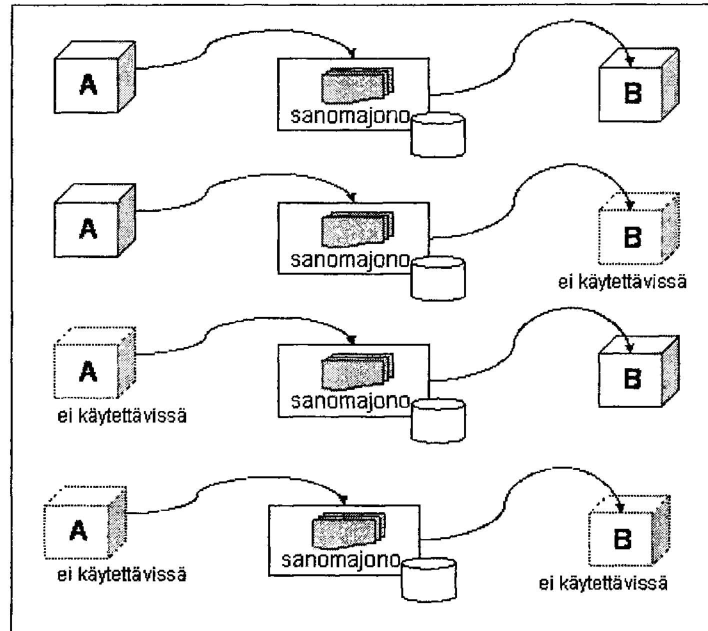
Sanomanvälityksessä on yksinkertaisesti kyse sanomien välittämisestä prosessilta toiselle. Sanomajonoissa sanomanvälitykseen on lisätty sanomien jonotusmalli (KUVIO 5). Siinä jonot voidaan nimetä ja prosessit voidaan kohdistaa palvelemaan tiettyä jonoa. Jonotusmalli toimii siten, että osapuolet avaavat yhteisen jonon, johon toinen kirjoittaa ja josta toinen lukee saapuneet viestit. Sanomajonomalli on parempi kuin sanomanvälitysmalli, koska vain yksi sovellus on aktiivinen kerrallaan. Tästä syystä ollaan siirtymässä sanomanvälityksestä sanomajonoihin. (Luoma ym. 1999, 159-162)



KUVIO 5. Sanomanvälityksen looginen toimintamalli (Luoma ym. 1999, 161)

Jono (queue) voidaan ajatella listana, johon sanomat on varastoitu odottamaan toimitusta vastaanottavalle sovellukselle. Jonot voivat sijaita keskusmuistissa, tallennettuna levylle tai molemmissa. Jokainen jono kuuluu jollekin tietylle jono-ohjaimelle (queue manager), joka huolehtii jonon ylläpidosta. Jono-ohjaimen tehtävänä on huolehtia sovellukselta tulevan sanoman laittamisesta eteenpäin oikeaan jonoon ja siitä eteenpäin vastaanottavalle sovellukselle. (Linthicum 2000, 166-168)

Sanomajonokommunikointi on asynkronista, jos vastaanottava ohjelma on tavoittamattomissa (esimerkiksi verkkoyhteys on poikki tai sovellus on alhaalla). Sanomat säilyvät jonossa, ja ne välitetään perille myöhemmin. Tällainen aikariippumaton sanomajonomalli mahdollistaa sovellusten välisen kommunikoinnin, vaikka sovellukset eivät olisi yhtä aikaa tavoitettavissa (KUVIO 6). Esimerkiksi jos sanomia vastaanottava sovellus (B) on kiireinen tai alhaalla, sanomia lähettävän sovelluksen (A) ei tarvitse jäädä odottamaan vastaanottavan sovelluksen suoritusta, vaan sanoma jää jonoon odottamaan. Myös sanomia lähettävä sovellus (A) voi olla tilapäisesti alhaalla, ja samanaikaisesti sanomajono voi välittää aikaisemmin lähetettyjä sanomia vastaanottajalle (B). (Monson-Haefel & Chappell 2001, 99-104)



KUVIO 6. Asynkronisen kommunikoinnin mahdollisia eri tiloja.

Useimmat sanomajonot tarjoavat kahta eri sanomavälitysmekanismia: pisteestä pisteeseen (point to point) eli kahden eri sovelluksen välinen viestintä ja julkaisuviestintä (publish and subscribe). Julkaisuviestinnässä voidaan lähettää sanomia useille vastaanottajille. Vastaanottajat voivat tilata (subscribe) tietyn otsikon (topic) viestejä. Tässä *otsikolla* tarkoitetaan sanomajonoa vastaavaa sanomanvälitysmenetelmää. Jokainen otsikon tilaaja saa kopioin kaikista otsikolle lähetetyistä viesteistä. Sanomat lähetetään tilaajille automaattisesti ilman, että vastaanottajien tarvitsisi erikseen hakea tai kysellä viestejä. Pisteestä pisteeseen malli mahdollistaa sovellusten lähettää ja vastaanottaa sanomia synkronisesti tai asynkronisesti sanomajonon kautta. Pisteestä pisteeseen menetelmä on perinteisesti toteutettu hakuun perustuvalla (polling) menetelmällä, mutta esimerkiksi JMS (Java Message Service) mahdollistaa sanomien automaattisen toimituksen vastaanottajalle, kuten julkaisuviestinnässä. (Barish 2001, 27-30)

Vaikka sanomajono tarjoaa EAI:hin paremman ratkaisun kuin esimerkiksi etäkutsut, se ei ole kuitenkaan täydellinen ratkaisu EAI:n ongelmiin.

Sanomanvälittäjät (message broker) tuovat lisäarvoa perinteisiin MOM-tuotteisiin. Tunnettuja MOM-tuotteita ovat muun muassa MSMQ (Microsoft Message Queue) ja IBM:n MQSeries. (Linthicum 2000, 166-168)

3.2.3 Sanomanvälittäjät

Sanomanvälittäjien (message broker) perusajatuksena on liittää erilaisia järjestelmiä helposti yhteen riippumatta siitä, ovatko ne valmisohjelmistoja vai itse rakennettuja, toimivatko ne keskuslaitteilla (mainframe) tai hajautetuissa ympäristössä ja riippumatta käytetyistä tekniikoista. Teknisesti sanomanvälitysratkaisussa on olennaista eri käyttöjärjestelmäalustojen ja ennen kaikkea eri tekniikoilla rakennettujen sovellusten liitettävyys. Yhteys on ratkaisussa pystyttävä muodostamaan monenlaisiin väliohjelmistotekniikoihin ja sovellusliittymiin. (Luoma ym. 1999, 164)

Sanomanvälittäjät laajentavat sanomakommunikointia ja sanomajonoja. Ratkaisu sisältää sanomaliikenteen muodon ja kulun hallintaan keskitetyt palvelut. Sanomanvälittäjät tarjoavat samoja palveluja kuin sanomajonot, mutta ne tarjoavat mahdollisuuden myös sisällön muokkaukseen, sääntöjen luomiseen ja datan muuntamiseen toiseen muotoon. Sanomanvälittäjät koostuvat kolmesta eri osasta: tietomuunnoskerros (message translation layer), päättelykone (rules engine) ja älykäs reititys (intelligent routing). Sovelluksen liittämiseksi sanomanvälittäjään ja sitä kautta muihin sovelluksiin, liitettävään sovellukseen tarvitaan sovitin (adapter). Sovittimen avulla sanomanvälittäjä ja sovellus saadaan toimimaan yhteen. Siirrettäessä data kulkee siis sovittimen kautta sanomanvälittäjälle, jossa tietomuunnoskerros tunnistaa dataformaatin ja muuttaa sen vastaanottavan sovelluksen haluamaan muotoon. Älykäs reititin tunnistaa viestin alkuperän ja lähettää viestin oikealle kohdesovellukselle. Päättelykoneen avulla voidaan ohjata sanomien reititystä ja muunnosta. (Linthicum 2000, 292; Luoma ym. 1999, 164-165)

3.2.4 Tapahtumapohjaiset väliohjelmistot

Hajautetussa ympäristössä tietokannan tiedot pyritään pitämään eheänä vastaavasti kuten keskitetyissä ympäristöissä, mutta menettelytavat ovat huomattavasti monimutkaisempia. Päivitystapahtumien kohdistuessa ainoastaan yhteen ympäristöön käytetään tietokantatoimittajan tarjoamaa tapahtumamekanismia tai tapahtumamonitoria. Hajautetussa ympäristössä saattaa olla tarvetta päivittää eri tietokantoja yhtenä kokonaisuutena, jolloin tapahtuman eheyden aikaansaamiseksi tarvitaan kehittyneitä menettelyjä eli niin sanottua kaksivaiheista sitoutumismekanismia (two-phase commit). Tyypillisiä käyttötarpeita ovat työaseman keskitetyn tietokannan päivittäminen samassa tapahtumassa tai paikalliseen ja keskitettyyn tietokantapalvelimeen kohdistuva, tapahtumakokonaisuuden muodostava päivitys. Kaksivaiheinen sitoutuminen on käytettävissä hajautetun ympäristön tapahtumamonitoreissa ja useimmissa relaatiotietokanympäristöissä. (Luoma ym. 1999, 180-181)

Tapahtuman käsitteen ymmärtäminen on olennaista tarkasteltaessa tapahtumanhallintaa ja tapahtumankäsittelyratkaisuja. Tapahtumankäsittelyjärjestelmissä käsittelyn perusyksikkönä on tapahtuma (transaktio). Tapahtumalla on neljä perus- tai eheysominaisuutta, joiden toteutuminen järjestelmässä on taattava. Näitä ominaisuuksia kutsutaan myös ACID-ominaisuuksiksi. ACID-ominaisuuksia ovat (Luoma ym. 1999, 181):

1. Jakamattomuus/atomisuus (atomicity). Tapahtuma on jakamaton tehtäväkokonaisuus. Kaikki sen osatehtävät suoritetaan tai jätetään suorittamatta, osittainen suorittaminen ei ole mahdollista.
2. Eheys (consistency). Tapahtuman tulee suorituksen jälkeen jättää koko järjestelmä eheään lopputilaan tai muussa tapauksessa peruuttaa koko tapahtuma, jolloin järjestelmä palautetaan tietojen osalta alkuperäiseen tilaansa.

3. Eristyvyys (isolation). Tapahtuman kanssa samanaikaisesti suoritettavat muut tapahtumat eivät vaikuta sen loogiseen käyttäytymiseen. Tapahtuman jaettuihin resursseihin tekemät muutokset eivät saa näkyä ulkopuolelle ennen kuin tapahtuma kokonaisuutena vahvistetaan (commit).
4. Pysyvyys (durability). Tapahtuman vahvistamisen jälkeen sen aikaansaamat muutokset ovat pysyviä. Muutosten tulee säilyä järjestelmän vikaantuessakin.

Esimerkkinä tapahtumanhallinnan käytöstä on yksinkertainen pankkisiirtotapahtuma. Kun toiselta tililtä otetaan rahaa, sama summa lisätään toiselle tilille. Otto ja pano muodostavat kiinteän kokonaisuuden eli tapahtuman. Tapahtuma ei saa keskeytyä esimerkiksi oton jälkeen tai koko tapahtuma tulee peruuttaa alusta alkaen. (Luoma ym. 1999, 180-181)

Tapahtumamonitorit (Transaction Processing Monitors) ovat perinteisiä tapahtumapohjaisia väliohjelmistoratkaisuja. Itse asiassa niitä voidaan pitää ensimmäisen sukupolven sovelluspalvelimina. Tapahtumamonitorit tarjoavat sijainnin liiketoimintalogiikalle ja pitävät huolen kahden tai useamman sovelluksen yhteyksistä. Tapahtumamonitorien toiminta perustuu tapahtumiin, joihin liiketoimintalogiikka on pakattu. Nämä tapahtumat joko suoritetaan tai ei (ACID-periaate). Etuna arkkitehtuurissa on juuri toimintojen jakaminen pieniin, helposti hallittaviin ja prosessoitaviin osiin. Useimmat tapahtumamonitorit voivat kommunikoida sekä asynkronisesti että synkronisesti. Markkinoiden tunnetuimpia tapahtumamonitoreja sisältäviä ohjelmistoja ovat: Bea Systems:in Tuxedo, Microsoftin MTS ja IBM:n CICS. Linthicumin (2000, 128-130)

3.2.5 Hajautetut oliot

Voidaan ajatella, että myös hajautettuihin olioihin (distributed objects) perustuvat teknologiat ovat väliohjelmistoja, koska ne mahdollistavat sovellusten välisten kommunikaation. Hajautettujen olioiden avulla on mahdollista toteuttaa koko yrityksen kattava metodien jakaminen. Hajautetut oliot soveltuvatkin parhaiten metoditason EAI:hin, samoin kuin tapahtumapohjaiset väliohjelmistoratkaisut. Hajautetut oliot ovat pieniä sovellusohjelmia, jotka käyttävät standardoituja liittymiä ja protokollia sovellusten väliseen kommunikointiin. (Linthicum 2000, 125-126)

Eräs EAI:n tavoitteista on integroida sovellukset siten, että sovelluksiin tarvitsee tehdä mahdollisimman vähän muutoksia. Tässä menetelmässä se ei ole kuitenkaan mahdollista, koska yrityksen jokainen integrointiin osallistuva sovellus on muutettava tukemaan hajautettujen olioiden teknologiaa. Hajautettujen olioiden käyttö vaatii siis valtavan määrän työtä, mutta se saattaa kuitenkin kannattaa.

Usein oliotekniikka tuodaan esiin käyttöliittymien ja erilaisten sovelluskehitysvälineiden yhteydessä, jolloin sen avulla toivotaan saatavan helppokäyttöisyyttä, joustavuutta ja uudelleenkäytettävyyttä. Perusajatukseltaan oliotekniikka soveltuu hyvin myös järjestelmien hajauttamiseen (Luoma 1999, 168-169):

- *Modulaarisuus (modularity).* Oliot ovat itsenäisiä komponentteja, jotka tarjoavat ulospäin tarkasti määritellyn palveluliittymän, jonka kautta voidaan käsitellä olion palveluita ja dataa.
- *Liitettävyys (connectivity).* Oliot voivat kommunikoida dynaamisesti tai ennalta kiinnitettyjen kutsumekanismien avulla erityyppisten verkkojen yli. Tämän toteuttamiseen tarvitaan oliovälittäjiä (object broker).

- *Siirrettävyys (portability).* Oliot voidaan siirtää vapaasti käyttöjärjestelmästä toiseen. Käytännössä olioiden siirrettävyys käyttöjärjestelmien välillä ei ole ollut kovin hyvä. Tämän vuoksi tarvitaan kieliriippumattomia oliomalleja (komponenttiajattelu), jolloin palveluja pyytävät ja tarjoavat oliot voivat olla eri ohjelmointikielillä toteutettuja.
- *Yhteensopivuus (interoperability).* Olemassa oleviin "perinteisillä" tekniikalla (Cobol, Fortran jne.) toteutettuihin järjestelmiin lisätään oliotekniikalla toteutetut liittymät. Tämä vaatii oliokääre (object wrapper) -tekniikoita, jotka eristävät palvelun toteutuksen sen liittymästä.
- *Kapselointi (encapsulation).* Olion hallinnassa tarvittavat toiminnot on kapseloitu olion sisään. Näin ollen se tulee toimeen melko autonomisesti.

Tällä hetkellä hajautettujen olioiden markkinoita hallitsevat kaksi eri määrittelyä: Object Management Groupin CORBA (Common Object Request Broker Architecture) ja Microsoftin COM (Component Object Model). (Luoma 1999, 168-169)

CORBA on käyttöjärjestelmä- ja ohjelmointikieliriippumaton komponenttiarkkitehtuuri, jonka avulla voidaan ohjelmoida hajautettuja komponenttipohjaisia sovelluksia. CORBA-sovelluksessa eri komponentit voivat sijaita verkon eri koneilla. Komponenttia käyttävän asiakkaan ei tarvitse tietää komponenttien sijainnista mitään. Hajautus tuo mukanaan myös sen edun, että enemmän resursseja vaativat komponentit voidaan sijoittaa vaikka jokainen omalle koneelleen. CORBA-arkkitehtuuri myös eristää ohjelmoijilta matalan tason verkko-ohjelmoinnin kokonaan. CORBA on saavuttanut markkinoilla suosiota ja sillä on toteutettu liiketoimintaan liittyvien sovelluksien lisäksi esimerkiksi telekommunikaatioon liittyviä sovelluksia, kuten verkon ja laitteiden hallintaa, terveydenhuollon järjestelmiä, ilmatilanvalvontajärjestelmiä, älyverkkoja ja sulautettuja järjestelmiä. (Niskanen, Kontio ja Vierimaa 2000, 559).

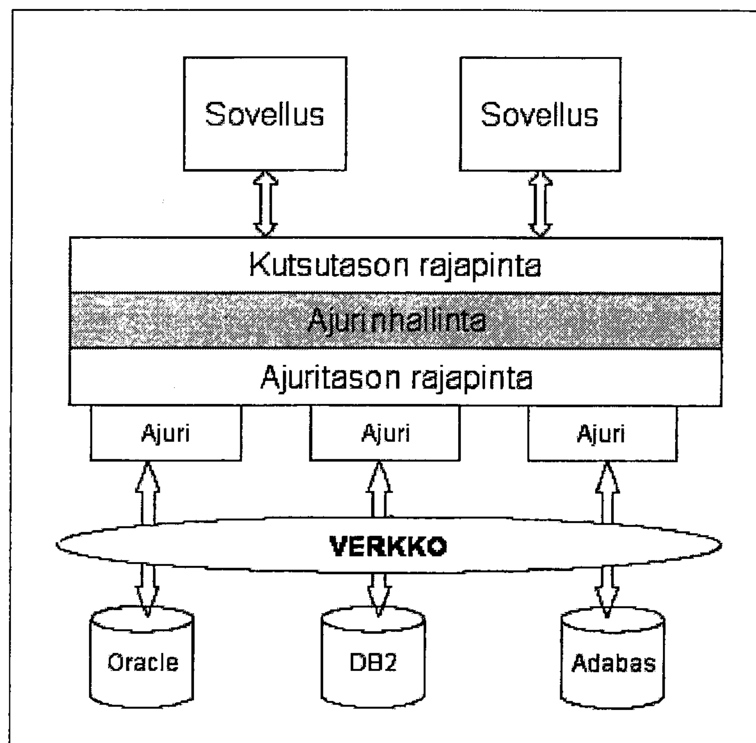
COM on Microsoftin kehittämä komponenttiteknologia ja DCOM (Distributed Component Object Model) sen laajennus, joka sopii hajautettujen komponenttiarkkitehtuurien rakentamiseen. COM on varsinaisesti määritelmä (ja osittain myös toteutus), jonka avulla voidaan tehdä komponenttikehitystä Microsoft Windows -ympäristöissä. Tässä on sekä hyvät että huonot puolensa. Hyvää on se, että COM-komponentit pystyvät keskustelemaan hyvin luontevasti käyttöjärjestelmän kanssa ja kutsumaan siellä toteutettuja palveluja. Huonoa on se, etteivät COM-komponentit toimi käytännössä missään muualla kuin Microsoft-alustoilla. COM-komponentit eivät myöskään ole suoraan yhteensopivia muiden yrityksien kehittämien tekniikoiden ja teknologioiden kanssa. (Niskanen ym. 2000, 547)

3.2.5 Tietokantaorientoituneet väliohjelmistot

Yhteys tietokantoihin on yksi EAI:n oleellisimmista asioista, erikoisesti tietokantason integroinnissa. Tietokantaorientoituneen väliohjelmiston tarkoituksena on tarjota yhteys yhteen tai useampaan tietokantaan, riippumatta siitä millä mallilla (moniulotteinen, relaatio- tai oliotietokanta) tietokanta on toteutettu.

Tietokantaväliohjelmistot voidaan jakaa kahteen eri tyyppiin: *kutsutason* liittymiin (Call Level Interfaces) ja *natiiveihin* (native) tietokantaväliohjelmistoihin. Kutsutason liittymät mahdollistavat yhteydet eri tietokantoihin yhden yleisen liittymän kautta. Esimerkiksi jos tietoa on talletettu Adabas-, Oracle- ja DB2-tietokantoihin, saadaan pääsy samanaikaisesti näihin tietokantoihin yhden yleisen liittymän kautta (KUVIO 7). Tämä helpottaa huomattavasti integraatiotyötä. Tunnetuimpia kutsutason liittymiä ovat ODBC (Open Database Connectivity) ja JDBC (Java Database Connectivity). Natiivilla väliohjelmistolla tarkoitetaan ohjelmistoja, joilla saadaan yhteys vain tiettyyn

tietokantaan tietyllä ohjelmointikielellä. Esimerkiksi Sybase on toimittanut C++ -kielisen väliohjelmiston Sybasen tietokantaan. Natiivilla väliohjelmistolla saavutetaan parempaa suorituskykyä ja käyttömahdollisuus tietokantariippuvaisiin ominaisuuksiin (esim. tallennetut proseduurit ja herättimet). Haittana tästä on sovelluksen huono siirrettävyys. Jos tietokantaa vaihdetaan, joudutaan myös vaihtamaan väliohjelmisto ja tekemään sovellukseen muutoksia, jotka saattavat olla hyvinkin työläitä. (Linthicum 2000, 191-195)



KUVIO 7. Tietokantason väliohjelmistoarkkitehtuuri (Linthicum 2000, 193)

ODBC (Open Database Connectivity) on Microsoftin määrittelemä matalan tason tietokantariippumaton liittymä (API), jonka avulla voidaan kommunikoida tietokantapalvelimien kanssa. ODBC-liittymän ansiosta sitä hyödyntävät sovellukset eivät ole riippuvaisia käytetystä tietokantapalvelimesta. ODBC noudattaa nelitaso-arkkitehtuuria (KUVIO 7). Ylimmällä tasolla on tietokantasovellus. Toisella tasolla on ajurinhallinta (Driver Manager).

Ajurinhallinta päättää, mitä tietokantaan käytetään ja lataa tietokantakohtaisen ajurin. Tämän lisäksi ajurinhallinta välittää ODBC-kutsuja oikeille ajureille, suorittaa tarkistuksia virhetilanteiden varalta (esimerkiksi kutsuparametrien oikeellisuus) ja tarvittaessa pitää kirjaa ajurikutsuista. Kolmannen tason muodostaa tietokantatoimittajan tarjoama tietokantakohtainen ajuri. Alimmalla tasolla on varsinainen tietokannanhallintajärjestelmä ja tietokanta. (Nance 1996)

JDBC (Java Database Connectivity) on universaali ja toimittajariippumaton standardi, jonka mukaan Java-sovellukset voivat operoida relaatiotietokantojen kanssa. JDBC-määritykset ovat hyvin pitkälti samanlaisia Microsoftin ODBC-määritysten kanssa. JDBC:tä käsitellään tarkemmin luvussa 4.4.

Linthicumin (2000) mukaan tietokantaväliohjelmistojen tärkeimmät ominaisuudet ovat:

- Kyky konvertoida sovellusten käyttämä kieli tietokannan ymmärtämään kieleen (esim. SQL).
- Kyky lähettää kyselyjä tietokannalle verkon yli.
- Kyky suorittaa kyselyjä kohdetietokannassa.
- Kyky siirtää kyselyn vastaus verkon yli kutsuvalle sovellukselle ja konvertoida vastaus sovelluksen ymmärtämään muotoon
- Palvelut kuorman tasaukseen säikeiden jakamiseen (thread pooling).

3.3 Yhteenveto väliohjelmistoista

Väliohjelmistot ovat hyvin keskeisessä osassa yrityksen sovellusten integraatiossa: lähes kaikki integrointiratkaisut perustuvat johonkin väliohjelmistoon. Markkinoilla on tarjolla useita eri väliohjelmistoratkaisuja ja moneen eri tarpeeseen. Integroinnin kannalta on tärkeätä selvittää, tarvitaanko asynkronista vai synkronista kommunikointia sovellusten välillä ja onko

integrointi tyypiltään pisteestä pisteeseen vai monesta moneen -mallin mukaista. Nämä tiedot auttavat jo paljon väliohjelmistojen valinnassa. Yksinkertaisimmissa integrointitapauksissa selvitään tietokanväliohjelmistoissa. Asynkronisen kommunikoinnin ollessa ehdoton vaatimus jää vaihtoehdoksi sanomapohjaiset väliohjelmistot ja sanomanvälittäjät. Uudet sovelluspalvelimet ja komponenttimallit (esim. Java EnterpriseBeans) ovat vallanneet tapahtumamonitoreilta ja hajautetuilta olioilta markkinoita.

4. JAVA 2 ENTERPRISE EDITION

Tässä luvussa esitellään lyhyesti Javan historiaan, Java 2 Enterprise Editionin (J2EE) sisältö ja sovellusintegroinnin kannalta keskeisimmät sovellusliittymät. Tämän luvun tarkoituksena on antaa vain yleiskuva Javasta ja perehtyä hieman tarkemmin J2EE-ohjelmointialustan sisältöön. Tämän luvun tarkoituksena ei ole selvittää Java-ohjelmointikieleen liittyviä piirteitä kuin niiltä osin, joilta se on välttämätöntä tutkielmassa esitettyjen asioiden ymmärtämisen takia.

4.1 Java

Syksyllä 1991 Sun Microsystems aloitti Green-nimisen projektin, jonka tarkoituksen oli luoda C- ja C++-kieliin pohjautuva uusi kulutuselektroniikkaa ohjaava olio-ohjelmointikieli. Green-projekti sai aikaiseksi Oak-ohjelmointikielen esiversion, mutta tutkimusprojekti keskeytettiin, kunnes WWW:n (World Wide Web) ja graafisten selaimien nopeasti kasvava suosio sai Sun Mircosystemsin käynnistämään projektin uudelleen. Ohjelmointikieltä muokattiin soveltuvaksi WWW:ssä käytettäväksi laitteistoriippumattomaksi graafisten sovellusten ohjelmointikieleksi. Samalla ohjelmointikielelle annettiin myyvempi nimi: Java. (Niskanen ym. 2000, 1)

Java on C- ja C++-kieliä hyvin paljon muistuttava puhdas olio-ohjelmointikieli, jonka ohjelmoinnissa käytetään alusta alkaen luokkia ja olioita. Javasta on pyritty poistamaan C- ja C++-kielten virhealttiita ominaisuuksia, kuten luokkien moniperiytyvyys ja osoittimet. Javassa lähdekoodi käännetään laiteriippumattomaksi tavukoodiksi, jota ajetaan tulkin eli virtuaalikoneen (Java Virtual Machine) avulla. Kielen tulkattavuus tekee siitä hitaamman kuin konekielelle käännetystä tavallisesta ohjelmasta, mutta tulkkaamisella saavutetaan se etu, että ohjelmat ovat käyttöjärjestelmästä riippumattomia.

Jokainen ympäristö tarvitsee kuitenkin oman kuhunkin järjestelmään sovitetun virtuaalikoneensa. (Vesterholm ja Kyppö 2001, 19)

Vuonna 1995 Sun Microsystems julkaisi ensimmäisen version Java-kehitysympäristöstä. Javan suosio kasvoi nopeasti, ja siihen tuli huomattavasti lisää luokkakirjastoja. Yksi ohjelmointialusta ei kuitenkaan sopinut kaikille, joten Sun Microsystems päätti ryhmitellä Java-kehitysympäristön uudelleen. Ohjelmointialustalla (platform) tarkoitetaan tässä tutkielmassa sellaista kehitys- ja suoritusympäristön määritelmää, jonka avulla voidaan kehittää ja suorittaa sovelluksia. Vuoden 1999 loppupuolella julkaistiin uusi kolmitasoinen Java 2 -määritelmä. Tunnetuin ja ehkä käytetyin näistä on Java 2 Standard Edition (J2SE), joka korvaa aiemman Java Development Kitin (JDK). J2SE-ohjelmointialusta sisältää kaikki "tavallisten" Java-ohjelmien ja -sovelmien toteuttamiseen tarvittavat sovellusliittymät. J2SE-ohjelmointialustaa käytetään esimerkiksi käyttöliittymien toteuttamiseen. Java 2 Micro Edition (J2ME) on vastaavasti erilaisiin mobiileihin ja pieniin laitteisiin, esimerkiksi kämmenmikroihin ja matkapuhelimiin tarkoitettu Java-ohjelmointialusta, joka sisältää vain erittäin rajoitetun osajoukon kaikista Javan ominaisuuksista. EAI:n kannalta mielenkiintoisin näistä ohjelmointialustoista on Java 2 Enterprise Edition (J2EE). J2EE:n sisältöön perehdytään tarkemmin kappaleessa 4.2.

Sun Microsystem tarjoaa veloituksetta sovelluskehittäjien käyttöön näiden kolmen ohjelmointialustojen määritysten mukaiset toteutukset Windows-, Solaris- ja Linux-käyttöjärjestelmäympäristöihin. Java-ohjelmointiin ei siis tarvitse välttämättä maksullisia ohjelmistoja: ohjelmoinnin voi aloittaa Sun Microsystemsin tarjoamilla ohjelmointilustoilla ja tavallisella tekstieditorilla. Java-ohjelmointiin on myös tarjolla useita kaupallisia tuotteita ja ohjelmointia helpottavia työkaluja kuten Borlandin JBuilder, jonka graafiset ominaisuudet helpottavat esimerkiksi käyttöliittymien toteutusta.

4.2 Yleistä J2EE-ohjelmointialustasta

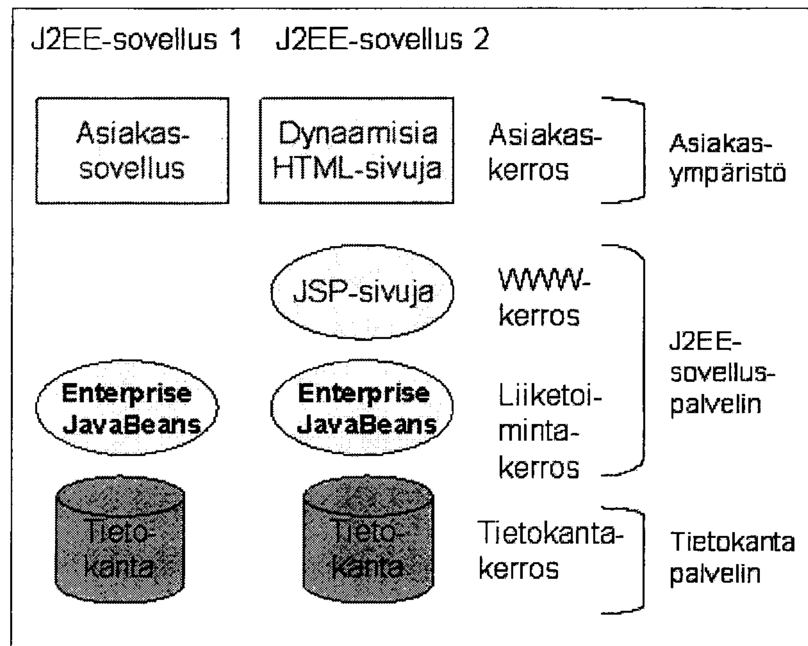
Java 2 Enterpise Edition (J2EE) on ohjelmointialusta, joka sisältää kaikki hajautettujen ja monikerroksisien liiketoimintaa tukevien järjestelmien toteuttamiseen tarvittavat sovellusliittymät. J2EE:n tarkoituksena on nimensä mukaisesti pyrkiä vastaamaan yritystason järjestelmien asettamiin vaatimuksiin. J2EE sisältää myös Java 2 Standard Editionin ominaisuuksia, kuten ”Kirjoita kerran, aja kaikkialla” -siirrettävyyden, JDBC-sovellusliittymän tietokantojen käsittelyyn ja CORBA-teknologiaa, jolla voidaan kommunikoida yrityksen CORBA-sovellusten kanssa.

Shannonin (2001) mukaan J2EE-arkkitehtuuri on suunniteltu vastaamaan tämän päivän yrityksien sovellusten tarpeita, joita ovat: korkea *käyttöaste*, *tietoturvaa*, jolla suojataan käyttäjien yksityisyys ja yrityksen tietoja ulkopuolisilta, *luotettavuus* ja *skaalautuvuus*, joilla taataan tapahtumien virheetön ja nopea suoritus. Yrityksien sovellukset kehitetään usein perustuen monikerrosarkkitehtuuriin, jossa välikerroksella yhdistetään uusien palvelujen data ja liiketoimintalogiikka olemassa oleviin järjestelmiin. Ensimmäinen kerros (käyttöliittymä) toteutetaan usein kehittyneillä web-teknologioilla, joilla päästään helposti käsiksi kompleksiseen liiketoimintalogiikkaan ja vähennetään merkittävästi ylläpidon ja käyttäjien koulutuksen tarvetta. J2EE pyrkii saavuttamaan näitä etuja neljällä eri standardilla kokonaisuudella:

- J2EE-alusta (Platform). Standardi alusta, jossa säilytetään J2EE-sovelluksia.
- J2EE-yhteensopivuuden todentamistestit (Compability Test Suite). Yhteensopivuuden todentamistesteillä voidaan varmistua siitä, että jokin tietty määritelmän toteutus on varmasti J2EE-määrityksien mukainen.
- J2EE-esimerkkitoteutukset (Reference Implementation). Esimerkkitoteutukset tarjoavat prototyyppejä ja toiminnallisen määrittelyn J2EE-sovelluksista.

- J2EE-hahmotelmat (Blueprints). Hahmotelmat esittävät hyviä käytäntöjä ja malleja J2EE-sovelluskehityksestä.

J2EE-sovellukset käyttävät monikerroksista hajautettua sovellusmallia. Tällä tarkoitetaan sitä, että sovelluslogiikka on jaettu toiminnallisuuden perusteella eri komponentteihin, jotka voivat sijaita eri koneilla. J2EE-sovellukset voidaan jakaa kolmeen tai neljään eri kerrokseen (KUVIO 8). Ensimmäisessä kerroksessa on asiakassovellus tai dynaamisia HTML-sivuja, jotka toimivat sovelluksen käyttöliittymänä. Mikäli sovelluksessa käytetään dynaamisia HTML-sivuja asiakastasolla, tarvitaan toiselle tasolle WWW-kerros, jota käytetään tuottamaan dynaamisia HTML-sivuja. Liiketoimintalogiikka sijaitsee toisessa tai kolmannessa kerroksessa riippuen siitä, käytetäänkö WWW-kerrosta. Alimpana sijaitsee tietokantakerros. (Pawlan 2001)



KUVIO 8. Monikerrosovelluksia (Pawlan 2001)

J2EE-ohjelmointialustaan on paketoitu yrityksen sovellusten kehittämiseen tarvittavia Java-sovellusliittymiä. J2EE:n sisältämät sovellusliittymät on esitelty taulukossa 1. Osa J2EE:n sovellusliittymistä esiintyy myös Java 2 Standard

Editionissa. Uusimpana tulokkaana on odotettu JCA 1.0 -määritys, joka julkaistiin syksyllä 2001.

TAULUKKO 1. J2EE:n sisältämät sovellusliittymät (Shannon 2001, 4-10; Rana ja Collins 2001)

Sovellusliittymä	**Kuvaus**
EJB	*Enterprise JavaBeans* on palvelintason komponenttimallin määritys. EJB-komponenteilla toteutetaan yleensä J2EE-sovelluksen liiketoimintalogiikka.
JNDI	*Java Naming and Directory Interface* on nimi- ja hakemistopalvelun sovellusliittymä, jolla paikallistetaan komponentteja ja resursseja.
RMI-IIOP	*Remote Method Invocation* on hajautettujen Java-olioiden kommunkointiprotokolla. *Internet Inter-ORB Protocol* on CORBA-standardin mukainen kommunikointiprotokolla.
JavaIDL	*JavaIDL:n* avulla voidaan Java-oliot määritellä CORBA-ympäristöön kuuuluvaksi. JavaIDL:n ja IIOP:n avulla Java-ohjelmat voivat kommunikoida myös muilla ohjelmointikielillä tehtyjen CORBA-sovellusten kanssa.
Java Servlets	*Java Servletit* ovat palvelintason komponentteja, jotka tarjoavat CGI-ohjelmien tapaisia palveluita asiakassovellukselle. Servletit voivat käsitellä HTTP-palvelupyyntöjä ja voivat tuottaa tuloksenaan dynaamisia HTML-sivuja.
JSP	*Java Server Pages* on laajennus Servletteihin. JSP-sivut ovat HTML-sivuja, joihin voidaan upottaa Java-koodia. Ajonaikana sovelluspalvelin kääntää JSP-sivun servletiksi ja suorittaa näin saadun servletin.

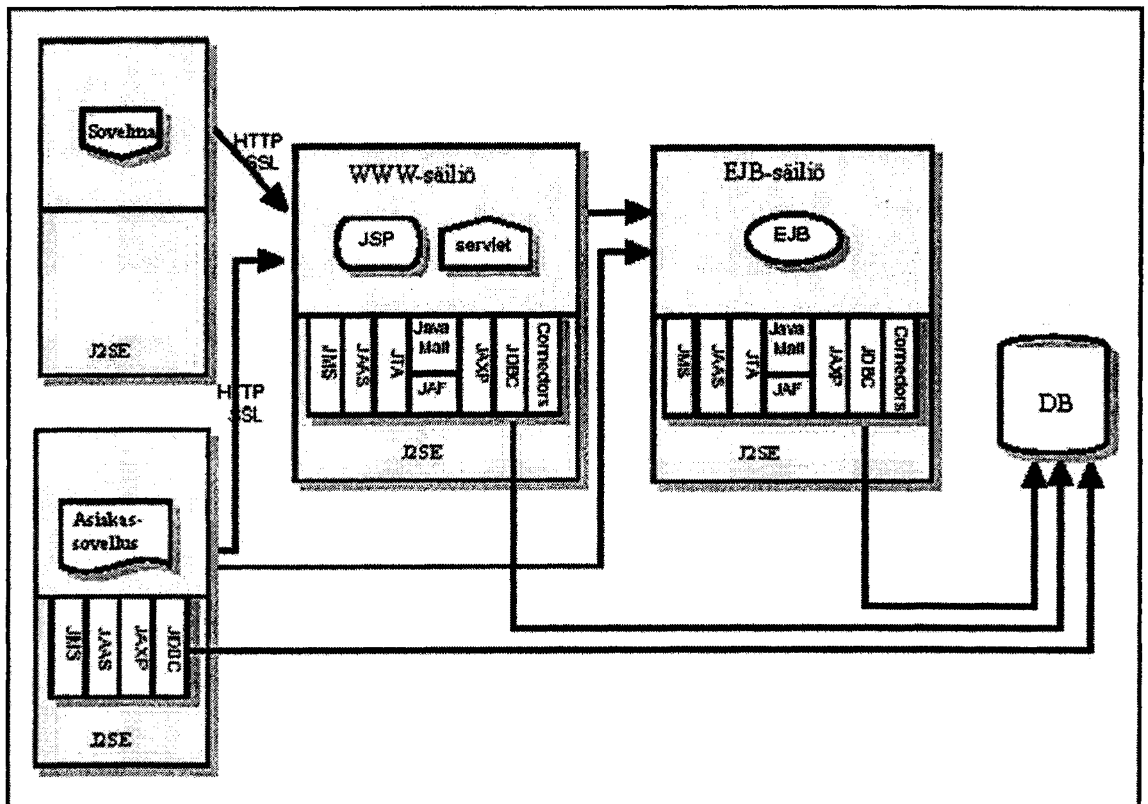
(jatkuu)

TAULUKKO 1. (jatkuu)

JMS	*Java Message Service* tukee asynkroonista ja sanomapohjaista sovellusten välistä kommunikointia, käyttäen jono tai julkaisuviestintämallia.
JTA, JTS	*Java Transaction API* ja *Java Transaction Service* mahdollistavat luotettavan tapahtumapohjaisten tehtävien hallinnan.
JDBC	*Java Database Connectivity* tarjoaa yleisen yhteyskäytännön, jonka välityksellä Java-sovellukset voivat operoida relaatiotietokantojen kanssa.
JavaMail	*JavaMail* tarjoaa protokolla- ja alustariippumattoman sovelluskehyksen, joka mahdollistaa sähköposti- ja viestintäsovelluksien kehittämisen.
JAF	*JavaBeans Activation Frameworkia* käytetään tunnistamaan datan tyyppiä. Esimerkiksi JavaMail käyttää JAF:ia tunnistamaan ja käsittelemään MIME-sähköpostiviestejä.
JAXP	*Java API for XML Processing* mahdollistaa sovelluksien jäsentää ja siirtää XML-dokumentteja.
JAAS	*Java Authentication and Authorization Service* tarjoaa mahdollisuuden autentikoida ja auktorisoida sovelluksen käyttäjiä ja käyttäjäryhmiä.
JCA	*J2EE Connector Architecture* mahdollistaa J2EE-sovellusten kommunikoinnin esimerkiksi vanhojen operationaalisten tietojärjestelmien a toiminnanohjausjärjestelmien kanssa.

J2EE-määritelmä esittää neljä erityyppistä suoritusympäristöä, joita kutsutaan säiliöksi (container) riippumatta suoritettavasta komponentista (KUVIO 9).

Säiliöt ovat: sovelma-, asiakassovellus-, WWW- ja EJB-säiliö. Kuviossa 9 jokainen säiliö on jaettu kahteen osaan: Säiliön yläosa kuvaa J2EE:n suoritusympäristöä ja ympäristön tukemia sovelluskomponentteja. Säiliön alaosassa kuvataan sen tarjoamat sovellusliittymät. Sovellusliittymät on esitelty lyhyesti taulukossa 1. On myös huomioitava, että kuviossa 9 esitetään J2EE-elementtien *loogiset* suhteet, millä ei kuitenkaan tarkoiteta säiliöiden fyysistä jakoa eri palvelimille, prosesseihin, osoiteavaruuksiin tai virtuaalikoneisiin. (Shannon 2001, 3-4)



KUVIO 9. J2EE -arkkitehtuurikaavio (Shannon 2001, 4)

Asiakassovellus- ja sovelmasäiliöt toimivat asiakastasolla. Näistä yksinkertaisin on sovelmasäiliö, jonka ei tarvitse toteuttaa mitään J2EE-sovellusliittymää. Asiakassovellussäiliö on huomattavasti monipuolisempi. Asiakassovellussäiliön täytyy toteuttaa useita sovellusliittymiä, kuten Java Messaging Service ja Java Database Connectivity -liittymät. Asiakassovellukset (Application clients) ovat

Java-ohjelmia, joita suoritetaan tyypillisesti PC-koneilta. Asiakassovellukset tarjoavat samanlaisen käyttökokemuksen kuin perinteiset paikalliset sovellukset, mutta ne ovat yleensä vain graafisia käyttöliittymiä, jotka käyttävät välikerroksen palveluita sovellusliittymien kautta. Sovelmat voivat olla tehokkaita graafisia käyttöliittymäkomponentteja, joita tyypillisesti suoritetaan selaimella. Sovelmasäiliön ja muiden säiliöiden välinen kommunikointi suoritetaan HTTP- (Hypertext Transfer Protocol) tai SSL-protokollan (Secure Socket Layer) kautta. (Shannon 2001, 4-5)

WWW- ja EJB-säiliöt toimivat palvelintasolla. WWW-säiliössä suoritetaan JSP- sekä Servlet-komponentteja ja EJB-säiliössä EJB-komponentteja. JSP- ja Servlet-komponenteilla tuotetaan ajonaikaisia HTML-sivuja, jotka toimivat tyypillisesti sovelluksen käyttöliittymänä. EJB-pavut sisältävät tyypillisesti sovelluksen liiketoimintalogiikan. WWW- ja EJB-säiliöt toteuttavat pääasiassa samat sovellusliittymät, mutta EJB-säiliön ei tarvitse tukea Serlvet- ja JSP-sovellusliittymiä.

4.3 Javan nimi- ja hakemistoliittymät

Java Naming and Directory Interface (JNDI) tarjoaa yhtenäisen sovellusliittymän, jonka avulla Java-sovellukset pääsevät käsiksi moniin nimi- ja hakemistopalveluihin. JNDI-sovellusliittymä ei ole riippuvainen nimi- tai hakemistopalvelun fyysisestä toteutuksesta; riittää että palvelu toteuttaa JNDI:n palveluntarjoajan sovellusliittymän. Tämä mahdollistaa useiden eri palveluiden käytön yhdellä sovellusliittymällä. (Sun 1999a, 1-2)

JNDI koostuu kahdesta eri sovellusliittymästä: Service Provider Interface API:sta eli palveluntarjoajan sovellusliittymästä ja Java-sovellusten käyttämästä JNDI-sovellusliittymästä. Palveluntarjoajan sovellusliittymän toteutuksia ovat kaikki sellaiset toteutukset, jotka mahdollistavat pääsyn johonkin tiettyyn nimi- tai

hakemistopalveluun. JNDI-sovellusliittymä sisältää myös tuen ja laajennukset LDAP-hakemistopalvelulle (Lightweight Directory Access Protocol). (Sun 1999a, 1-5)

Java-sovellukset voivat käyttää JNDI-sovellusliittymää hyvin moniin eri tarkoituksiin esimerkiksi EJB-komponenttien tai tietokantayhteyksien paikallistamiseen nimipalvelun avulla tai henkilötietojen hakemiseen hakemistopalvelusta. Esimerkissä 1 on esitetty, kuinka JNDI:tä voidaan käyttää EJB-asiakassovelluksessa.

```
import javax.naming.Context;
import javax.naming.InitialContext;
.
.
.
try{
  Context initial = new InitialContex();
  Object objRef = initial.lookup("HelloWorld");

  HelloWorldHome home =
    (HelloWorldHome) PortableRemoteObject.narrow(
       objRef, HelloWorldHome.class);

  HelloWorld helloref = home.create();
  String greetings = helloref.sayHello();
  System.out.println(greetings);

} catch(Exception e) {}
.
.
```

ESIMERKKI 1. JNDI-palvelun käyttö EJB-asiakassovelluksessa

Esimerkki 1 on osa kuvitteellista EJB-asiakassovelluksesta. Sovelluksessa *InitialContext*-luokka määrittelee *Context*-liittymän, joten näin voidaan kutsua *Context*-luokan *lookup*-metodia. *lookup*-metodilla haetaan *HelloWorld*-niminen olio. Olio palautuu *Object*-yliluokassa, joten se pitää konvertoida halutun tyyppiseksi olioksi, mutta sitä ennen tarkistetaan *narrow*-metodilla voiko olion konvertoida tietyn tyyppiseksi. Näin saadaan JNDI-palvelusta *HelloWorld*-pavun kotirajapinta, jonka *create*-metodin kutsu palauttaa *HelloWorld*-pavun

etärajapinnan. Tämän jälkeen voidaan käyttää *HelloWord*-pavun tarjoamia liiketoimintaloogisia palveluita (*sayHello*-metodi).

4.4 Javan tietokantaliittymä

JDBC (Java Datavase Connectivity) on korkean tason abstraktioon perustuva sovellusliittymä. JDBC tarjoaa tietokannanhallintajärjestelmästä riippumattoman ohjelmointiliittymän tietokantapalvelujen käyttöön. Tämä mahdollistaa sen, että sovellusohjelmoijan ei tarvitse tietää, minkä tietokannanhallintajärjestelmän kanssa on tekemisissä. Tietyn tietokannanhallintajärjestelmän käyttö JDBC-sovellusliittymän kautta edellyttää kuitenkin tietokannanhallintajärjestelmäkohtaisen ajurin hankintaa. Esimerkiksi Oracle toimittaa Oracle-tietokantaan sopivan JDBC-ajurin. Arkkitehtuurisesti JDBC on hyvin samankaltainen kuin Microsoftin määrittelemä ODBC (Open Database Connectivity) (katso luku 3.2.5. ja KUVIO 7).

JDBC-sovellusliittymän keskeisimmät luokat ovat:

- DriverManager
- Connection
- Statement
- PreparedStatement
- Resultset

Pystyäkseen käyttämään tietokantahallintajärjestelmän palveluita on ohjelman käytettävä järjestelmäkohtaista ajuria. Sovelluksessa ajuri otetaan käyttöön DriveManager-luokan avulla. DriverManager tarjoaa myös palvelun yhteyden luomiseksi tietokantaan. Tietokantayhteydellä hallitaan ohjelman ja tietokannan välistä vuorovaikutusta. Tietokannan suorakäytössä yhteys vastaa tietokantaistuntoa, joka alkaa käyttäjän sisään kirjoittautumisesta ja päättyy

istunnon lopetukseen. Samoin tietokantayhteys alkaa yhteyden perustamisella. JDBC:ssä yhteyden perustaminen tarkoittaa Connection-luokan olion luomista. Tietokantaan kohdistuvat operaatiot suoritetaan JDBC:ssä Statement-olioiden avulla. Statement-olio tarjoaa palvelut, esimerkiksi tietokantakyselyiden, tiedon lisäämisen ja päivittämisen suoritukseen. JDBC:ssä kyselyn vastausta käsitellään ResultSet olion palveluilla. Nämä mahdollistavat vastausjoukon rivien käsittelyn rivi kerrallaan. ResultSet tarjoaa myös palveluita metatiedon saamiseksi vastauksesta. Metatietoa ovat esimerkiksi vastauksen sarakkeiden lukumäärä, sarakkeiden nimet ja tietotyypit. Tietokannan käsittelyssä tarvitaan usein kyselyitä, joissa rakenteeltaan samanlainen kysely suoritetaan toistuvasti, mutta esimerkiksi hakuehdossa vaihtuu vakioarvo. Tällainen kysely voidaan parametrisoida niin, että tuo muuttuva vakio annetaan kyselyn parametrina. Parametrisoidut kyselyt valmistellaan käyttöä varten. Valmistelussa ne käännetään valmiiksi odottamaan suoritusta vaihtuvin parametriarvoin. JDBC tarjoaa parametrisoitujen kyselyjen käsittelyyn PreparedStatement-luokan. (Sun 1999b)

JDK 1.1 tai uudempien Java-ympäristöjen mukana tulee myös JDBC-ODBC-silta, joka mahdollistaa Java-sovelluksien pääsyn ODBC-liittymää tukeviin tietokantoihin. JDBC-ODBC-silta kääntää JBDC-operaatiot ODBC-operaatioiksi. On kuitenkin suorituskyvyn ja toimivuuden kannalta suositeltavaa käyttää tietokantatoimittajan tarjoamaa JDBC-ajuria, jos sellainen on saatavilla. (Sun 1999b)

J2EE-sovellukset eivät normaalisti luo tietokantayhteyttä itse, vaan yhteys haetaan JNDI-palvelua käyttäen sovelluspalvelimeen määritellystä yhteysvarastosta (connection pool). Tämä menettely on huomattavasti suorituskykyisempi. Kun sovellus tarvitsee yhteyttä tietokantaan, haetaan valmiiksi avattu yhteys yhteysaltaasta. Kun yhteyttä ei enää tarvita, sitä ei suljeta, vaan se palautetaan yhteysaltaaseen muiden mahdollisten sovelluksien käyttöön.

4.5 Javan tapahtumanhallinnan liittymät

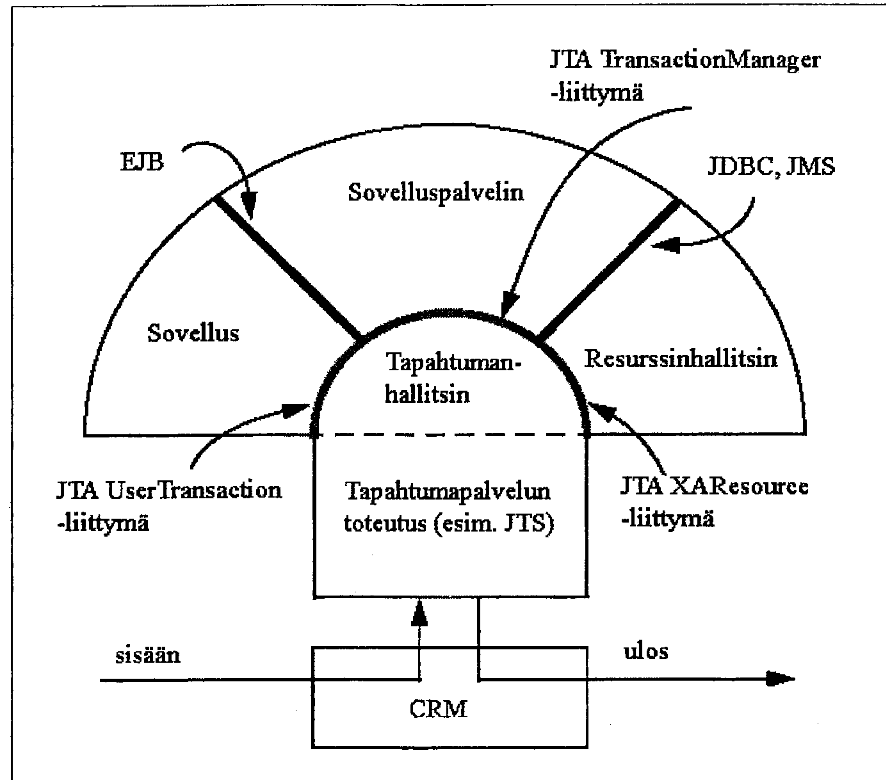
Tapahtumien (transactions) hallinta on yksi palvelinpuolen sovelluskehityksen keskeisimmistä palveluista. Oikein käytettynä tapahtumien avulla voidaan toteuttaa tapahtumakriittisiä sovelluksia, jotka toimivat odotuksien mukaisesti. Tapahtumaan liittyviä tapahtuma- ja eheyskäsitteitä on käsitelty luvussa 3.2.4

JTA (Java Transaction API) ja JTS (Java Transaction Service) ovat J2EE:n tapa toteuttaa tapahtumanhallintaa. JTA on sovellusliittymä, jolla sovelluskehittäjät ja säiliöiden toteuttajat voivat käyttää tapahtuman hallintaan liittyviä palveluita. JTS on määritys tapahtumanhallitsimen (transaction manager) toteutukselle. JTS:n mukainen tapahtumanhallitsin tukee korkeammalla tasolla JTA:ta ja alemmalla tasolla OMG:n CORBA Object Tranaction Service -määritelmää. Alemmalla tasolla JTS siis käyttää Corba-standardin mukaista tapahtumanhallintaa ja määrittelee ylemmällä tasolla liittymät Java-sovelluksille. (Cheung 1999, 4; Cheung & Matena 1999, 6)

JTA määrittelee sovellusliittymät tapahtumanhallitsimen ja hajautettuun tapahtumaan osallistuvien osapuolien (sovellus, resurssihallitsin ja sovelluspalvelin) välille. JTA voidaan katsoa koostuvan kolmesta kokonaisuudesta. Ensimmäinen näistä on korkean tason sovellusliittymä, jonka palveluiden avulla sovellus voi osallistua tapahtumaan. Toisen kokonaisuuden muodostaa X/Open XA -protokollan (teollinen standardi hajautettujen tapahtumien käsittelyyn) mukainen toteutus, jonka avulla resurssihallitsin (resource manager) voi osallistua ulkoisen globaalin tapahtumanhallitsimen hallinnoimiin tapahtumiin. Kolmas kokonaisuus on korkean tason sovellusliittymä, jonka avulla sovelluspalvelin voi käyttää tapahtumanhallitsimen palveluja kontrolloidessaan tapahtumia. (Cheung & Matena 1999, 4)

Hajautettujen tapahtumien hallintaan Java Enterprise -ympäristössä on osallisena viisi eri osapuolta: tapahtumanhallitsin, sovelluspalvelin, resurssihallitsin, tapahtumasovellus ja kommunikoinnin resurssihallitsin (communication resource manager, CMR). Jokainen näistä osapuolista on osallisena hajautetun tapahtuman prosessointiin toteuttamalla eri osan tapahtumaan liittyvistä sovellusliittymistä ja palveluista. (Cheung & Matena 1999, 4-5)

Kuviossa 10 on esitetty edellä mainittujen osapuolien väliset suhteet ja liittymät, joiden välityksellä ne voivat kommunikoida. Tapahtumanhallitsin sisältää palvelut ja hallinnolliset toiminnot, joita tarvitaan tukemaan tapahtuman käsittelyä, synkronisointia, tapahtumaresurssien hallintaa ja tapahtumakontekstin välitystä. Sovelluspalvelimen tehtävänä on tarjota sovelluksille tapahtuman tilan hallintaa tukeva ajonaikainen ympäristö. Resurssihallitsin tarjoaa sovelluksille pääsyn resursseihin resurssisovittimen (resource adapter) kautta. Resurssihallitsin voi olla esimerkiksi tietokannanhallintajärjestelmä, ja resurssisovitin voi olla esimerkiksi JDBC-ajuri. Moderneissa sovelluspalvelimissa suoritettavissa komponenttipohjaisissa sovelluksissa (esimerkiksi EJB-sovellukset) tapahtumanhallinta toteutetaan attribuuttimäärittelyin, ja sovelluspalvelin vastaa tapahtuman oikeellisesta toiminnasta. Vaihtoehtoisesti itsenäiset Java-asiakassovellukset voivat kontrolloida eksplisiittisesti niiden tapahtumien hallintaa sovelluspalvelimen tai tapahtumanhallitsimen toimittamalla korkeantason sovellusliittymällä. Kommunikoinnin resurssihallitsin toimii alimmalla tasolla eri tapahtumanhallitsimien välillä ja tarjoaa sisäänpäin tuleville ja ulospäin meneville kyselyille pääsyn tapahtumapalveluihin. (Cheung & Matena 1999, 5-6)



KUVIO 10. Tapahtumanhallinan osapuolet (Cheung & Matena 1999, 5)

4.6 Enterprise JavaBeans -komponenttimalli

Enterprise JavaBeans (EJB) on ehkä keskeisin J2EE:n tarjoamista määrityksistä. Enterprise JavaBeans on strandardi palvelimen komponenttiarkkitehtuuri J2EE-alustalle, jonka avulla voidaan kehittää hajautettuja olio-orientoituneita liiketoimintasovelluksia Java-ohjelmointikielellä. EJB-teknologialla kehitetyt yrityssovellukset yksinkertaistavat toiminnallisten, skaalautuvien ja siirrettävien monikerrossovellusten suunnittelua. EJB-määrityksen tavoitteena on ollut tehdä läpinäkyväksi alhaisen tason tapahtuman- ja tilanhallinnan, monisäikeisyyden, yhteysaltaan sekä muiden monimutkaisten liittymien yksityiskohdat ja siten mahdollistaa sovelluskehittäjien keskittyä liiketoimintalogiikan ohjelmointiin. EJB-komponentimalli noudattaa myös ”kirjoita kerran, aja kaikkialla” -periaatetta, joten kerran kirjoitettu komponentti voidaan sijoittaa minkä tahansa

toimittajan EJB:tä tukevalle sovelluspalvelimelle. Koska EJB-komponenttimalli on pelkkä määritys, voi kuka tahansa sovelluspalvelimen toimittaja lisätä tuotteeseensa tuen EJB-komponenteille toteuttamalla EJB-komponenttimallin määrityksen mukaisen komponenttien suoritusympäristön.

EJB-määrityksessä on kuvattu kuusi eri roolia, jotka osallistuvat EJB-mallin mukaiseen sovelluskehitykseen. Ensimmäinen näistä kuudesta roolista on tuottaja (provider), jonka tehtävänä on ohjelmoida yksittäisiä komponentteja ja vastata niiden liiketoimintalogiikasta. Toinen rooleista on kokooja (assembler), jonka tehtävänä on yhdistää tuottajien tekemät komponentit toimivaksi kokonaisuudeksi. Sijoittajan (deployer) vastuulla on kokoojan tuottaman sovelluksen käyttöönotosta sovelluspalvelimella, sovelluksen toimintaan liittyvistä palvelimen asetuksista ja resursseista. Sovelluspalvelimen toimittajan (server provider) vastuulla on toteuttaa EJB-määrityksen mukaiset palvelut palvelimelle, kuten tapahtumien ja hajautuksen hallinta. Säiliön toteuttajan (container provider) vastuulla on vastaavasti varsinaisen ajoympäristön tarjoaminen komponenteille sekä sijoittajan tarvitsemien työkalujen toteuttaminen. Sovelluspalvelimen ja EJB-säiliön välille ei ole määritelty liittymää, joten käytännössä käytetään saman toimittajan sovelluspalvelinta ja säiliötä. (Roman, Ambler ja Jewel 2002, 16-21) Komponenttien kehittämisen roolijako saattaa vaikuttaa teennäiseltä, mutta käytännössä olen huomannut, että rooleja ei ole määritelty turhaan. Ne kuvaavat kehityksen vaiheita, ja on tehokkaampaa jakaa kehitykseen osallistuville henkilöille selkeitä vastuualueita, jotka opetellaan hyvin, kuin se että kaikki sovelluskehittäjät tekisivät kaikkien roolien tehtäviä.

EJB-komponentit voidaan jakaa kolmeen eri päätyyppiin: istuntopohjaisiin (session bean), kohdepohjaisiin (entity bean) ja sanomapohjaisiin (message-driven bean) komponentteihin. Istuntopohjaiset komponentit edustavat palvelimella asiakasta. Jokaisella asiakkaalla on oma istuntopohjainen

komponentti. Istuntopohjaiset komponentit sisältävät tyypillisesti liiketoimintalogiikkaa esimerkiksi tilauksen tekemiseen liittyvät toiminnot tai asiakasrekisterin hallinta. Istuntopohjaisia komponentteja on kahdenlaisia, tilattomia ja tilallisia. Tilattomalla istuntopohjaisella komponentilla ei ole nimensä mukaisesti tilaa, joka muuttuisi sen elinkaaren aikana. Tilallinen istuntopohjainen komponentti on vain yhtä asiakasta varten (jokaiselle asiakkaalle luodaan oma versio). Tilallinen komponentti säilyttää metodikutsujen välillä tilansa, minkä vuoksi se pystyy keskustelunomaiseen kommunikointiin. Tilallista komponenttia voidaan käyttää esimerkiksi ostoskoritoteutuksissa. (Niskanen ym. 2000, 644-645) Jokainen asiakas ei tarvitse omia tilattomia istuntopohjaisia komponentteja, säiliössä voi olla esimerkiksi sata ilmentymää tilattomasta komponentista, joiden käyttöä asiakkaat (tilalliset komponentit) vuorottelevat. Tilattomia komponentteja käytetään usein myös tietokantaloogisena välikerroksena tilallisten (ja myös tilattomien) istuntopohjaisten ja kohdepohjaisten komponenttien välissä.

Kohdepohjaiset komponentit edustavat jotain tallennettavaa yksikköä, kuten asiakasta tai tilausta. Kohdepohjaisten komponenttien tieto tallennetaan yleensä relaatiotietokantaan, johon kohdepohjaiset komponentit tarjoavat oliomaisen liittymän. Kohdepohjaiset komponentit antavat sovelluskehittäjille mahdollisuuden keskittyä liiketoimintalogiikkaan ja jättää tiedon hakemisen ja tallentamisen sekä tapahtumien hallinnan vähemmälle huomiolle. Kohdepohjaisia komponentteja on kahdenlaisia: säiliön ylläpitämiä (Container-Managed Persistence, CMP) ja komponentin ylläpitämiä (Bean-Managed Persistence, BMP) kohdepohjaisia komponentteja. BMP-komponenttien koodiin täytyy sovelluskehittäjän itse kirjoittaa koodi, joka lataa ja tallentaa komponentin sisältämän tiedon. CMP-komponenttien tallentamisesta ja lataamisesta huolehtii säiliö. (Niskanen ym. 2000, 645). Tyypillisesti yksi kohdepohjainen komponentti vastaa yhtä relaatiotietokannan taulua ja yksi komponentti ilmentymä vastaa

tietokantataulun yhtä riviä, mutta uusi EJB 2.0 -määrittely mahdollistaa myös monimutkaisemmat suhteet taulujen ja komponenttien välillä.

EJB-komponentit koostuvat kolmesta osasta: etärajapinnasta (remote interface), kotirajapinnasta (home interface) ja varsinaisesta komponenttiluokasta. Etärajapinta määrittelee EJB-komponentin ulkoisen rajapinnan eli rajapinnan, joka sisältää komponentin tarjoaman liiketoimintalogiikan metodien esittelyt. Kotirajapinta määrittelee rajapinnan, jonka avulla asiakas voi luoda, etsiä ja poistaa komponentteja. Kohdepohjaisten komponenttien kotirajapinnassa on myös metodeja tiedon hakemiseen.

Sanomapohjainen komponentti on uusi komponenttityyppi, joka on esitelty EJB 2.0 -määrittelyssä. Sanomapohjaiset komponentit poikkeavat hieman muista komponenteista. Sanomapohjaisilla komponenteilla ei ole ollenkaan ulkoisia rajapintoja, joten asiakassovellukset eivät voi käyttää niitä suoraan. Sanomapohjaiset komponentit on suunniteltu käsittelemään asynkronisia Java Messase Service (JMS) -viestejä. Sanomapohjaiset komponentit ovat kuuntelijoita, jotka on rekisteröity kuuntelemaan tiettyä sanomajonoa. Kun sanomajonoon tulee viesti, säiliö herättää sanomapohjaisen komponentin, joka käsittelee viestin. Istuntopohjaisilla komponenteilla voidaan lähettää ja vastaanottaa sanomia synkronisesti, mutta uusi sanomapohjainen komponentti määrittely mahdollistaa sanomien käsittelyn myös asynkronisesti, joka on integroinnin kannalta merkittävä lisäys EJB-määritelmään.

4.7 Yhteenveto J2EE-alustasta

Tässä luvussa esiteltiin lyhyesti Java-ohjelmointikieltä, J2EE-alustaa ja sen sisältämiä liittymiä. J2EE-alusta tarjoaa monipuolisen joukon liittymiä hajautettujen ja monikerroksisien yrityksien sovellusten kehittämiseen. J2EE-ohjelmointialusta on suunniteltu vastaamaan tämän päivän yrityksen sovelluksien

vaatimuksiin, kuten korkea käyttöaste, tietoturvallisuus, luotettavuus ja skaalautuvuus.

J2EE-alusta muodostuu suuresta joukosta eri teknologioiden määrityksiä ja eri J2EE-komponenttien suoritusympäristöistä eli säiliöistä. Säiliöiden on tarjottava erilaisia sovellusliittymiä ja tuettava siten eri teknologioita säiliöstä riippuen. Säiliöitä on määritelmässä esitetty neljä. Nämä säiliöt ovat sovelma-, asiakassovellus,- EJB- ja WWW-säiliöt, kahden viime mainitun ollessa palvelimella toimivia ja kahden ensiksi mainitun asiakassovelluksen päässä toimivia säiliöitä.

Luvussa esiteltiin lyhyesti kaikki J2EE-määrityksen sisältämät teknologioiden määritykset ja lisäksi tarkemmin muutamia tärkeimpiä liittymiä. Keskeisen osan J2EE-ohjelmointialustasta muodostaa nimipalvelu Java Naming and Directory Interface (JNDI), jonka avulla erilaisten resurssien paikallistaminen on mahdollista hajautetuissa järjestelmissä. Tietovarastojen (relaatiotietokantojen) hallinta on hyvin oleellinen osa J2EE-sovelluksia. J2EE-alusta tarjoaa tietokantariippumattoman Java Database Connectivity (JDBC) -sovellusliittymän tietovarastojen käsittelyyn. Tapahtumanhallinta J2EE-alustassa on toteutettu korkean tason sovellusliittymän Java Transaction API:n (JTA) sekä Java Transaction Servicen (JTS) avulla. JTS:n määritys esittää, kuinka yhteistoiminta alemman tason palvelun eli OMG:n Object Transaction Servicen kanssa toteutetaan. Vastaavasti JTA määrittelee korkean tason sovellusliittymän, jonka avulla sovellusohjelmat voivat käyttää tapahtumanhallitsimen palveluja hyväkseen. Viimeisenä on esitelty Enterprise JavaBeans (EJB) -komponenttimalli. EJB on strandardi palvelimen komponenttiarkkitehtuuri J2EE-alustalle, jonka avulla voidaan kehittää hajautettuja olio-orientoituneita liiketoimintasovelluksia Java-ohjelmointikielellä. EJB-teknologia yksinkertaistaa toiminnallisten, skaalautuvien ja siirrettävien monikerrossovellusten suunnittelua.

5. J2EE CONNECTOR ARCHITECTURE

J2EE Connector Architecture on edellisessä luvussa esitellyn J2EE-sovellusalustan uusimpia ja sovellusten integroinnin kannalta hyvin merkittävä lisäys, jolla parannetaan Java-sovellusten liitettävyyttä olemassa oleviin tietojärjestelmiin. Tässä luvussa esitellään JCA-määrittelyn sisältö pääperiaatteiltaan.

Yrityksien tarpeet sovellusarkkitehtuuria valittaessa vaihtelevat. Yhteistä on kuitenkin halu tuottaa helposti saatavissa olevia, tietoturvallisia, luotettavia ja skaalautuvia palveluja. Java 2 Enterprise Edition vastaa näihin vaatimuksiin varsin hyvin, joten se onkin saavuttanut suurta suosiota yrityksien sovellusten sovellusalustana. J2EE määrittää standardin yrityssovellusten kehittämiseen ja käyttöönottoon. Monitasoisten yrityspalvelujen rakentaminen J2EE-teknologialla alentaa kustannuksia ja vähentää monimutkaisuutta. Lisäksi eri toimintamalleilla voidaan parantaa joustavuutta, suorituskykyä, ylläpidettävyyttä, tietoturvaa ja sovelluksenhallintaa. On siis varsin perusteltua kehittää uudet yrityksen sovellukset perustuen J2EE-teknologiaan, mutta yrityksen sovellusten tulee myös usein sopia yhteen yrityksen olemassa olevien tietojärjestelmien kanssa. Yrityksen sovellusten integroinnin ongelmaa J2EE:n osalta helpottaa JCA-määrittely.

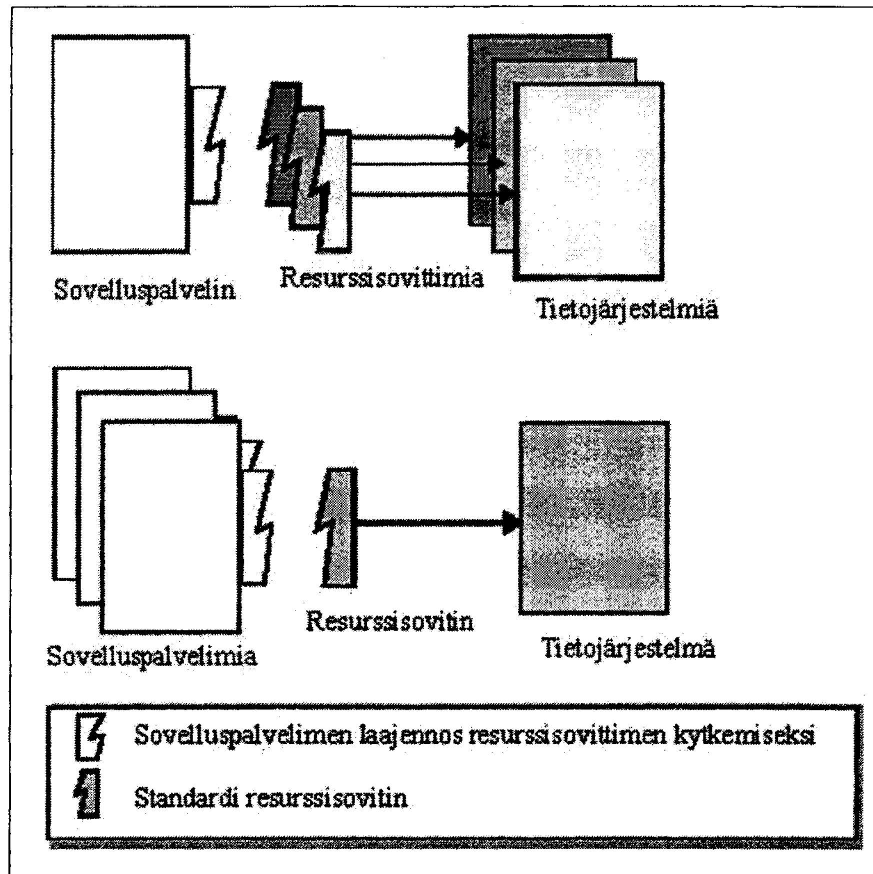
Selvyyden vuoksi jatkossa käytetään nimitystä ”integroitava tietojärjestelmä” jo olemassa olevasta yrityksen sovelluksesta, joka on integroinnin kohteena. Vastaavasti uudesta kehitettävästä J2EE-sovelluksesta, joka käyttää olemassa olevia integroitavia järjestelmiä, käytetään nimitystä ”sovellus” tai ”sovellus komponentti”.

5.1 JCA yleisesti

Ennen syksyllä 2002 julkaistua JCA-määrittelyä ei ollut olemassa Java-alustalle standardia arkkitehtuuria, joka tarjoaisi ratkaisun Java-sovellusten integroimiseen toiminnanohjausjärjestelmiin, keskustietokonejärjestelmiin (main frame) tai perinnejärjestelmiin (legacy systems), jotka eivät ole ohjelmoitu Java-kielellä. Tästä johtuen monet tietojärjestelmien ja sovelluspalvelimien toimittajat ovat käyttäneet standardoimattomia ja toimittajakohtaisia ratkaisuja integroidessaan järjestelmiä, mutta näiden ratkaisujen ongelmana on kertakäyttöisyys.

JCA-määrittelyä käyttäen tietojärjestelmien toimittajien ei enää tarvitse mukauttaa tuotteitansa jokaiseen eri sovelluspalvelimeen liitettäväksi. Samoin sovelluspalvelimien toimittajille riittää, että heidän sovelluspalvelimensa tukee vain yhtä standardia arkkitehtuuria, jota käytetään tietojärjestelmien liittämiseksi sovelluspalvelimeen. JCA-määrittely mahdollistaa tietojärjestelmätoimittajien kehittää tietojärjestelmiinsä standardin sovittimen. Sovitin kytketään J2EE-sovelluspalvelimeen, jolloin on mahdollista luoda yhteys sovelluspalvelimen, yrityksen tietojärjestelmän ja Java-sovellusten välille. Tällainen standardi määrittely siis helpottaa sekä sovelluspalvelimien että tietojärjestelmien toimittajien ja myös sovelluskehittäjien työtä. (Rodoni 2001)

Kuten kuviossa 11 on havainnollistettu, standardi JCA:n mukainen tietojärjestelmäspesifinen resurssisovitin voidaan kytkeä mihinkä tahansa J2EE 1.3 -määrittelyn mukaiseen sovelluspalvelimeen. Samoin useita eri resurssisovittimia voidaan kytkeä yhteen sovelluspalvelimeen, joka tukee standardia JCA-määrittelyä. Standardin määrittelyn hyöty voidaan todeta myös matemaattisesti. Jos on m kappaletta sovelluspalvelimia ja n kappaletta integroitavia tietojärjestelmiä, JCA-määrittely vähentää integroinnin ongelman laajuutta $m*n$ määrästä $m+n$ määrään. (Rodoni 2001)



KUVIO 11. Resurssisovittimen liitettävyys (Rodoni 2001)

JCA:lle on asetettu seuraavat tavoitteet (Sharma 2001):

1. JCA-määrittely yksinkertaistaa skaalautuvan, tietoturvallisen ja tapahtumanhallinnallisen resurssisovittimen kehitystyötä useisiin tietojärjestelmiin.
2. JCA-määrittely on riittävän yleinen kattaakseen laajasti eri heterogeenisiä järjestelmiä. Riittävän yleinen arkkitehtuuri mahdollistaa erilaiset toteutusvaihtoehdot eri resurssisovittimille.
3. JCA:n mukainen resurssisovitin ei ole sidottu tietyn toimittajan sovelluspalvelimeen. JCA on yhteensopiva eri toimittajien J2EE-sovelluspalvelimien kanssa.
4. Se sisältää standardin asiakasliittymän, joka on yhteinen eri tietojärjestelmien resurssisovittimille.

5. JCA on helppo ymmärtää ja noudattaa. Tällä helppoudella tarkoitetaan sitä, että JCA esittää vain vähän uusia konsepteja ja täyttää minivaatimukset, jotta se olisi sopiva eri integraatioskenaarioihin ja ympäristöihin.
6. JCA määrittää sopimukset ja vastuut eri rooleille, jotka vastaavat eri osa-alueista standardin mukaisessa integroinnissa.

5.1 Roolit JCA-pohjaisessa sovelluskehityksessä

JCA-määrittelyssä (Sharma 2001, 12-15) on esitelty kahdeksan eri roolia ja vastuualuetta. Resurssisovittimen (resource adapter) toimittaja on integroitavan tietojärjestelmän asiantuntija ja hänen vastuulla tuottaa resurssisovitin tietojärjestelmään. Koska tämä rooli on hyvin tietojärjestelmäspesifinen, resurssisovittimen toimittaja on yleensä tietojärjestelmän toimittaja. Resurssisovittimen toimittaja voi olla myös jokin kolmas osapuoli, esimerkiksi yritys, joka on erikoistunut kehittämään resurssisovittimia ja niihin liittyviä työkaluja eri tietojärjestelmiin.

Sovelluspalvelimen toimittaja toimittaa toteutuksen J2EE-määrityksen mukaiselle sovelluspalvelimelle. Tyypillinen sovelluspalvelimen toimittaja on käyttöjärjestelmä-, väliohjelmisto- tai tietokantatoimittaja. Sovelluspalvelimen toimittajan rooli on tyypillisesti sama kuin säiliön (container) toimittajan. Säiliön toimittajan vastuulla on toimittaa säiliö (ympäristö) erityyppisille sovelluskomponenteille. Säiliöitä on käsitelty kappaleessa 4.2.

Sovelluskomponentin toimittaja toimittaa sovelluskomponentin, jolla on pääsy yhteen tai useampaan tietojärjestelmään, jonka toiminnallisuutta komponentti käyttää. Sovelluskomponentin toimittaja on kehitettävän sovelluksen asiantuntija, eikä hänen tarvitse vastata järjestelmätason ohjelmoinnista, kuten tapahtumanhallinnasta, suojauksista tai hajautuksesta, mutta hän käyttää näitä

säiliön palveluita "läpinäkyvästi". Sovelluskomponentin toimittaja käyttää sovelluskehitystä ja integrointia helpottavia työkaluja avukseen. Työkalujen toimittajan (enterprise tools provider) tehtävänä on toimittaa työkaluja, jotka helpottavat kompleksisia sovelluskehityksen ja sovellusintegroinnin tehtäviä. Tiedonlouhintatyökalujen avulla voidaan tutkia integroitavan järjestelmän datan rakennetta ja laajuutta. Analysoinnin ja suunnittelun työkalut helpottavat suunnittelua integroitavan järjestelmän datan ja toimintojen termejä käyttäen. Lähdekoodia generoivat työkalut auttavat integroitavan järjestelmän datan ja toimintojen kartoittamista Java-luokiksi.

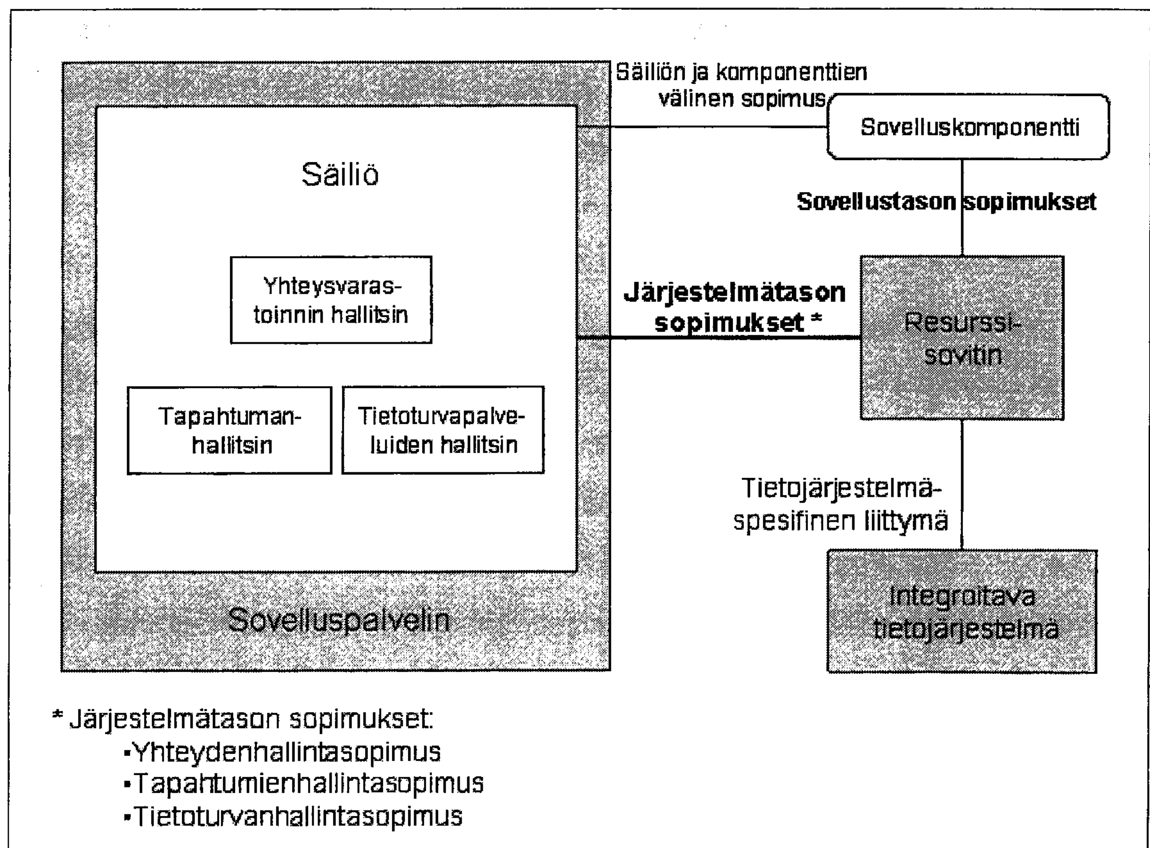
Kokooja (assembler) yhdistää erilaiset sovelluskomponentit laajemmaksi sovelluspalvelimelle asennettavaksi kokonaisuudeksi. Sovelluskomponentin toimittajat toimittavat paketoituja Java-tiedostoja (Java Archive, JAR). Asentaja (deployer) yhdistää nämä JAR-tiedostot yhdeksi tai useammaksi JAR-tiedostoksi sekä kirjoittaa asennuksen kuvaustiedostot (deployment descriptors). Asennuksessa tarvittavat kuvaustiedostot sisältävät palvelimen tarvitsemia tietoja, esimerkiksi komponentin tyyppi ja sen käyttämät resurssit, tapahtumien ja yhteyksien hallinnan asetukset.

Järjestelmän pääkäyttäjä (administrator) on vastuussa järjestelmän asetuksista, hallinnoinnista ja koko infrastruktuurista, johon kuuluu useita sovellussäiliöitä ja eri järjestelmiä. Toiminnallisessa ympäristössä, jossa on useita tietojärjestelmiä, asentajan tulisi tehdä yhteistyötä pääkäyttäjän kanssa. Yhteistyö mahdollistaa monimutkaisten asennustilanteiden ratkaisemisen, joita esiintyy usein komponenttien ja resurssisovittimien asentamisessa.

5.2 Järjestelmä- ja sovellustason sopimukset

Resurssisovitin (resource adapter) on hyvin keskeisessä roolissa sovelluspalvelimilla toimivien sovelluksien integroinnissa ja liitettävyydessä.

Resurssisovitin toimii eräänlaisena kontaktipisteenä sovelluskomponenttien, sovelluspalvelimien ja integroitavan tietojärjestelmän välillä. JCA-määrittelyssä on määritelty erilaisia sopimuksia (contracts), joiden avulla resurssisovitin ja muut komponentit voivat kommunikoida eri osapuolien kanssa. Kuviossa 12 on kuvattu nämä eri komponentit ja niiden väliset vuorovaikutukset. Sovellustason sopimukset määrittävät, kuinka sovelluskomponentit (esimerkiksi EJB:t) kommunikoivat resurssisovittimen kanssa. Järjestelmätason sopimukset määrittävät, kuinka resurssisovitin ja sovelluspalvelin kommunikoivat keskenään. Resurssisovittimen ja integroitavan tietojärjestelmän välillä on tietojärjestelmäspesifinen liittymä, joka määrittää kuinka resurssisovitin on yhteydessä tietojärjestelmään. Säiliön ja komponentin väliset sopimukset on määritelty eri komponenttikohtaisissa määrittelyissä, esimerkiksi EJB-määrittely määrittää EJB-komponentin ja säiliön välisen yhteyden.



KUVIO 12. Resurssisovittimen sopimukset (Sharma ym. 2001, 32)

Resurssisovittimen järjestelmätason sopimukset on jaettu kolmeen osaan, jotka ovat: yhteydenhallintasopimus, tapahtumienhallintasopimus ja tietoturvanhallintasopimus. Tietoturvanhallintasopimus (security management contract) mahdollistaa tietoturvallisen sovellusympäristön luomisen ja suojaa integroitavan järjestelmän hallinnoimia resursseja. Yhteydenhallintasopimus (connection management contract) mahdollistaa sovelluspalvelimen ylläpitää yhteyksiä tietojärjestelmään yhteysvarastossa (connection pool) ja samalla mahdollistaa sovelluskomponenttien ottaa yhteyttä integroitavaan tietojärjestelmään. Yhteyksien säilyttäminen yhteysaltaassa on hyvin tärkeätä erityisesti silloin, kun ollaan luomassa skaalautuvaa sovellusympäristöä, jossa useat asiakassovellukset tarvitsevat yhteyden integroitavaan tietojärjestelmään. (Sharma ym. 2001, 33)

Tapahtumanhallintasopimus (transaction management contract) on sovelluspalvelimen toimittaman tapahtumanhallitsimen ja tapahtumanhallintaa tukevan integroitavan tietojärjestelmän välinen sopimus. Tapahtumanhallintasopimus mahdollistaa sovelluspalvelimen tapahtumanhallitsimen hallinnoida tapahtumia integroitavan tietojärjestelmän resurssihallitsimen ylitse. (Sharma ym. 2001, 33) Sovelluspalvelimen tapahtumanhallintaa on käsitelty kappaleessa 4.5.

JCA määrittää myös sovellustason sopimuksen sovelluskomponentin ja resurssiadapterin välille. Sopimus määrittää asiakasliittymän, jonka avulla sovelluskomponentit pääsevät käsittelemään integroitavan tietojärjestelmän tarjoamia palveluita. Asiakasliittymä voi olla JCA:n mukainen Common Client Interface -toteutus tai liittymäspesifinen tiettyyn resurssisovittimeen ja tietojärjestelmään sopiva. Common Client Interface määrittää yleisen asiakastason liittymän, jolla saadaan yhteys useisiin heterogeenisiin tietojärjestelmiin.

5.4 Yhteyden hallinta

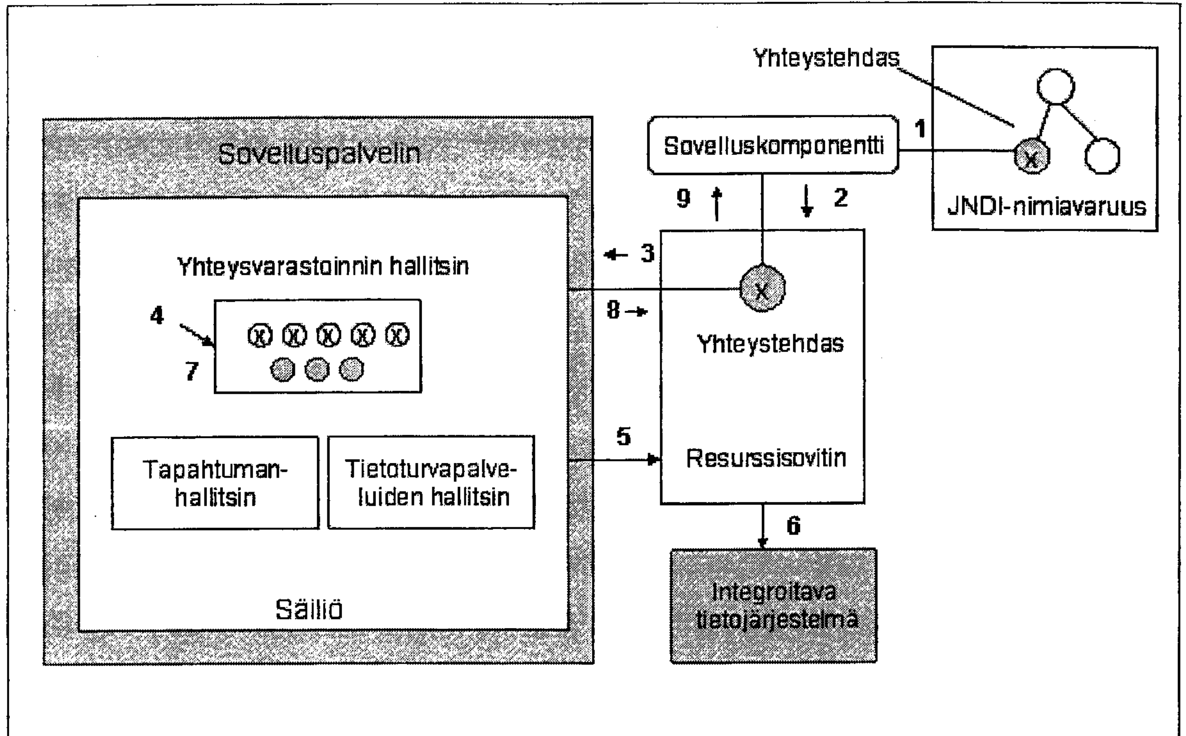
Sovellukset tarvitsevat yhteyden voidakseen kommunikoida integroitavan tietojärjestelmän kanssa. Yhteyden avulla saadaan pääsy integroitavan tietojärjestelmän tarjoamiin resursseihin tai palveluihin. Yhteys voi olla, esimerkiksi JDBC-yhteys relaatiotietokantoihin, Java Message Service -yhteys sanomapalveluihin tai SAP R/3 toiminnanohjausjärjestelmän yhteys.

Yrityksen sovelluksissa yhteyksien varastoinnilla (connection pooling) on keskeinen merkitys. Yhteyksien varastointi on eräs tapa hallita yhteyksiä. Yhteyksien avaaminen ja sulkeminen kuluttavat palvelimen ja järjestelmien resursseja. Siksi on edullisempaa varastoida yhteyksiä ja jakaa niitä tarvitseville sovelluksille sen sijaan, että yhteyttä tarvitseva asiakassovellus avaisi aina uuden yhteyden. Yhdenaikaisten fyysisten yhteyksien määrä integroitavaan tietojärjestelmään on usein varsin rajoitettu, ja samalla tämä rajoittaa yhdenaikaisten asiakasistuntojen määrää. Sovelluspalvelimen hallinnoima yhteysvarasto mahdollistaa yhteyksien jakamisen useiden asiakassovelluksien kesken. Tällä menettelyllä mahdollistetaan useampien yhdenaikaisten asiakasistuntojen määrä. Asianmukainen yhteyksien varastointi johtaa siis yrityksen sovellusten parempaan skaalautuvuuteen ja suorituskykyyn. (Sharma ym. 2001, 45-46)

JCA-määrittely sisältää yhteydenhallintasopimuksen (connection management contract), joka sisältää tuen yhteyden hallinnalle ja yhteyksien varastoinnille. Yhteydenhallintasopimuksen toteuttaminen on resurssisovittimen ja sovelluspalvelimen toimittajien vastuulla, mutta määrittely tarjoaa myös sovelluskehittäjälle tärkeätä tietoa yhteydenhallintasopimukseen perustuvasta ohjelmoinnista. Yhteydenhallintasopimus on resurssisovittimen ja sovelluspalvelimen välinen sopimus, kuinka sovelluspalvelimen tulee toteuttaa yhteyksien varastointi ja hallinnointi standardilla tavalla.

Yhteydenhallintasopimus määrittää lisäksi perusteet, kuinka hallita yhteyttä sovelluksien ja integroitavan tietojärjestelmän välillä. Sovelluspalvelin käyttää yhteydenhallintasopimusta: uusien yhteyksien luomisessa integroitavaan tietojärjestelmään, yhteystehtaan määrittelyssä JNDI-nimiavaruuteen ja löytääkseen sopivan yhteyden olemassa olevista yhteysvarastoista. (Sharma ym. 2001, 47-48)

Sovelluskomponentti, joka tarvitsee yhteyden integroituun tietojärjestelmään, käyttää hyväkseen resurssisovittimen tarjoamia palveluita resurssisovittimen toimittaman liittymän kautta. Integroidussa ympäristössä tapahtuu määrätyt vaiheet, kun sovelluskomponentti yrittää ottaa yhteyden tietojärjestelmään. Näitä tapahtumia on havainnollistettu kuviossa 13. Kuvion 13 esittämässä ympäristössä resurssisovitin on asennettu sovelluspalvelimelle, ja sen sisältämä yhteystehdas määritelty JNDI-nimiavaruuteen. Yhteystehtaaseen on määritelty integroitavan tietojärjestelmän asetuksia, kuten palvelin, tietoliikenneportin numero ja muuta järjestelmäriippuvaista tietoa, jota tarvitaan yhteyttä muodostaessa. (Sharma ym 2001, 47-48)



KUVIO 13. Resurssisovittimen sopimukset (Sharma ym. 2001, 32)

Sovelluspalvelinpohjaisessa integroidussa ympäristössä sovelluskomponentti saa yhteyden seuraavien vaiheiden kautta (Sharma ym. 2001, 48-49):

1. Sovelluskomponentti hakee yhteystehtaan JNDI-liittymän avulla JNDI-nimiavaruudesta.
2. Onnistuneen JNDI-haun jälkeen sovelluskomponentti kutsuu yhteystehtaan metodia, joka palauttaa sovelluskomponentille yhteyden integroituun tietojärjestelmään.
3. Ennen kuin resurssisovittimen toimittama yhteystehdas luo yhteyden, se delegoi pyynnön sovelluspalvelimelle käyttäen yhteydenhallintasopimusta. Sovelluspalvelin tarjoaa palveluita, kuten yhteyden varastointia ja tapahtumien hallintaa, joita resurssisovitin hyödyntää.
4. Kun sovelluspalvelin saa pyynnön luoda yhteyden, se etsii sopivan olemassa olevan yhteyden sovelluspalvelimen yhteysvarastosta. Yhteyksiä yhteysaltaassa kutsutaan hallituiksi yhteyksiksi (managed connection),

jotka vastaavat todellista fyysistä yhteyttä integroituun tietojärjestelmään. Jos sovelluspalvelin löytää sopivan yhteyden, se palauttaa sen yhteystehtaalle.

5. Jos sovelluspalvelin ei löydä sopivaa yhteyttä yhteysvarastosta, se käyttää resurssisovitinta luodakseen uuden fyysisen yhteyden integroituun tietojärjestelmään.
6. Resurssisovitin luo sovelluspalvelimen pyynnöstä uuden yhteyden integroituun tietojärjestelmään ja palauttaa sen sovelluspalvelimelle.
7. Sovelluspalvelin lisää yhteyden yhteysvarastoon. Osana tätä prosessia sovelluspalvelin luo sovelluspalvelintason "kahvan" hallitulle yhteydelle, jonka se palauttaa edelleen resurssisovittimelle ja sovelluskomponentille.
8. Sovelluskomponentti käyttää resurssisovittimen palauttamaan kahvaa saadakseen pääsyn integroituun tietojärjestelmään.
9. Kun sovelluskomponentti on viimeistellyt työn yhteyden kanssa, se sulkee kahvan ja yhteys jää vapaaksi yhteysvarastoon.

Yhteydenhallintasopimuksen avulla resurssisovittimen on mahdollista hyödyntää tehokkaasti sovelluspalvelimen tarjoamia palveluita. Tästä on sovelluskomponentin näkökulmasta useita hyötyjä. Yhteydenhallintasopimus tekee yhteyksien varastoinnista täysin läpinäkyviä sovelluskomponentille; sovelluskomponentin ei tarvitse välittää yhteydenvarastoinnin toteutuksesta eikä tarvitse tietää, kuinka sovelluspalvelin on toteuttanut yhteysvarastoinnin ja hallinnoi sitä. Tämä yksinkertaistaa sovelluksien ohjelmointimallia yhteyksien hallinnan osalta. Yhteydenhallintasopimus mahdollistaa sovelluspalvelimen tarjoamien laadullisten palveluiden käytön, kuten tapahtumien ja tietoturvan hallinnointi, virheiden lokitietojen kirjaaminen ja virhetilanteiden jäljitys.

Yhteyden saamisesta sovelluskomponentille on esitetty lähdekooditason esimerkki liitteessä 1.

5.5 Tapahtumien hallinta

JCA tukee sekä paikallista tapahtumanhallintaa että globaalia tapahtumanhallintaa, josta käytetään myös nimityksiä JTA- ja XA-tapahtumanhallinta. Globaalissa mallissa tapahtumia hallinnoi resurssihallitsimen näkökulmasta ulkoinen tapahtumanhallitsin. Ulkoinen tapahtumanhallitsin voi olla yhteydessä useisiin resurssihallitsimiin. JTA on Java-toteutus teollisuusstandardille X/Open XA -protokollalle, joka tukee hajautettuja tapahtumia ja kaksivaiheista vahvistamista. Paikallisella tapahtumanhallinnalla tarkoitetaan resurssihallitsimen (esimerkiksi tietokannanhallintajärjestelmä) sisäisesti hallinnoimia tapahtumia. Tällaisten tapahtumien koordinointiin ei tarvita ulkoisia tapahtumanhallitsinta. (Sharma ym 2001, 53-54)

Resurssisovittimelle on määritelty kolme eri tasoa, joiden mukaan se tukee tapahtumanhallintaa:

- Resurssisovitin ei tue paikallista tapahtumanhallintaa eikä globaalia tapahtumanhallintaa
- Resurssisovitin tukee ainoastaan paikallista tapahtuman hallintaa
- Resurssisovitin tukee paikallista sekä globaalia tapahtuman hallintaa

Tapahtumienhallintasopimus laajentaa yhteydenhallintasopimusta lisäämällä siihen tuen paikalliselle ja globaalille tapahtumanhallinnalle. Tapahtumanhallintasopimus on sovelluspalvelimen ja resurssisovittimen (ja sen taustajärjestelmän) välinen sopimus tapahtumien hallinnasta (KUVIO 12). Tapahtumanhallintasopimus mahdollistaa resurssisovittimen ja sovelluskomponenttien hyödyntää sovelluspalvelimen infrastruktuuria ja tapahtumien hallinnan ajonaikaista tukea. (Sharma ym 2001, 53-54)

Tapahtumanhallinnan tarkoituksena on varmistaa datan eheys. Käytännössä tämä on toteutettu tiukoilla säännöillä, jotka rajoittavat sovelluksien mahdollisuuksia

päästä käsiksi dataan. J2EE-alustan tapahtumanhallinnan tuki helpottaa sovelluskehittäjien työtä tekemällä lähes läpinäkyväksi monet kompleksiset tilanteet, kuten virhetilanteista toipuminen ja yhtäaikaisten käyttäjien huomioiminen ohjelmoinnissa. Vaikka J2EE-alusta pelkistää tapahtumanhallinnallisten sovelluksien toteutusta, se ei kuitenkaan rajoita tapahtumia yhteen tietokantaan tai yhteen sovelluspalvelimeen. Globaalit hajautetut tapahtumat voivat samanaikaisesti päivittää dataa useissa eri integroiduissa tietojärjestelmissä ja eri sovelluspalvelininstansseissa.

Tapahtumanhallinnassa on hyvin keskeistä tapahtuman vahvistaminen (commit) ja siihen liittyvät kaksi eri protokollaa: yksivaiheinen vahvistaminen (one-phase commit) ja kaksivaiheinen vahvistaminen (two-phase commit). Yhden resurssihallitsimen hallinnoimissa tapahtumissa käytetään yksivaiheista vahvistamista. Kun yhteen tapahtumaan liittyvät kaikki toimenpiteet (esimerkiksi tietokannan lukeminen, kirjoittaminen ja päivittäminen) ovat täysin valmiit, tapahtuma sitoutetaan eli tämän jälkeen tapahtuman tekemät muutokset ovat varmistettu ja tapahtuman ACID-ominaisuudet toteutuvat. (Sharma ym 2001, 56)

Kaksivaiheisen vahvistamisen tarkoituksena on varmistaa, että tapahtumaan sisältyvät toimenpiteet sitoutetaan oikein useiden resurssihallitsimien välillä. Kaksivaiheisen vahvistamisen periaatteena on päivittää resurssihallitsimia kahdessa eri vaiheessa: ensimmäinen on valmistautumisvaihe ja toinen on varsinainen vahvistamisvaihe. Ensimmäisessä vaiheessa tapahtumanhallitsin pyytää jokaista resurssihallitsinta valmistautumaan tapahtuman vahvistamiseen. Kun kaikki resurssihallitsimet ovat ilmoittaneet olevansa valmiita vahvistamiseen, tapahtumanhallitsin pyytää resurssihallitsimia vahvistamamaan tapahtuman. Jos yksikin tapahtumaan osallistuva resurssihallitsin ei ole valmistautunut vahvistamiseen (esimerkiksi virhetilanteen vuoksi), tapahtumanhallitsin käyttää toisen vaiheen ilmoittaakseen resurssihallitsimille, että peruuttavat (rollback) tapahtuman. (Sharma ym 2001, 56-57) Kun tapahtuma

peruutetaan, tietokannan tila palautetaan alkuperäiseen tilaan siltä osin kuin peruutettu tapahtuma muutti sitä.

EJB-komponentit ovat keskeisessä roolissa J2EE-sovelluksien tapahtumien hallinnassa. EJB-sovelluksien kannalta tapahtumia voi hallita kahden eri periaatteen mukaan: komponenttipohjainen tapahtumanhallinta ja säiliöpohjainen tapahtumanhallinta. Kohdepohjaiset komponentit käyttävät ainoastaan säiliöpohjaista tapahtumanhallintaa, kun istuntopohjaiset komponentit voivat käyttää kumpaa tapaa tahansa. Komponenttipohjaisessa tapahtuman hallinnassa komponentit käyttävät JTA-liittymän tarjoamia palveluita. Komponentin toteuttajan täytyy eksplisiittisesti kirjoittaa komponentin koodiin tapahtuman hallintaan liittyvät toimenpiteet. (Sharma ym. 2001, 59)

Säiliöpohjaisessa tapahtuman hallinnassa säiliö huolehtii komponenttien tapahtumien hallinnasta implisiittisesti. Komponenttien toteuttajan ei tarvitse kirjoittaa komponentin koodin tapahtumanhallintaan liittyvää koodia, mutta komponentin asentajan täytyy määritellä tapahtumankäsittelyyn liittyvät attribuutit komponentin asennuskuvauksiin. Asennuskuvauksien perusteella säiliö voi päättää osallistuuko, komponentista kutsuttava palvelu tapahtumaan ja millä tasolla. Esimerkiksi kun sovellus kutsuu komponentin metodia, säiliö voi päätellä asetuksista, että sen täytyy aloittaa uusi tapahtuma. Kun metodin suoritus on valmis, säiliö vahvistaa tapahtuman. (Sharma ym. 2001, 59)

Kun J2EE-sovelluspalvelin tukee tapahtumanhallintaa useiden integroitujen tietojärjestelmien välillä, voi sovelluskomponentti ottaa yhteyden näihin tietojärjestelmiin ja päivittää useita tietokantoja yhden JTA-tapahtuman sisällä. Esimerkissä 2 on havainnollistettu komponenttipohjaista tapahtumanhallintaa tilanteessa, jossa päivitetään kahta eri tietokantaa yhden JTA-tapahtuman sisällä. (Sharma ym. 2001, 61) Kuten esimerkistä voi huomata, komponenttipohjainen tapahtumanhallinta on melko yksinkertaista, koska J2EE-soveluspalvelin

huolehtii tapahtuman koordinoinnista ja jakamisesta palvelimen sekä integroitujen tietojärjestelmien välillä. Tapahtuman kannalta sovelluskomponenttiin ei tarvitse kirjoittaa varsinaisen datan käsittelyn lisäksi kuin tapahtuman aloittamiseen, lopettamiseen ja mahdollisten virhetilanteisiin liittyvät toimenpiteet.

```
...
InitialContext ic = new IntialContext( "java:comp/env" );
DataSource db1 =
    (DataSource)ic.lookup( "TilausTietokantaDS " )
DataSource db2 =
    (DataSource)ic.lookup( "VarastoTietokantaDS " )

Connection con1 = db1.getConnection();
Connection con2 = db2.getConnection();

UserTransaction ut = ejbContext.getUserTransaction();

//Aloitetaan tapahtuma
ut.begin();

//Suoritetaan päivitykset Tilaus-tietokantaan käyttäen yhteyttä
// con1.
//Suoritetaan päivitykset Varasto-tietokantaan käyttäen yhteyttä
// con2.

// Sitoutetaan tapahtuma
ut.commit().
```

ESIMERKKI 2. Komponenttipohjainen tapahtumanhallinta

5.6 Tietoturvan hallinta

Tietoturvallisuus on hyvin tärkeä vaatimus kaikille yrityksen sovelluksille. Tietoturvan merkitys korostuu yhä enemmän, kun sovelluksia hajautetaan ja sovelluksien toimintoja suoritetaan Internetin yli. Yrityksen sovelluksille tulisi olla selkeästi määritellyt vaatimukset ja arkkitehtuuri, kuinka tietoturvallisuus varmistetaan integroidussa ympäristössä. Koska JCA on kohdistettu J2EE-sovellusten ja yrityksen tietojärjestelmien integrointiin, se määrittää mekanismit ja standardin joukon liittymiä, jotka tukevat tietoturvallista sovellusten

integrointia. Tietoturvanhallintasopimus (secure management contract) on yksi kolmesta järjestelmätason sopimuksista (KUVIO 12), joka laajentaa J2EE:n tietoturvamallia ja yhteydenhallintasopimusta.

Tietoturvanhallintasopimus mahdollistaa tuen erilaisille tietoturvamekanismeille, jotka suojaavat integroitua tietojärjestelmää erilaisilta uhilta, kuten valtuuttamattomilta pääsyiltä järjestelmään. Tietoturvanhallintasopimus auttaa varmistamaan, että käyttäjät ovat, keitä he sanovat olevansa. Tässä yhteydessä käyttäjällä voidaan tarkoittaa myös sovelluskomponenttia tai asiakassovellusta. Käyttäjien korrektista tunnistamisesta käytetään myös termiä autentikointi (authentication). Kun käyttäjä on kerran autentikoitu, tietoturvanhallintasopimus auttaa määrittämään käyttäjän valtuuksia. Valtuuttamisella (authorization) määritellään käyttäjälle tietyt oikeudet, joiden perusteella voidaan päätellä, onko käyttäjällä oikeus päästä tiettyihin integroidun tietojärjestelmän resursseihin. Tietoturvanhallintasopimus suojaa myös kommunikointiyhteyttä sovelluspalvelimen ja integroitavan tietojärjestelmän välillä tukemalla tietoturvallisia yhteyskäytäntöjä, kuten SSL-protokollaa (Secure Socket Layer). (Sharma ym 2001, 69-70)

Sovelluskomponentin toimittajalla on kaksi vaihtoehtoa suunnitella, kuinka kirjautuminen suoritetaan integroitavaan tietojärjestelmään: säiliöpohjainen kirjautuminen (container-managed sign-on) ja komponenttipohjainen kirjautuminen (component-managed sign-on). Säiliöpohjaisessa kirjautumisessa sovelluskomponentti jättää säiliön vastuulle integroituun tietojärjestelmään kirjautumisen asetukset ja hallinnoinnin. Säiliö päättää myös, millä käyttäjätunnuksella ja salasanalla yhteys perustetaan. Komponenttipohjaisessa kirjautumisessa komponentin koodiin kirjoitetaan kirjautumiseen liittyvät toimenpiteet. (Sharma ym. 2001, 75)

5.7 Yleinen asiakasliittymä

Common Client Interface (CCI) on JCA:n yleinen asiakastason ohjelmointiliittymä, jonka avulla asiakasohjelmat voivat ottaa yhteyden integroituun tietojärjestelmän ja päästä käsiksi sen resursseihin. CCI on matalatasoinen ohjelmointiliittymä, aivan kuten JDBC. JDBC:n avulla sovellukset pääsevät käsiksi relaatiotietokantoihin ja vastaavasti CCI:n avulla sovellukset pääsevät käsiksi ei-relaationaalisiin järjestelmiin. CCI:lla voidaan hallita tietovirtoja asiakassovelluksen ja taustajärjestelmän välillä välittämättä säiliön ja sovelluspalvelimen toiminnoista. CCI on suunniteltu muutamia tavoitteita noudattaen. CCI:stä on pyritty tekemään mahdollisimman geneerinen, jotta se soveltuisi mahdollisimman moneen erityyppiseen tietojärjestelmään. Vaikka CCI:stä on pyritty tekemään helppokäyttöinen, on se silti laajennettava. CCI on tarkoitettu perustason ohjelmointiliittymäksi, jonka päälle voidaan kehittää tietojärjestelmäspesifisiä toimintoja yhteyden hallinnan lisäksi. (Modi 2001)

Resurssisovittimeen liittyy kahdenlaisia sopimuksia: järjestelmätason ja sovellustason sopimukset. CCI on sovellustason sopimus resurssisovittimen ja sovelluskomponentin välissä (katso kuvio 12). Järjestelmätason sopimukset ovat riippumattomia sovellustason sopimuksista, joten resurssisovittimeen voidaan toteuttaa tietojärjestelmäspesifinen asiakasliittymä yleisen CCI-liittymän sijasta. CCI määrittää joukon liittymiä ja luokkia, joiden metodit mahdollistavat asiakassovelluksen luoda tyypillisen yhteyden tietojärjestelmään, suorittaa tietojärjestelmän etäkutsuja ja operoida dataa. (Sharma ym. 2001, 111-112) CCI:n luokat ja liittymät on jaettu seuraaviin kategorioihin: yhteys, interaktio, data ja metadata. CCI-liittymän toteuttamista ja käyttöä on esitelty seuraavassa luvussa sekä liitteessä 1.

5.8 Yhteenveto J2EE Connector Architecturesta

Tässä luvussa on käsitelty JCA-määrittelyn sisältöä sekä perehdytty resurssisovittimen toimintaan sovelluspalvelimella. JCA-määrittely helpottaa yrityksen sovellusten integroinnin ongelmaa tapauksissa, joissa toisena osapuolena on jokin J2EE-sovellus. JCA-määrittely määrittelee standardin arkkitehtuurin resurssisovittimille, joka mahdollistaa resurssisovittimen kytkemisen mille tahansa JCA-määrittelyä tukevalle J2EE-sovelluspalvelimelle. Tietojärjestelmien toimittajien ei tarvitse enää rakentaa sovelluksiinsa useita sovelluspalvelinkohtaisia sovittimia, vain yksi standardin mukainen sovitin riittää. Vastaavasti sovelluspalvelimien toimittajien tarvitsee tukea vain yhtä sovitinmäärittelyä.

JCA-määrittely jakaa integroitiin liittyvän sovelluskehityksen roolit kahdeksaan eri osaan. Resurssisovittimen toimittaja on integroitavan järjestelmän asiantuntija ja tuottaa sovittimen integroitavaan tietojärjestelmään Sovelluspalvelimen toimittaja toimittaa JCA-määrittelyä tukevan J2EE-sovelluspalvelimen. Sovelluskomponentin toimittajaa kehittää J2EE-sovelluksia, jotka käyttävät sovittimia kommunikointiin integroitavien järjestelmien kanssa. Työkalujen toimittajat toimittavat esimerkiksi tiedonlouhintatyökaluja, jotka helpottavat datan rakenteen ja laajuuden tutkimista. Kokooja yhdistää eri sovelluskomponentit ja sovittimet sovelluspalvelimelle asennettavaksi kokonaisuudeksi sekä kirjoittaa sovelluspalvelinkohtaiset asennuksen kuvaustiedostot. Järjestelmän pääkäyttäjä on vastuussa järjestelmän asetuksista, hallinnoinnista ja koko infrastruktuurista, johon kuuluu useita sovellussäiliöitä ja eri järjestelmiä.

JCA-määrittelyssä on määritelty erilaisia sopimuksia, joiden avulla resurssisovitin ja muut komponentit voivat kommunikoida eri osapuolien kanssa. Sovellustason sopimukset määrittävät, kuinka sovelluskomponentit

kommunikoivat resurssisovittimen kanssa. Järjestelmätason sopimukset määrittävät kuinka resurssisovitin ja sovelluspalvelin kommunikoivat keskenään. Järjestelmätason sopimukset mahdollistavat J2EE-sovelluspalvelimen tarjoamien palveluiden hyödyntämisen, kuten yhteyksien hallinnoinnin ja varastoinnin, tapahtumien hallinnoinnin eri tasoilla sekä tietoturvanhallinnan.

6. RESURSSISOVITTIMEN RAKENTAMINEN

Resurssisovitin on järjestelmätason ohjelmistokirjasto, jota sovellukset tai sovelluskomponentit käyttävät ottaessaan yhteyden integroituun tietojärjestelmään. Resurssisovittimen tyypillisiä toimittajia ovat tietojärjestelmien, väliohjelmistojen ja sovelluspalvelimien toimittajat. Tässä luvussa tarkastellaan JCA-arkkitehtuurin mukaisen resurssisovittimen rakentamista sekä konstruoidaan sovellusten integroinnista esimerkki, joka sisältää kaksi esimerkkisovellusta ja resurssisovittimen.

Resurssisovitin koostuu Java-luokista, jotka toteuttavat järjestelmätason sopimukset ja asiakastason API:n, joka voi olla standardi yleinen asiakasliittymä (CCI) tai jokin järjestelmäspesifinen liittymä. Lisäksi sovittimen asennusta varten pitää kirjoittaa XML-pohjaiset (Extensible Markup Language) asennuksen kuvaustiedostot. Sovittimen toteuttaminen voidaan jakaa kolmeen osaan: järjestelmätason sopimukset, asiakastason API ja kommunikointi integroitavaan tietojärjestelmään. Resurssisovitin kommunikoi integroitavan tietojärjestelmän kanssa tietojärjestelmäspesifisen protokollan avulla. Resurssisovitin voi toteuttaa tuen kommunikoinnille kokonaan tai käyttää integroitavan tietojärjestelmään liittyviä matalatasoisia kommunikointikirjastoja. JCA-määrittely ei ota kantaa, kuinka resurssisovittimen ja integroitavan tietojärjestelmän välinen kommunikointi toteutetaan, koska toteutus on hyvin järjestelmäriippuvainen.

6.1 Resurssisovittimen liittymät

Resurssisovittimen tulee toteuttaa tietty joukko liittymiä, jotta se olisi JCA-määrittelyn mukainen. Taulukossa 3 on esitelty järjestelmätason liittymät, liittymän pakollisuus ja lyhyt kuvaus. Näiden liittymien lisäksi resurssisovittimen on toteutettava luokat yhteydelle (Connection) ja yhteystehtaalle

(ConnectionFactory), jotka ovat esitelty esimerkiksi CCI-paketissa (taulukko 2). Jos resurssisovitin ei toteuta CCI:n mukaista asiakastason sovellusliittymää, voidaan resurssisovittimen yhteystehdas- ja yhteysluokan toteuttaa mistä tahansa Java-liittymästä.

TAULUKKO 2. Yleisen asiakasliittymän liittymät

Liittymä	**Kuvaus**
javax.resource.cci	Paketin nimi
Connection	*Connection*-liittumän toteutus sisältää "kahvan" fyysiseen yhteyteen integroituun järjestelmään.
Connectionfactory	*Connectionfactory*-liittymän toteutuksen avulla saadaan yhteys yhteysvarastosta.
ConnectionMetaData	*ConnectionMetaData*-liittymän toteutus sisältää tietoa yhteydestä ja integroidusta järjestelmästä.
ConnectionSpec	*ConnectionSpec*-liittymän toteutus sisältää yhteyden luontiin liittyviä asetuksia.
IndexedRecord	*IndexedRecord*-liittymän toteutus on *java.util.List*-luokkaan perustuva järjestetty lista *Record*-olioista.
Interaction	*Interaction*-liittymän toteutuksen avulla suoritetaan integroidun järjestelmän toimintoja (esim. suoritetaan kyselyitä).
InteractionSpec	*InteractionSpec*-liittymän toteutus sisältää edellä mainittujen toimintojen suorituksiin liittyviä asetuksia.
LocalTransaction	*LocalTransaction*-liittymän toteutus mahdollistaa paikallisen tapahtumanhallinnan käytön.
MappedRecord	*MappedRecord*-liittymän toteutusta käytetään *Record*-olioiden tallentamiseen avain-arvo periaatteen mukaisesti.
Record	*Record*-liittymän toteutusta käytetään kapseloiman dataa, jota välitetään ja palautetaan *Interaction*-toteutuksen metodien suorituksissa.
RecordFactory	*RecordFactory*-liittymän toteutuksen avulla luodaan *MappedRecord*- ja *IndexedRecord*-oliot.
ResourceAdapter-MetaData	*ResourceAdapterMetaData*-liittymän toteutus sisältää sovittimen ja sen ominaisuuksiin liittyvää tietoa.
ResultSet	*ResultSet*-liittymän toteutus sisältää taulukkomuotoista dataa, joka voidaan saada paluuarvona suoritetuista integroidun tietojärjestelmän toiminnoista. Pohjautuu JDBC-liittymän ResultSet-toteutukseen.
ResultSetInfo	*ResultSetInfo*-liittymän toteutus sisältää *ResultSet*-toteutukseen liittyvää tietoa.
Streamable	*Streamable*-liittymän toteutus mahdollistaa sovittimen käsitellä saapuvien ja lähtevien *Record*-luokkien datan tavumuotoisena tietovirtana.

Edellä mainittu *ConnectionFactory* on liittymä, jonka avulla sovelluskomponentti voi pyytää yhteyden (*Connection*) integroituun tietojärjestelmään. Sovellus pyytää yhteyden *ConnectionFactory.getConnection*-metodin avulla, joka kutsuu edelleen sovelluspalvelimen *ConnectionManager.allocateConnection*-metodia. Tämän jälkeen resurssisovitin siirtää suorituksen yhteyden luomisesta sovelluspalvelimelle, koska sovelluspalvelin on vastuussa yhteyksien varastoinnista ja muista palveluista. *Connection*-luokka sisältää sovellustason "kahvan", jonka avulla asiakaskomponentti saa fyysisen yhteyden integroitavaan järjestelmään. Varsinainen fyysinen yhteys, johon *Connection*-luokka on assosioitu, on esitelty *ManagedConnection*-luokassa.

Tapahtumanhallintaan liittyvät liittymät (taulukko 3) muodostavat osan sovelluskehyksestä, jota sovelluspalvelin käyttää tapahtumien hallinnointiin ja suorittamiseen yrityksen sovellusten välillä. Tapahtumanhallinnan liittymät tukevat tapahtuman hallintaa jokaisella eri tasolla, joten on resurssisovittimen toteuttajan päätettävissä, millä tasolla resurssisovitin todellisuudessa tukee tapahtuman hallintaa. Resurssisovitin voi tukea paikallisia tapahtumia, paikallisia sekä globaaleja tapahtumia tai olla tukematta tapahtumia ollenkaan. Riippumatta tasosta, millä tapahtumanhallintaa tuetaan, resurssisovittimen *ManagedConnection*-luokkaan tulee toteuttaa *getXAResource*- ja *getLocalTransaction*-metodit. Mikäli tapahtumanhallintaa ei haluta toteuttaa, tämä tulee huomioida tapahtumien hallintaa liittyvien metodien poikkeusten hallinnassa.

Tietoturvasopimus takaa turvallisen yhteyden sovelluskomponentin ja integroitavan tietojärjestelmän välille. Tietoturvasopimus yhdistää J2EE Java Authentication ja Authorization Service (JAAS) -määrittelyiden ominaisuuksia yhteydenhallinnan liittymiin. Resurssisovitin käyttää JAAS:n *Subject*-luokkaa ja

Pricipal-liittymää. *Subject*-luokka ryhmittelee tietoja, jotka liittyvä yhteen kokonaisuuteen. *Subject*-luokka sisältää identiteettejä, joita kuvaa *Principal*-liittymän toteutukset. Identiteeteillä on eri valtuuksia, joita kuvaa *GenericCredential*-liittymän toteutukset. Tietoturvasopimuksessa on lisäksi määritelty *PasswordCredential*-luokka, joka kapseloi käyttäjätunnuksen, salasanan ja *ManagedConnectorFactory*:n ilmentymän.

Tietoturvasopimuksen määritykset tulee huomioida kolmen eri yhteydenhallinnan liittymän toteutuksessa, jotka ovat: *ConnectionFactory*, *ManagedConnectionFactory* ja *ManagedConnection*. Sovelluspalvelimen tietoturvapalveluita käytetään, kun *ConnectionFactory*-instanssi pyytää sovelluspalvelimelta yhteyttä *ConnectionManager.allocateConnection*-metodin avulla. Riippuen siitä, onko käytössä säiliö- vai komponenttipohjainen tietojärjestelmään kirjautuminen, resurssisovitin kutsuu *allocateConnection*-metodia eri tavalla. Jos käytössä on säiliöpohjainen kirjautuminen, sovelluskomponentti ei välitä mitään tietoturvainformaatiota resurssisovittimelle yhteyspyynnön parametrina. Jos sovelluskomponentti on vastuussa tietojärjestelmään kirjautumisesta, se välittää tarvittavan tietoturvainformaation resurssisovittimelle.

TAULUKKO 3. Järjestelmätason liittymät

Liittymä	**Pakol-lisuus**	**Kuvaus**
Paketti: javax.resource.spi		
ConnectionManager	Ei	ConnectionManager-liittymän toteutaminen mahdollistaa yhteyksienhallinnan ympäristössä, jossa resurssisovitinta käyttävää asiakassovellusta ei suoriteta sovelluspalvelimella. Sovelluspalvelimella suoritettavat sovellukset voivat käyttää sovelluspalvelimen toteuttamaa ConnectionManageria.

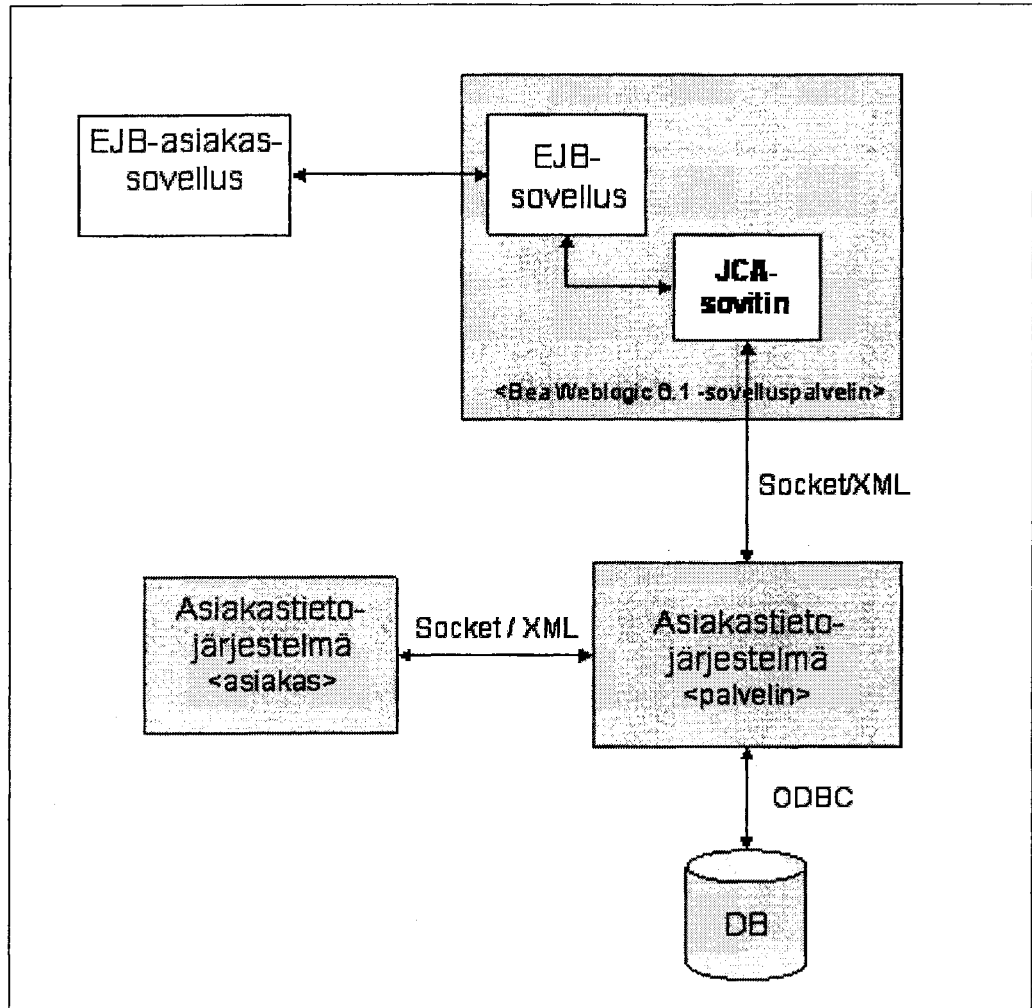
(jatkuu)

TAULUKKO 3. (jatkuu)

ConnectionRequest-Info	Ei	ConnectionRequestInfo-liittymän toteutus sisältää tietoturvaan ja asiakassovellukseen liittyvää informaatiota.
LocalTransaction	Ei	LocalTransaction.liittymän toteutus mahdollistaa paikallisten tapahtumien hallinnan.
ManagedConnection	Kyllä	ManagedConnection-liittymä kuvaa fyysistä yhteyttä integroituun tietojärjestelmään.
Managed-ConnectionFactory	Kyllä	ManagedConnectionFactory-liittymä kuvaa yhteystehdasta hallituille yhteyksille (ManagedConnection).
ManagedConnection-MetaData	Kyllä	ManagedConnectionMetaData-liittymän toteutus sitältää hallittuihin yhteyksiin liittyvää informaatiota.
Paketti: javax.transaction.xa.		
XAResource	Ei	XAResource-liittymän toteutus sisältää metodeja hajautettujen XA-tapahtumien hallintaa. Sisältää myös tuen kaksivaiheiselle vahvistamiselle.

6.2 Konstruktion ympäristö ja toiminnallisuus

Tässä kappaleessa kuvataan konstruktion osapuolet, tapahtumien kulku sekä ympäristö. Konstruktion arkkitehtuuria on selvennetty kuviossa 14. Integroitavan sovelluksen osaa esittää asiakastietojärjestelmä, joka koostuu palvelinohjelmistosta ja asiakasohjelmistosta. Integroinnin toista osapuolta, J2EE-sovellusta, esittää yksinkertainen tilaton istuntopohjainen EJB-komponentti ja EJB-asiakassovellus. Sovelluksien "välissä" on JCA-määrittelyn mukainen sovitin, joka mahdollistaa J2EE-sovelluksen ja asiakastietojärjestelmän välisen kommunikoinnin. Sovelluspalvelimena on esimerkissä käytetty Bea Weblogic 6.1 -palvelinta, joka on J2EE 1.3 -määrittelyn mukainen sovelluspalvelin. Weblogic on IBM Webspheren ohella yksi suosituimmista sovelluspalvelimista.



KUVIO 14. Konstruktion eri osapuolet

Tässä esimerkissä kuvattu asiakastietojärjestelmä ei ole todellisessa käytössä oleva yrityksen asiakastietojärjestelmä, mutta sitä simuloi konstruktiossa käytetty, täysin toimiva, tietokantaa käyttävä palvelin-asiakasohjelmisto. Asiakastietojärjestelmällä voidaan hakea asiakastietoja tietokannasta. Asiakastietojärjestelmän kommunikointi perustuu Javan verkko-ominaisuuksiin ja tarkemmin ServerSocket/Socket-ominaisuuksiin. Javan verkkotoiminta perustuu TCP/IP (Transmission Control Protocol / Internet Protocol) -protokolliin, jota myös Internetissä käytetään yleisimmin. Asiakastietojärjestelmän palvelin (ServerSocket) kuuntelee asiakkaiden (Socket) yhteydenottoja ja vastaanottaa tietovirtana lähetettyjä sanomia. Palvelinohjelma vastaanottaa asiakkaalta ennalta määrätyn muotoisia XML-sanomia, jäsentää

sanomien sisällön, suorittaa sanomien mukaiset tietokantakyselyt ja lopuksi palauttaa kyselyn tuloksen edelleen asiakkaalle. Asiakasohjelma jäsentää palvelimen lähettämän kyselyn tuloksen XML-kirjastojen avulla ja tulostaa hakutuloksen käyttöliittymään. Palvelinohjelman ja tietokannan välinen kommunikointi on toteutettu ODBC-liittymän kautta.

Integroinnin toisena osapuolena on J2EE-sovellus, joka sisältää yhden EJB-komponentin ja asiakassovelluksen. Tässä konstruktiossa asiakassovellus sisältää vain yhden toiminnon, jolla haetaan asiakkaan tai asiakkaiden henkilötietoja annetulla hakuehdolla. Asiakassovellus on niin sanottu ohut asiakas (thin client), joka sisältää vain käyttöliittymän toiminnallisuuden. Varsinainen liiketoimintalogiikka, tässä tapauksessa asiakastietojen haku, on EJB-komponentin vastuulla.

EJB-komponentin asiakkaalle näkyvässä etärajapinnassa on asiakastietojen haulle yksi metodi, jonka palvelinpuolen luokat toteuttavat. Se, miten palvelu on toteutettu ja mistä data haetaan, on asiakassovellukselle täysin läpinäkyvää. EJB-komponentti käyttää asiakastietojärjestelmää tietovarastona CCI-liittymän kautta, lähes samalla tavalla kuin mikä tahansa muu sovellus, joka käyttää tietokantaa esimerkiksi ODBC- tai JDBC-liittymän avulla. EJB-komponentti vastaanottaa asiakassovelluksen hakupyynnön ja sen parametrina haun ehdot. Tämän jälkeen EJB-komponentti suorittaa haun annetuilla ehdoilla resurssisovittimen CCI-liittymää käyttäen ja palauttaa hakutuloksen asiakkaalle.

6.3 Resurssisovittimen konstruointi

Resurssisovittimen tärkeimpinä tehtävinä on luoda ja hallita yhteyksiä integroitavaan järjestelmään sekä tarjota liittymä, jonka avulla voidaan suorittaa integroidun järjestelmän palveluita. Tässä konstruktiossa toteutetaan kappaleessa 6.1 kuvatut järjestelmätason liittymät, lukuun ottamatta XAResource- ja

LocalTransaction-liittymiä, koska asiakastietojärjestelmä ei tue tapahtumanhallintaa millään tasolla. Resurssisovittimen asiakasliittymäksi on valittu JCA-määrittelyn sisältämä yleinen asiakasliittymä (CCI). Resurssisovittimen ja EJB-komponentin toteutukset ovat kokonaisuudessaan liitteessä 1, joten niitä ei käydä tässä yksityiskohtaisesti läpi.

Resurssisovittimen luokat voidaan jakaa kolmeen osaan: yhteyden hallintaan käytettävät sovelluspalvelimen tarvitsemat luokat, fyysiseen yhteyteen liittyvät luokat ja datan käsittelyyn liittyvät asiakasliittymän luokat. Yhteyden hallintaan liittyvistä luokista keskeisin on *ManagedConnectionFactory*-liittymän toteutus, jonka *createManagedConnection*-metodissa luodaan varsinainen yhteys integroitavaan järjestelmään. Samassa luokassa suoritetaan myös autentikointi. Tässä resurssisovittimessa käytetään säiliön hallinnoimaa käyttäjäntunnistusta. Autentikointikäytäntö, käyttäjätunnus ja salasana on määritelty resurssisovittimen asennustiedostoissa (weblogic-ra.xml ja ra.xml). Kun säiliö luo yhteyksiä, autentikointiin liittyvät tiedot välitetään *ManagedConnectionFactory*-oliolle, joka tarvittaessa välittää tiedot edelleen integroitavalle järjestelmälle.

Asiakasliittymäksi on tähän resurssisovittimeen valittu Common Client Interface (CCI) -liittymän mukainen toteutus. CCI-liitymän avulla voidaan hakea tietoa kahdessa eri muodossa: listamuotoista dataa *IndexedRecord*-liittymän toteutuksen avulla tai avain/arvo muotoista dataa *MappedRecord*-liittymän toteutuksen avulla. Tässä asiakasliittymässä tuetaan vain *MappedRecord*-liittymän mukaista tiedonvälitystä. Asiakasliittymän keskeisin luokka on *Interaction*-liittymän toteutus, joka sisältään Asiakastietojärjestelmän mukaisen toteutuksen tiedon hakemiseksi. CCI-liittymän toteutus on esitelty liitteessä 1 ja sen käyttöä on demonstroitu *GraduPapuBean*-komponentissa, joka on myös liitteessä 1.

6.4 Yhteenveto resurssisovittimen rakentamisesta

Tässä luvussa esiteltiin resurssisovittimen ja sen asiakasliittymän toteuttamiseen tarvittavia liittymiä, toteutukseen liittyviä yksityiskohtia sekä konstruktion sovellusympäristö. Resurssisovittimen tulee toteuttaa useita liittymiä, joista järjestelmätason yhteydenhallintaan liittyvien liittymien toteuttaminen on pakollista. Tapahtumien hallinnan toteuttamien riippuu integroitavasta järjestelmästä. Tässä esimerkissä integroitava järjestelmä ei tue tapahtumien hallintaa, joten sitä ei ole toteutettu. Resurssisovittimen asiakasliittymä voi olla JCA-määrittelyn sisältämän Common Client Interface (CCI) -liittymän mukainen tai sovelluskohtainen. Tässä esimerkissä on asiakasliittymäksi valittu CCI-liittymän toteutus.

Resurssisovitin hyödyntää sovelluspalvelimen tarjoamia palveluita yhteyden muodostamisessa ja hallinnassa, tapahtumien hallinnassa sekä tietoturvan hallinnassa. Tämä helpottaa huomattavasti resurssisovittimen kehittämistä. Valmiiden palveluiden hyödyntämisestä huolimatta, jopa yksikertaisen ja vähän toiminnallisuutta sisältävän resurssisovittimen kehittäminen on melko työlästä. Resurssisovittimeen tulee toteuttaa useiden liittymien lisäksi integroitavan järjestelmän kommunikointikäytännöstä riippuvainen osuus, sekä liittymään sopiva yleinen asiakasliittymä.

7. POHDINTA

Tässä luvussa pohditaan JCA-määrittelyä ja sen sijoittumista toisessa ja kolmannessa luvussa esitettyihin integroinnin tasoihin ja malleihin, JCA-määrittelyn vahvuuksia ja heikkouksia sekä sen soveltuvuutta yrityksen sovellusten integrointiin.

7.1 Miksi JCA?

Yrityksien tietojärjestelmät käsittelevät yrityksen sisäistä informaatiota useilla ohjelmistoilla ja eri muodoissa. Informaatio voi sijaita esimerkiksi ERP-järjestelmissä, relaatiotietokannoissa tai keskustietokonejärjestelmissä. Yhä useampi yritys on siirtänyt liiketoimintaansa Internet-pohjaiseksi, jonka tueksi tarvitaan uusia Internet-pohjaisia sovelluksia. Usein ei ole kuitenkaan mahdollista tai kannattavaa korvata olemassa olevia sovelluksia uusilla sovelluksilla. Tämän seurauksena sovellusten integrointi on tullut välttämättömäksi. Kuten luvussa 4 on esitetty, J2EE-alusta on yksi suosituimmista ja monipuolisimmista alustoista Internet-pohjaisille sovelluksille, ja nykyään yhä useampi yrityksen Internet-pohjainen sovellus tehdään J2EE-alustalle. Jotta J2EE-sovellukset vastaisivat mahdollisimman hyvin yrityksen liiketoiminnallisia vaatimuksia, tulee J2EE-sovellusten pystyä kommunikoimaan myös olemassa olevien sovellusten ja tietovarastojen kanssa.

J2EE-alustan sisältämät Java Database Connectivity (JDBC) ja Java Messaging Service (JMS) ovat hyvin tunnettuja ja laajasti käytettyjä sovellusliittymiä, jotka soveltuvat yrityksen sovellusten integrointiin, mutta data ei sijaitse aina relaatiotietokannoissa ja kaikki yrityksen sovellukset eivät tue sanomajonoja. Tällaisessa tapauksessa tarvitaan sovitin, jonka avulla saadaan pääsy integroitavaan järjestelmään. Ohjelmoinnin kannalta olisi myös ideaalista, että

edellä mainittu pääsy eri järjestelmiin ja datan käsittely onnistuisi yhden yleisen käytännön mukaisesti. Tätä varten JCA-määrittely sisältää yleisen asiakasrajapinnan (Common Client Interface).

Useat yrityksien sovellusten toimittajat ovat kehittäneet sovelluskohtaisia sovittimia J2EE-sovelluspalvelimien ja sovelluksiensa välille. Näiden sovittimien ongelmina ovat olleet standardoimaton arkkitehtuuri ja toimittajakohtaiset ratkaisut, jotka ovat aiheuttaneet epäyhteensopivuutta eri J2EE-sovelluspalvelimilla. Näiden ongelmien helpottamiseksi on kehitetty standardi JCA-määritelmä.

7.2 JCA:n sijoittuminen integrointiarkkitehtuureihin ja -malleihin

JCA-arkkitehtuurin voidaan ajatella olevan kappaleessa 3.1 kuvattujen pisteestä pisteeseen ja monesta moneen -arkkitehtuurien välimuoto. JCA-arkkitehtuuri ei ole puhtaasti pisteestä pisteeseen -mallin mukaista. JCA-arkkitehtuurin mukaan voi useampi eri sovellus käyttää samaa resurssisovitinta tai jopa samaa fyysistä yhteyttä integroituun järjestelmään. JCA-arkkitehtuurin mukaisen resurssisovittimen ja J2EE-sovelluspalvelimen voidaan myös ajatella toimivan eräänlaisena väliohjelmistona, jonka kautta *useat* eri Java- tai J2EE-sovellukset saavat yhteyden integroitavaan järjestelmään. Näin ajateltuna JCA-arkkitehtuuri on lähellä monesta moneen -arkkitehtuuria. Kuitenkin yhdellä toteutetulla JCA-resurssisovittimella päästää yhteyteen vain yhteen tiettyyn järjestelmään, joten JCA-arkkitehtuuri on ennemminkin monesta yhteen -arkkituurin mukaista. Tässä tutkielmassa käytetyt lähteet eivät tunne monesta yhteen -arkkitehtuurin määritelmää, joten sitä ei ole esitelty aikaisemmin.

Integrointitasoltaan JCA-arkkitehtuuri vastaa kappaleessa 2.2 Wanglerin ja Paheerathanin (2000) esittämässä jaottelussa selkeästi kohtaa 4: ”Sovellusten integrointi sovittimilla”. Linthicum (2000) ja muiden tutkijoiden nelijaottelussa

(data-, sovellusliittymä-, metodi- ja käyttöliittymätaso) JCA-määrittely vastaa parhaiten datatason integrointia. Datatason integroinnissa on tarkoituksena jakaa dataa integroitavien sovellusten kesken. JCA-resurssisovittimen avulla saadaan jaettua integroidun järjestelmän dataa muille järjestelmille. JCA-määrittely ei kuitenkaan ota kantaa resurssisovittimen järjestelmäspesifisen osuuden toteuttamiseen, joten on myös mahdollista, että JCA-määrittelyyn perustuva sovellusintegrointi voisi olla *sovellusliittymätason* mukaista. JCA-resurssisovitin voi sisäisesti käyttää integroitavan järjestelmän (natiiveja) sovellusliittymiä ja välittää resurssisovittimelle tulevat pyynnöt integroitavalle järjestelmälle niiden avulla. JCA-resurssisovittimen ja sen asiakasrajapinnan ei tarvitse välttämättä olla ODBC:n tai JBDC:n kaltainen liittymä ei-relaationaalisiin tietokantoihin tai perinnejärjestelmiin, vaikka JCA-määrittely ja sen CCI-liittymä on tällaiseen ensisijaisesti suunniteltu.

Sovittimiin perustuvassa yrityksen sovellusten integroinnissa ei ole itsessään mitään uutta, mutta vihdoin sellaisen luomiseen on olemassa standardi arkkitehtuuri, joka täydentää sovittimen osalta Javan "kirjoita kerran, aja kaikkialla" -periaatetta.

7.3 JCA:n tavoitteet

JCA-määrittely mahdollistaa sovittimen toteuttamisen, joka vastaa Beveridgen (2000) asettamia vaatimuksia perinnejärjestelmiin kehitettäville sovittimille. JCA-resurssisovittimesta saa hyvin suorituskykyisen J2EE-palvelimen mahdollistamien hajautuksen ja yhteysaltaiden käytön ansiosta. Yhteysaltaan ansiosta kerran luotua yhteyttä voidaan uudelleen käyttää: yhteyttä ei tarvitse sulkea ja luoda aina uudestaan, vaan yhteyksiä voidaan pitää varastossa ja jakaa useiden asiakassovelluksien kesken. JCA-määrittely tukee sovelluspalvelimien tarjoamia tapahtumanhallinnallisia ominaisuuksia paikallisella ja globaalilla

tasolla, joka mahdollistaa datan eheyden säilymisen ja poikkeustilanteista toipumisen.

Kappaleessa 5.1 on esitelty JCA-määrittelylle asetettuja tavoitteitta. Seuraavaksi pohditaan näiden tavoitteiden täyttymistä:

1. J2EE-alusta tarjoaa skaalautuvan sovellusympäristön ja palvelut tapahtumien ja tietoturvan hallintaan, jotka yksinkertaistavat osaltaan resurssisovittimen kehitystä. Resurssisovittimen sovelluskohtaisten osuuksien toteutuksiin määrittely ei ota kantaa, joten monimutkaisiin järjestelmiin sovittimen kehittäminen on edelleen monimutkaista.
2. JCA-määrittely on hyvin suuntaa antava, vaikka tietyt liittymät pitääkin toteuttaa, toteutustapa on aina sovelluskohtainen. Myöskään sovittimen asiakasliittymän ei tarvitse olla CCI:n mukainen, mikä mahdollistaa sovittimen kehittämisen hyvin erilaisiin järjestelmiin.
3. JCA-määrittelyn mukainen sovitin ei ole sidottu vain yhteen sovelluspalvelimeen. Riittää, että sovelluspalvelimen toimittaja toteuttaa sovelluspalvelimeen tuen yhdelle standardille sovitinarkkitehtuurille. Tämän jälkeen sovelluspalvelimessa voidaan käyttää kaikkia standardin mukaisia sovittimia ilman erillistä järjestelmäkohtaista tukea. Tällä hetkellä markkinoilta löytyy ainakin kymmenen J2EE-sovelluspalvelinta, jotka tukevat JCA 1.0 -määrittelyä, joihin lukeutuu muun muassa Sunin, Bean, IBM:n, Borlandin, Oraclen ja Sybasen sovelluspalvelimet.
4. JCA-määrittely sisältää standardin asiakasliittymän, joka on yhteinen eri resurssisovittimille. Tämä helpottaa sovittimen käyttöä, koska ohjelmoijien ei tarvitse opetella useiden eri liittymien käytäntöjä. Sovittimen ei ole kuitenkaan pakko sisältää standardia CCI-liittymää, sen sijasta voidaan myös toteuttaa sovelluskohtainen asiakasliittymä.
5. JCA-määrittelyn mukaista resurssisovitinta on helppo käyttää CCI-liittymän avulla, ja esimerkiksi kokeneelle tietokantaohjelmoijalle se ei esitä uusia konsepteja. JCA:n helppoutta on vaikea määritellä, mutta

varsinkin resurssisovittimen toteuttaminen tarvitsee laajaa tuntemusta muun muassa. J2EE-ohjelmointialustasta, tapahtumienhallinnasta ja erityisesti integroitavasta tietojärjestelmästä.

6. JCA-määrittely sisältää selkeät määrittelyt resurssisovittimen toteuttamiseen osallistuvista osapuolista ja niiden vastuista. Tämä selkeyttää ja helpottaa osaltaan integrointiprojektin organisointia. Roolien jako on lähes vastaava aikaisemmin esitetyn EJB-määrittelyn sisältämän jaon kanssa.

7.4 JCA:n heikkouksia

Eräs merkittävä heikkous JCA-määrittelyssä on puute metadatan käsittelylle. Kun käsitellään useiden eri yrityksen sovelluksien sisältämää dataa, tulee myös tarve saada haettua integroitaviin sovelluksiin ja dataan liittyvää metadataa. Nykyinen JCA 1.0 -versio ei sisällä standardia liittymää metadatan käsittelyyn, joten sovittimeen pitää tehdä järjestelmäkohtainen liittymä metadatan käsittelyyn.

Myöskään nykyään hyvin suositetulle tiedon esitysmuodolle, XML:lle (Extensible Markup Languege), ei ole tukea JCA-määrittelyssä. CCI-liittymällä voidaan käsitellä dataa vain taulukkomuodossa. Jos integroitu järjestelmä palauttaa dataa XML-muodossa, se pitää muuntaa sovittimessa taulukkomuotoiseksi. Tämän kaltainen datan muuntaminen on turhaa työtä, varsinkin jos sovitinta käyttävä asiakassovellus ymmärtää XML-muotoista dataa.

CCI-liittymä tukee vain synkronista pyyntö/vastaus kommunikointia. Tämä tarkoittaa, että sovitinta käyttävä sovellus joutuu odottamaan vastausta pyyntöön, kunnes pyyntö on käsitelty. Monimutkaisissa integrointiskenaarioissa tarvitaan usein lisäksi asynkronista kommunikointia, joten myös tämä puute rajoittaa JCA:n perustuvien sovittimien käyttöä.

7.5 Muita huomioita

JCA-määrittelyn julkaisemisella saattaa olla myös positiivinen vaikutus J2EE-sovelluspalvelimien markkinointiin ja yleistymiseen. Yrityksien on helpompi lähteä kehittämään uusia J2EE-pohjaisia sovelluksia, koska tiedetään, että niiden integroiminen olemassa oleviin järjestelmiin on mahdollista ja aikaisempaa kustannustehokkaampaa. Oletetaan, että yrityksellä on toimittajan A:n toiminnanohjausjärjestelmä ja yritys on päättänyt kehittää liiketoiminnallisten vaatimuksien täyttämiseksi J2EE-sovelluksen. Toimija A toimittaa järjestelmäänsä sovittimen vain toimittaja B:n sovelluspalvelimeen, joten yrityksen täytyy valita toimittaja B:n sovelluspalvelin. Standardi määrittely voi muuttaa edellä mainittua esimerkkiä yrityksen hyödyksi. Toimittaja A:n tarvitsee toteuttaa vain yksi standardin mukainen sovitin. Yritys voi valita esimerkiksi lähes ilmaisen, avoimeen lähdekoodiin perustuvan sovelluspalvelimen kalliin kaupallisen tuotteen sijasta.

8. YHTEENVETO

Tämän tutkielman tarkoituksena on ollut selvittää, mitä JCA tarjoaa yrityksen sovellusten integroinnille ja miten sitä sovelletaan käytännössä.

Tutkielman toisen luvun alussa on esitetty perusteita yrityksen sovellusten integroinnille. Yrityksissä on hyvin monenikäisiä ja eri teknologioihin perustuvia sovelluksia, joita ei ole edes tarkoitettu toimimaan yhdessä toisten sovelluksien kanssa. Sovellukset on rakennettu yrityksen jonkin tietyn toiminnan ja yksikön tarpeita vastaamaan tietämättä, että joskus voitaisiin tarvita sovellusten välistä yhteistoimintaa. Yrityksen sovellusten integroinnin perimmäinen ongelma on siis yrityksissä olevien sovellusten välinen kommunikointikyvyn puute. Tarvetta sovellusten väliselle kommunikoinnille on aiheuttanut liiketoiminnalliset muutokset, siirtyminen elektroniseen liiketoimintaan ja yrityksen arvoketjun muutokset kilpailukyvyn säilyttämiseksi tai saavuttamiseksi.

Lisäksi toisessa luvussa on selvitetty, mitä eri tasoja yrityksen sovellusten integrointi käsittää. Yrityksen sovellusten integrointia voidaan toteuttaa data-, rajapinta-, metodi- tai käyttöliittymätasolla. Datatason integrointi on ehkä nopein, edullisin ja riskittömin lähestymistapa integroinnin ongelmille, mutta usein pelkän datan hakeminen tietokannoista tai tietovarastoista ei ole riittävää. Tarvitaan ratkaisuja, joissa pystytään suorittamaan kokonaisia liiketoimintaprosesseja sovellusten kesken.

Väliohjelmistot ovat hyvin oleellisia yrityksen sovellusten kehityksessä ja integroinnissa. Kolmannessa luvussa on kuvattu korkean tason väliohjelmistoarkkitehtuureja ja yleisimpiä väliohjelmistoteknologioita, joita on esitetty integrointiongelmien ratkaisuksi. Markkinoilla on tarjolla useita väliohjelmistotuotteita, jotka tukevat integrointia eri tasolla ja eri arkkitehtuurien

pohjalta. Yrityksen sovellusten integroinnin ongelman kannalta suosituimmat teknologiat liittyvät sanomanvälittäjiin, jotka mahdollistavat asynkronisen kommunikoinnin sovellusten välillä.

Luvussa neljä luodaan pohjaa JCA-määrittelyn ymmärtämiselle esittelemällä J2EE-ohjelmointialusta. J2EE-ohjelmointialusta tarjoaa monipuolisen joukon liittymiä hajautettujen ja monikerroksisien yritysten sovellusten kehittämiseen. J2EE-ohjelmointialusta on suunniteltu vastaaman tämän päivän yrityksen sovelluksien vaatimuksiin, kuten korkea käyttöaste, tietoturvallisuus, luotettavuus ja skaalautuvuus.

J2EE-ohjelmointialustan käsittely on aloitettu esittelemällä lyhyesti taustaa Java-ohjelmointikielestä ja Javan eri ohjelmointialustoista. Tämän jälkeen on esitelty suoritusympäristöjen ja säiliöiden perusasiat. Luvun keskeisimmän osan muodostaa J2EE:n sisältämien sovellusliittymien esittely, joista JCA-määrittelyn kannalta tärkeimmät on esitelty yksityiskohtaisemmin. Yksityiskohtaisemmin esiteltäväksi sovellusliittymiksi on valittu nimi- ja hakemistopalvelut tarjoava JNDI, tietovarastojen kanssa kommunikoinnin mahdollistava JDBC, komponenttimallin määrittelevä EJB sekä tapahtumanhallinnan määrittelevät JTA ja JTS.

Neljännen luvun muodostaman pohjan J2EE-ohjelmointialustasta jälkeen voidaan perehtyä tarkemmin JCA:han. JCA-määrittely määrittelee standardin arkkitehtuurin resurssisovittimille, joka mahdollistaa resurssisovittimen kytkemisen mille tahansa JCA-määrittelyä tukevalle J2EE-sovelluspalvelimelle. Viidennen luvun alussa on esitelty perusteita JCA:n käytölle ja selvitetty JCA:lle asetettuja tavoitteita sekä JCA:han perustuvan ohjelmistokehityksen eri osapuolet. Tämän jälkeen on pureuduttu syvemmälle JCA-määrittelyyn ja esitelty arkkitehtuurin eri osia. Luvun tärkeimmän osan muodostaa selvitys

resurssisovittimen yhteyden-, tapahtumien- ja tietoturvanhallinnan käytännöistä. Luvun lopussa on lisäksi esitelty yleistä CCI-asiakasliittymää.

Kun on selvitetty JCA-määrittelyn ominaisuuksia ja tarkempaa sisältöä, voidaan tutkia resurssisovittimen rakentamista. Kuudennessa luvussa on tarkasteltu resurssisovittimen toteuttamiseen tarvittavia sovellusliittymiä sekä konstruoitu integraatioskenaario ja resurssisovitin esimerkin mukaiseen ympäristöön. Resurssisovitin hyödyntää vahvasti sovelluspalvelimen tarjoamia palveluita yhteyden-, tapahtumien- ja tietoturvanhallinnan toteuttamiseksi. Resurssisovitin käytännössä delegoi yhteyden, tapahtumien ja tietoturvan hallintaan liittyvät toimenpiteet sovelluspalvelimelle. Voidaan ajatella, että resurssisovitin sisältää vain sovelluskohtaisia ohjeita edellä mainittujen palvelujen käytöstä. Tämä helpottaa huomattavasti resurssisovittimen kehittämistä. Valmiiden palveluiden hyödyntämisestä huolimatta, resurssisovittimen kehittäminen on melko työlästä. Resurssisovittimeen tulee toteuttaa useiden liittymien lisäksi integroitavan järjestelmän kommunikointikäytännöstä riippuvainen osuus, sekä liittymään sopiva yleinen asiakasliittymä.

Seitsemännessä luvussa on analysoitu JCA-määrittelyä. Luvussa on pohdittu määrittelyn vahvuuksia ja heikkouksia, verrattu sitä toisessa ja kolmannessa luvussa esitettyihin integroinnin tasoihin ja väliohjelmistoarkkitehtuureihin sekä kommentoitu JCA:lle esitettyjen tavoitteiden täyttymistä.

Tämän tutkimusotteeltaan käsitteellis-konstruktiivisen tutkielman keskeisempiä tuotoksia ovat katsaus yrityksen sovellusten integroinnissa käytettyihin väliohjelmistoihin ja selvitys JCA:n sisällöstä sekä konkreettinen esimerkki sen soveltamisesta yrityksen sovellusten integroinnissa.

Jatkotutkimusaiheena voisi olla hyödyllistä tutkia J2EE:n mahdollisuuksia asynkronisen sovellusintegroinnin kannalta hyödyntäen esimerkiksi Java Message Serviceä (JMS) ja Extensible Markup Languagea (XML).

LÄHDELUETTELO

Barish, G. 2001. The medium is the message. Intelligent Enterprise Magazine, Vol. 4, No.7, 27-31.

Beveridge, T. 2000. Building EAI Adapters for Legacy Systems. The Journal of Object-Oriented Programming, Vol. 13, No.4, 2-5.

Cheung, S. & Matena, V. 1999. JavaTM Transaction API (JTA) version 1.01. Sun Microsystems Inc.

Cheung, S. 1999. JavaTM Transaction Service (JTS) version 1.0. Sun Microsystems Inc.

Freedman, A. 1993. Computer Glossary. New York: American Management Association.

Hirschheim, R., Klein, H. & Lyytinen, K. 1995. Information Systems Development and Data Modelling: Conceptual and Philosophical Foundations. Cambridge: University Press.

Iyer, S. 1999. Enterprise Integration. Distributed Computing, Vol. 8. [online]. [Viitattu 10.12.2001]. Saatavilla: <http://207.87.15.219/fullarticle.asp?ID=8199994337AM.>.

Johannesson P., Jayaweera P., Wangler B. 2000. Application and process integation – Concepts, Issues and research directions [online], [viitattu

1.11.2001]. Saatavilla www-muodossa:
<http://www.dsv.su.se/~perjons/fossilpaul2.pdf>.

Johannesson P., Perjons, E. 2000. Desing principles for application integration. Advanced information systems engineering, 12th Int. Conference CaiSE'2000.

King, N. 2000. EAI Directions. Intelligent Enterprise Magazine, Vol. 3, No. 4, 41- 45.

Kuhn, D. 1990. On The Effective Use of Software Standards in System Integration. Teoksessa Ng, P., Ramamoorthy, C., Seifert, L. & Yeh, R. (toim.) Proceedings of the First International Conference on Systems Integration, Morristown, NJ, April 1990. IEEE Computer Society Press, 455-461.

Linthicum, D. 2000. Enterprise Application Integration. Reading, Massachusetts: Addison-Wesley Longman Inc.

Luoma, J., Muhonen, T. & Huomo, T. 1999. Uudistuva tietotekniikka arkkitehtuuri. Helsinki: HM&V Research Oy.

Modi, T. 2001. Plug 'n' Play Enterprise Apps, Enterprise application integration is made easy with J2EE Connector Architecture [online], [viitattu 15.4.2002]. Saatavilla www-muodossa:
<http://www.fawcette.com/javapro/2002_02/magazine/features/tmodi/>.

Monson-Haefel, R. & Chappel, D. 2001. Java Message Service. Sebastopol: O'Reilly & Associates, Inc.

Nance, B. 1996. Data Access Via ODBC and JDBC. Network Computing [online], [viitattu 15.1.2002]. Saatavilla www-muodossa: <http://www.networkcomputing.com/netdesign/odbc1.html>.

Niskanen, P., Kontio, K. & Vierimaa K. 2000. Enterprise Java. Helsinki: Oy Edita Ab.

Pawlan, M. 2001. The J2EE Tutorial. Sun Microsystems Inc [online], [viitattu 20.1.2002]. Saatavilla www-muodossa: < http://java.sun.com/j2ee/tutorial/1_3-fcs/doc/Overview.html >.

Porter, M. 1985. Kilpailuetu: Miten ylivoimainen osaaminen luodaan ja säilytetään. Espoo: Weilin+Göös.

Poutsaari, H. & Holopainen, M. 1996. Tietojenkäsittely. Porvoo: WSOY.

Pressman, R. 1997, Software Engineering, A Practitioner's Approach, Fourth Edition. New York: McGraw-Hill.

Rana, A. & Collins, A. 2001. Enterprise application integration with J2EE connectors, Java Junction. Intelligent EAI [online], [viitattu 20.1.2002]. Saatavilla www-muodossa:
<http://www.intelligenteai.com/feature/010416/feat1.shtml >.

Rodoni, J. 2001. The Java™ 2, Enterprise Edition (J2EE™) Connector Architecture's Resource Adapter [online], [viitattu 1.3.2002]. Saatavilla www-muodossa:
<http://developer.java.sun.com/developer/technicalArticles/J2EE/connectorclient/resourceadapter.html>

Roman E., Ambler, S., Jewel, T. 2002. Mastering Enterprise JavaBeans – Second Edition. New York: John Wiley & Sons.

Ruh, W., Maginnis F., Brown, W. 2001. Enterprise Application Integration. New York: A Wiley Tech Brief.

Shannon, B. 2001. Java™ 2 Platform Enterprise Edition Specification, v1.3. Final Release. Sun Microsystems Inc.

Sharma, R. 2001. Java™ 2 Platform Enterprise J2EE™ Connector Architecture Specification, v1.0. Sun Microsystems Inc.

Sharma, R. Stearns, B. & Ng, T. 2001. J2EE™ Connector Architecture and Enterprise Application Integration. New York: Addison-Wesley

Stavridou, V. 1999. Integration in Software Intensive Systems. The Journal on systems and software, Vol. 48, No. 2

Sun Microsystems Inc. 1999a. Java Naming and Directory Interface Application Programmin Interface (JNDI API).

Sun Microsystems Inc. 1999b. Getting Started with the JDBC API [online], [viitattu 20.1.2002]. Saatavilla www-muodossa: <http://java.sun.com/j2se/1.3/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html>.

Vander Hey, D. 2000. One Customer, One View. Intelligent Enterprise Magazine, Vol. 3, No. 4, 34-39.

Wangler, B., Paheerathan S. 2000. Horizontal and vertical integration organisational IT systems [online], [viitattu 1.11.2001]. Saatavilla www-muodossa: <http://www.dsv.su.se/~perjons/newhv2.pdf>.

LIITE 1

JCA-resurssisovitin

Tässä liitteessä on listattu esimerkkisovittimen, sitä käyttävän EJB-komponentin ja EJB-komponentin asiakassovelluksen lähdekoodit sekä resurssisovittimen että EJB-komponentin asennuksen kuvaustiedostot (XML-tiedostot).

Resurssisovitin koostuu seuraavista luokista:

1. GraduConnection.java
2. GraduConnectionEventListener.java
3. GraduConnectionFactory.java
4. GraduConnectionManager.java
5. GraduConnectionMetaData.java
6. GraduConnectionRequestInfo.java
7. GraduConnectionSpec.java
8. GraduInteraction.java
9. GraduManagedConnection.java
10. GraduManagedConnectionFactory.java
11. GraduMappedRecord.java
12. GraduMetaData.java
13. GraduRecordFactory.java
14. Util.java

EJB-sovellus koostuu seuraavista luokista:

1. Client.java
2. GraduPapu.java
3. GraduPapuBean.java
4. GraduPapuHome.java

Resurssisovittimen lähdekoodit

//GraduConnection.java

```
package gradu.jca;

import java.util.*;
import java.net.*;

import javax.resource.cci.*;
import javax.resource.ResourceException;
import javax.resource.spi.ConnectionEvent;
import javax.resource.spi.IllegalStateException;
import javax.resource.spi.*;
import javax.resource.NotSupportedException;

public class GraduConnection implements javax.resource.cci.Connection
{

    private boolean destroyed;
    private GraduManagedConnection gmc;

    GraduConnection(GraduManagedConnection gmc)
    {
        System.out.println("GraduConnection: konstruktori kutsuttu");
        this.gmc = gmc;
    }

    public Interaction createInteraction() throws ResourceException
    {
        return new GraduInteraction(this);
    }

    public javax.resource.cci.LocalTransaction getLocalTransaction()
      throws ResourceException
    {
        throw new NotSupportedException("LocalTransaction:ia ei
tueta.");
    }

    public void setAutoCommit(boolean autoCommit) throws
ResourceException
    {
        //ei toteuteta
    }

    public boolean getAutoCommit() throws ResourceException
    {
      return true;
    }

    public ResultSetInfo getResultSetInfo() throws ResourceException
    {
         throw new NotSupportedException("ResultSettia ei tueta.");
    }
```

```
    public void close() throws ResourceException
    {
        System.out.println("GraduConnection.close()");
        if (gmc == null)
        return;
        gmc.sendEvent(ConnectionEvent.CONNECTION_CLOSED, null, this);
        gmc = null;
    }

    public ConnectionMetaData getMetaData() throws ResourceException
    {
        return new GraduConnectionMetaData(gmc);
    }

    void invalidate()
    {
        gmc = null;
    }

    //palutetaan socket-yhetys.
    Socket getSocket() throws ResourceException
    {
        return gmc.getSocket();
    }
}
```

//GraduConnectionEventListener.java

```
package gradu.jca;

import javax.resource.*;
import javax.resource.spi.*;
import java.io.Serializable;
import javax.resource.spi.ConnectionEventListener;
import javax.resource.spi.ConnectionEvent;
import java.util.*;

public class GraduConnectionEventListener implements
ConnectionEventListener {

    private Vector listeners = null;

    public GraduConnectionEventListener()
    {
        listeners = new Vector();
    }

    void sendEvent(ConnectionEvent ce)
    {
        Vector list = (Vector) listeners.clone();
        int size = list.size();

        for (int i=0; i<size; i++)
        {
            ConnectionEventListener l =
                (ConnectionEventListener) list.elementAt(i);
```

```
            switch (ce.getId())
            {
                case ConnectionEvent.CONNECTION_CLOSED:
                    l.connectionClosed(ce);
                    break;
                case ConnectionEvent.LOCAL_TRANSACTION_STARTED:
                    l.localTransactionStarted(ce);
                    break;
                case ConnectionEvent.LOCAL_TRANSACTION_COMMITTED:
                    l.localTransactionCommitted(ce);
                    break;
                case ConnectionEvent.LOCAL_TRANSACTION_ROLLEDBACK:
                    l.localTransactionRolledback(ce);
                    break;
                case ConnectionEvent.CONNECTION_ERROR_OCCURRED:
                    l.connectionErrorOccurred(ce);
                    break;
                default:
                    throw new IllegalArgumentException(
                        "Laiton tapahtumatyyppi: " + ce.getId());
            }
        }
    }

    void addConnectorListener(ConnectionEventListener l)
    {
        listeners.addElement(l);
    }

    void removeConnectorListener(ConnectionEventListener l)
    {
        listeners.removeElement(l);
    }

    public void connectionClosed(ConnectionEvent event)
    {
        //ei toteuteta
    }

    public void connectionErrorOccurred(ConnectionEvent event)
    {
        sendEvent(event);
    }

    public void localTransactionCommitted(ConnectionEvent event)
    {
        sendEvent(event);
    }

    public void localTransactionStarted(ConnectionEvent event)
    {
        sendEvent(event);
    }

    public void localTransactionRolledback(ConnectionEvent event)
    {
        sendEvent(event);
    }
}
```

```
//GraduConnectionFactory.java

package gradu.jca;

import java.io.*;

import javax.resource.Referenceable;
import javax.resource.*;
import javax.resource.spi.*;
import javax.naming.Reference;
import javax.resource.cci.*;

public class GraduConnectionFactory implements
    ConnectionFactory, Serializable, Referenceable {

    private ManagedConnectionFactory mcf;
    private ConnectionManager cm;
    private Reference reference;
    private PrintWriter writer = null;

    public GraduConnectionFactory(ManagedConnectionFactory mcf,
                          ConnectionManager cm) {

      System.out.println("GraduConnectionFactory: konstruktori
kutsuttu");

        this.mcf = mcf;

        if (cm == null)
        {
            this.cm = new GraduConnectionManager();
        }
        else
        {
            System.out.println("\tA ConnectionManager -luokka: " +
                cm.getClass().toString());

            this.cm = cm;
        }
    }


    public GraduConnectionFactory(ManagedConnectionFactory mcf)
    {
        System.out.println("GraduConnectionFactory: " +
            "konstruktori ilman ConnectionManager-parametria. ");
        this.mcf = mcf;
        this.cm = new GraduConnectionManager();
    }

    public javax.resource.cci.Connection getConnection()
        throws ResourceException
    {
        System.out.println("GraduConnectionFactory.getConnection()");

        javax.resource.cci.Connection con = null;
        con = (javax.resource.cci.Connection)
cm.allocateConnection(mcf, null);
```

```
        return con;
    }

    public javax.resource.cci.Connection getConnection(
                                        ConnectionSpec properties)
        throws ResourceException
    {
        javax.resource.cci.Connection con = null;

        ConnectionRequestInfo info =
            new GraduConnectionRequestInfo(
            ((GraduConnectionSpec)properties).getUser(),
            ((GraduConnectionSpec)properties).getPassword());

        con = (javax.resource.cci.Connection)
cm.allocateConnection(mcf,info);

        return con;
    }

    public ResourceAdapterMetaData getMetaData() throws
ResourceException
    {
        throw new ResourceException("Metadataa ei ole saatavilla.");
    }

    public RecordFactory getRecordFactory() throws ResourceException
    {
        return new GraduRecordFactory();
    }

    public void setReference(Reference ref)
    {
        this.reference = ref;
    }

    public Reference getReference() {
        return reference;
    }

    public void setLogWriter(java.io.PrintWriter writer)
    {
      this.writer = writer;
    }

    public PrintWriter getLogWriter()
    {
        return writer;
    }

    public void setTimeout(int n)
    {
        //ei toteutettu
    }

    public int getTimeout()
    {
       return 1000000;
    }}
```

//GraduConnectionManager.java

```
package gradu.jca;

import javax.resource.*;
import javax.resource.spi.*;
import java.io.Serializable;

public class GraduConnectionManager
    implements ConnectionManager, Serializable {

    public Object allocateConnection(ManagedConnectionFactory mcf,
                                     ConnectionRequestInfo info)
        throws ResourceException
    {

System.out.println("GraduConnectionManager.allocateConnection()");

        ManagedConnection mc = mcf.createManagedConnection(null,
info);

        return mc.getConnection(null, info);
    }
}
```

//GraduConnectionMetaData.java

```
package gradu.jca;

import javax.resource.ResourceException;
import javax.resource.cci.*;

public class GraduConnectionMetaData implements ConnectionMetaData {

    private GraduManagedConnection gmc;

    public GraduConnectionMetaData (GraduManagedConnection mc)
    {
        this.gmc = mc;
    }

    public String getEISProductName() throws ResourceException
    {
        return "Gradu Socket Server";
    }

    public String getEISProductVersion() throws ResourceException {
        return "1.0";
    }

    public String getUserName() throws ResourceException
    {

        if (gmc.isDestroyed())
        {
            throw new javax.resource.spi.IllegalStateException
                ("ManagedConnection on tuhottu.");
```

```
        }

        return gmc.getUserName();
    }
}
```

//GraduConnectionRequestInfo.java

```
package gradu.jca;

import javax.resource.spi.ConnectionRequestInfo;

public class GraduConnectionRequestInfo implements
ConnectionRequestInfo {

    private String user;
    private String password;

    public GraduConnectionRequestInfo(String user, String password)
    {
        this.user = user;
        this.password = password;
    }

    public String getUser()
    {
        return user;
    }

    public String getPassword()
    {
        return password;
    }

    public boolean equals(Object obj)
    {
        if (obj == null) return false;

        if (obj instanceof GraduConnectionRequestInfo)
        {
            GraduConnectionRequestInfo other =
                (GraduConnectionRequestInfo) obj;

            return (isEqual(this.user, other.user) &&
                    isEqual(this.password, other.password));
        }
        else
        {
            return false;
        }
    }

    public int hashCode()
    {
        String result = "" + user + password;
        return result.hashCode();
    }
```

```
    private boolean isEqual(Object o1, Object o2)
    {
        if (o1 == null)
        {
            return (o2 == null);
        }
        else
        {
            return o1.equals(o2);
        }
    }
}
```

//GraduConnectionSpec.java

```
package gradu.jca;

import javax.resource.cci.*;

public class GraduConnectionSpec implements ConnectionSpec
{

    private String user;
    private String password;

    public GraduConnectionSpec(String user, String password)
    {
        this.user = user;
        this.password = password;
    }

    public GraduConnectionSpec()
    {
        //tyhjä konstruktori
    }

    public String getUser()
    {
        return user;
    }

    public String getPassword()
    {
        return password;
    }
}
```

//GraduInteraction.java

```
package gradu.jca;

import java.util.*;
import javax.resource.ResourceException;
import javax.resource.spi.ConnectionEvent;
import javax.resource.spi.IllegalStateException;
import javax.resource.cci.*;
import java.lang.reflect.*;
import java.lang.*;
```

```
import java.net.*;
import java.io.*;


public class GraduInteraction implements Interaction {

    Connection con = null;

    public GraduInteraction(Connection con)
    {
        this.con = con;
    }

    public javax.resource.cci.Connection getConnection()
    {
        return con;
    }

    public void close() throws ResourceException
    {
        con = null;
    }

    public boolean execute (InteractionSpec ispec, Record input,
Record output)
            throws ResourceException
    {
        if(!(input instanceof MappedRecord) ||
            !(output instanceof MappedRecord))
        {
          throw new ResourceException("Virheelliset parametrit!");
        }

        output = exec((MappedRecord)input,(MappedRecord)output);

        if (output != null)
        {
          return true;
        }
        else
        {
          return false;
        }
    }

    public Record execute (InteractionSpec ispec, Record input)
        throws ResourceException
    {
        if(!(input instanceof MappedRecord))
        {
            throw new ResourceException("GraduInteraction.execute(): "
+
            "input parametri väärää tyyppiä!");
        }

        MappedRecord output = new GraduMappedRecord();

        return exec((MappedRecord)input,output);
    }
```

```
    public ResourceWarning getWarnings() throws ResourceException
    {
        return null;
    }

    public void clearWarnings() throws ResourceException
    {
        //ei toteuteta
    }

    Record exec(MappedRecord input, MappedRecord output)
        throws ResourceException
    {
        try
        {

            System.out.println("GraduInteraction:exec()");

            Socket socket = ((GraduConnection)con).getSocket();

            PrintWriter out = new
PrintWriter(socket.getOutputStream(), true );

            BufferedReader in = new BufferedReader(
                new InputStreamReader(socket.getInputStream()));

            String key = (String) input.get("key");
            String value = (String) input.get("value");

            //Debug
            System.out.println("input:" + key +"/"+value);


out.println("<sanoma><hakuavain>"+key+"</hakuavain><hakuehto>"
                +value+"</hakuehto></sanoma>");

            out.flush();

            String fromServer = "";
            String reponse = "";

            while ((fromServer = in.readLine()) != null)
            {
                //Debug
                System.out.println("Server: " + fromServer);

                reponse = fromServer;
            }

            output.put("response",reponse);

            return output;

        }
        catch(Exception e)
        {
            throw new ResourceException(e.getMessage());
        }    } }
```

//GraduManagedConnection.java

```
package gradu.jca;

import java.io.*;
import java.util.*;

import javax.resource.*;
import javax.resource.spi.*;
import javax.resource.spi.security.PasswordCredential;
import javax.resource.spi.IllegalStateException;
import javax.resource.spi.SecurityException;
import javax.resource.NotSupportedException;
import javax.security.auth.Subject;
import javax.transaction.xa.XAResource;
import java.net.*;

public class GraduManagedConnection implements ManagedConnection {

    private GraduConnectionEventListener listener;
    private String user;
    private ManagedConnectionFactory mcf;
    private PrintWriter logWriter;
    private boolean destroyed;
    private Socket socket;

    GraduManagedConnection(ManagedConnectionFactory mcf,
        String user, Socket socket)
    {
        System.out.println("GraduManagedConnection: "+
            "konstruktori kutsuttu käyttäjällä: " + user);

        this.mcf = mcf;
        this.user = user;
        this.socket = socket;

        listener = new GraduConnectionEventListener();
    }

   public Object getConnection(Subject subject,
                                   ConnectionRequestInfo
connectionRequestInfo)
        throws ResourceException
   {
        System.out.println("GraduManagedConnection:getConnection()");

        PasswordCredential pc =
            Util.getPasswordCredential(mcf, subject,
connectionRequestInfo);

        if (pc == null)
        {
            if (user != null)
            {
                throw new SecurityException("Tunnistamaton käyttäjä");
            }
        }
        else
        {
```

```
            if (!pc.getUserName().equals(user))
            {
                throw new SecurityException("Tunnistamaton käyttäjä");
            }
        }

        checkIfDestroyed();

        GraduConnection con = new GraduConnection(this);

        return con;
    }

    public void destroy() throws ResourceException
    {
        System.out.println("GraduManagedConnection.destroy()");

        try
        {
            if (destroyed) return;
            cleanup();
        }
        catch (Exception ex)
        {
            throw new ResourceException(ex.getMessage());
        }
        finally
        {
            destroyed = true;

            try
            {
                socket.close();
            }
            catch(IOException e)
            {
                throw new ResourceException(e.getMessage());
            }
        }
    }

    public void cleanup() throws ResourceException
    {
        System.out.println("GraduManagedConnection.cleanup()");

        try
        {
            checkIfDestroyed();
        }
        catch (Exception ex)
        {
            throw new ResourceException(ex.getMessage());
        }
    }

    public void associateConnection(Object connection)
        throws ResourceException
    {
        throw new ResourceException(
```

```
            "GraduManagedConnection.associateConnection() "+
            "-metodia ei ole toteutettu.");
    }

    public void addConnectionEventListener(ConnectionEventListener
listener)
    {
        System.out.println(
            "GraduManagedConnection.addConnectionEventListener()" +
            " kutsuttu parametrina: " +
listener.getClass().toString());

        this.listener.addConnectorListener(listener);
    }

    public void removeConnectionEventListener
        (ConnectionEventListener listener)
    {
        this.listener.removeConnectorListener(listener);
    }

    public XAResource getXAResource() throws ResourceException
    {
        throw new NotSupportedException("XA-tapahtumanhallintaa ei
tueta.");
    }

    public LocalTransaction getLocalTransaction() throws
ResourceException
    {
        throw new NotSupportedException(
            "Paikallista tapahtumanhallintaa ei tueta.");
    }

    public ManagedConnectionMetaData getMetaData() throws
ResourceException
    {
        checkIfDestroyed();

        return new GraduMetaData(this);
    }

    public void setLogWriter(PrintWriter out) throws ResourceException
    {
        this.logWriter = out;
    }

    public PrintWriter getLogWriter() throws ResourceException
    {
        return logWriter;
    }


    Socket getSocket() throws ResourceException
    {
        checkIfDestroyed();

        return socket;
    }
```

```
    String getUserName()
    {
        return user;
    }

    void sendEvent(int eventType, Exception ex)
    {
        ConnectionEvent ce = null;
        if (ex == null)
            ce = new ConnectionEvent(this, eventType);
        else
            ce = new ConnectionEvent(this, eventType, ex);

        listener.sendEvent(ce);
    }

    void sendEvent(int eventType, Exception ex, Object
connectionHandle)
    {
        System.out.println("GraduManagedConnection.sendEvent()");

        ConnectionEvent ce = null;
        if (ex == null)
            ce = new ConnectionEvent(this, eventType);
        else
            ce = new ConnectionEvent(this, eventType, ex);
        ce.setConnectionHandle(connectionHandle);
        listener.sendEvent(ce);
    }

    boolean isDestroyed()
    {
        return destroyed;
    }

    private void checkIfDestroyed() throws ResourceException
    {
        if (destroyed)
        {
            throw new IllegalStateException("ManageedConnection
suljettu.");
        }
    }

    ManagedConnectionFactory getManagedConnectionFactory()
    {
        return mcf;
    }
}
```

//GraduManagedConnectionFactory.java

```
package gradu.jca;

import java.io.*;
import java.util.*;
```

```
import javax.resource.*;
import javax.resource.spi.*;
import javax.resource.spi.security.PasswordCredential;
import javax.resource.spi.SecurityException;
import javax.security.auth.Subject;
import javax.naming.Context;
import javax.naming.InitialContext;
import java.net.*;

public class GraduManagedConnectionFactory
    implements ManagedConnectionFactory, Serializable {

    private PrintWriter out = null;
    private String url = null;
    private Integer port = null;

    public GraduManagedConnectionFactory()
    {
         System.out.println(
            "GraduManagedConnectionFactory konstruktori kutsuttu.");
    }

    public Object createConnectionFactory(ConnectionManager cxManager)
        throws ResourceException
    {
        System.out.println(
"GraduManagedConnectionFactory.createConnectionFactory()");

        return new GraduConnectionFactory(this, cxManager);
    }

    public Object createConnectionFactory() throws ResourceException
    {

        System.out.println(
"GraduManagedConnectionFactory.createConnectionFactory()");

        return new GraduConnectionFactory(this, null);
    }

    public ManagedConnection createManagedConnection(Subject subject,
                                ConnectionRequestInfo info)
        throws ResourceException
    {

        System.out.println(
"GraduManagedConnectionFactory.createManagedConnection()");

        Socket socket;


        try
        {
            String userName = null;
            PasswordCredential pc =
                Util.getPasswordCredential(this, subject, info);
```

```
            if (pc == null)
            {
                System.out.println("PasswordCredential: null");
                socket = new Socket(url, port.intValue());

            }
            else
            {
                userName = pc.getUserName();

                //Debug
                System.out.println("Käyttäjä: " + userName );

                String password = new String(pc.getPassword());

                //Debug
                System.out.println("password: " + password );

                //luodaan socket
                socket = new Socket(url, port.intValue());

            }

            return new GraduManagedConnection(this, userName, socket);

        } catch (Exception ex) {
            throw new ResourceException(ex.getMessage());
        }

    }

    public ManagedConnection
            matchManagedConnections(Set connectionSet,
                                    Subject subject,
                                    ConnectionRequestInfo info)
        throws ResourceException
        {

        System.out.println(
"GraduManagedConnectionFactory.matchManagedConnections()");

        PasswordCredential pc =
            Util.getPasswordCredential(this, subject, info);

        String userName = null;
        if (pc != null)
        {
            userName = pc.getUserName();
        }

        Iterator it = connectionSet.iterator();
        while (it.hasNext())
        {
            Object obj = it.next();
            if (obj instanceof GraduManagedConnection)
            {
```

```
                GraduManagedConnection mc = (GraduManagedConnection)
obj;

                ManagedConnectionFactory mcf =
                    mc.getManagedConnectionFactory();

                if (Util.isEqual(mc.getUserName(), userName)
                    && mcf.equals(this))
                {
                    System.out.println(
                        "GraduManagedConnectionFactoryconnection." +
                        "matchManagedConnections() ok");
                    return mc;
                }
            }
        }

        System.out.println("GraduManagedConnectionFactoryconnection."
+
            "matchManagedConnections() NOK");

        return null;
    }

    public void setLogWriter(PrintWriter out)
        throws ResourceException
    {
        this.out = out;
    }

    public PrintWriter getLogWriter() throws ResourceException
    {
        return this.out;
    }

    public String getConnectionURL()
    {
        return url;
    }

    public void setConnectionURL(String url)
    {

        System.out.println(
            "GraduManagedConnectionFactory.setConnectionURL():" + url
);
        this.url = url;
    }

    public Integer getConnectionPort()
    {
        return port;
    }

    public void setConnectionPort(Integer port)
    {

        System.out.println(
```

```
            "GraduManagedConnectionFactory.setConnectionPort():" +
port );
        this.port = port;
    }

    public boolean equals(Object obj) {
        if (obj == null) return false;
        if (obj instanceof GraduManagedConnectionFactory) {
            String v1 = ((GraduManagedConnectionFactory) obj).url;
            String v2 = this.url;
            return (v1 == null) ? (v2 == null) : (v1.equals(v2));
        } else {
            return false;
        }
    }

    public int hashCode() {
        if (url == null) {
            return (new String("")).hashCode();
        } else {
            return url.hashCode();
        }
    }
}
```

//GraduMappedRecord.java

```
package gradu.jca;

import java.util.*;

public class GraduMappedRecord implements
javax.resource.cci.MappedRecord {

    private String recordName;
    private String description;
    private HashMap mappedRecord;

    public GraduMappedRecord()
    {
        mappedRecord= new HashMap();
    }

    public GraduMappedRecord (String name)
    {
        mappedRecord = new HashMap();
        recordName = name;
    }

    public String getRecordName()
    {
        return recordName;
    }

    public void setRecordName(String name)
    {
        recordName = name;
    }
```

```
public String getRecordShortDescription()
{
    return description;
}

public void setRecordShortDescription(String description)
{
    description = description;
}

public boolean equals(Object other)
{
    if(!(other instanceof GraduMappedRecord))
    {
        return false;
    }

    GraduMappedRecord m = (GraduMappedRecord)other;

    return (recordName == m.recordName)
        && mappedRecord.equals(m.mappedRecord);
}

public int hashCode()
{
    String result = "" + recordName;
    return result.hashCode();
}

public Object clone() throws CloneNotSupportedException
{
    return this.clone();
}

public void clear()
{
    mappedRecord.clear();
}

public boolean containsKey(Object key)
{
    return mappedRecord.containsKey(key);
}

public boolean containsValue(Object value)
{
    return mappedRecord.containsValue(value);
}

public Set entrySet()
{
    return mappedRecord.entrySet();
}

public Object get(Object o)
{
    return mappedRecord.get(o);
}
```

```
    public boolean isEmpty()
    {
        return mappedRecord.isEmpty();
    }

    public Set keySet()
    {
        return mappedRecord.keySet();
    }

    public Object put(Object key, Object value)
    {
        return mappedRecord.put(key,value);
    }

    public void putAll(Map c)
    {
        mappedRecord.putAll(c);
    }

    public Object remove(Object o)
    {
        return mappedRecord.remove(o);
    }

    public int size()
    {
        return mappedRecord.size();
    }

    public Collection values()
    {
        return mappedRecord.values();
    }
}
```

//GraduMetaData.java

```
package gradu.jca;

import javax.resource.ResourceException;
import javax.resource.spi.IllegalStateException;
import javax.resource.spi.*;

public class GraduMetaData implements ManagedConnectionMetaData
{

    private GraduManagedConnection mc;

    public GraduMetaData(GraduManagedConnection mc)
    {
        this.mc = mc;
    }

    public String getEISProductName() throws ResourceException
    {
        return "Gradu Socket Server";
```

```
    }

    public String getEISProductVersion() throws ResourceException
    {
        return "1.0";
    }

    public int getMaxConnections() throws ResourceException
    {
        return 1;
    }

    public String getUserName() throws ResourceException
    {
        if (mc.isDestroyed())
        {
            throw new IllegalStateException("ManagedConnection
tuhottu");
        }
        return mc.getUserName();
    }
}
```

//GraduConnection.java

```
package gradu.jca;

import javax.resource.cci.*;
import java.util.Map;
import java.util.Collection;
import javax.resource.ResourceException;

public class GraduRecordFactory implements
javax.resource.cci.RecordFactory
{

    public MappedRecord createMappedRecord(String recordName)
        throws ResourceException
    {
        return new GraduMappedRecord(recordName);
    }

    public IndexedRecord createIndexedRecord(String recordName)
        throws ResourceException
    {
      throw new ResourceException("IndexedRecord:ia ei tueta.");
    }

}
```

//Util.java

```
package gradu.jca;

import java.util.*;
import javax.security.auth.Subject;
import java.security.AccessController;
```

```
import java.security.PrivilegedAction;
import javax.resource.spi.security.PasswordCredential;
import javax.resource.ResourceException;
import javax.resource.spi.SecurityException;
import javax.resource.spi.*;

public class Util
{
    static public PasswordCredential getPasswordCredential
        (ManagedConnectionFactory mcf,
         final Subject subject, ConnectionRequestInfo info)
        throws ResourceException
    {

        if (subject == null)
        {
            if (info == null)
            {
                return null;
            }
            else
            {
                GraduConnectionRequestInfo myinfo =
                    (GraduConnectionRequestInfo) info;

                PasswordCredential pc =
                    new PasswordCredential(
                        myinfo.getUser(),
                        myinfo.getPassword().toCharArray());

                pc.setManagedConnectionFactory(mcf);

                return pc;
            }
        }
        else
        {
            Set creds = (Set) AccessController.doPrivileged
                (new PrivilegedAction() {
                    public Object run() {
                        return subject.getPrivateCredentials
                            (PasswordCredential.class);
                        }
                    });

            PasswordCredential pc = null;

            Iterator iter = creds.iterator();

            while (iter.hasNext())
            {
                PasswordCredential temp =
                    (PasswordCredential) iter.next();

                if (temp.getManagedConnectionFactory().equals(mcf))
                {
                    pc = temp;
                    break;
                }
```

```
            }
            if (pc == null)
            {
                throw new SecurityException("PasswordCredentialia ei
löydy.");
            }
            else
            {
                return pc;
            }
        }
    }

    static public boolean isEqual(String a, String b)
    {
        if (a == null)
        {
            return (b == null);
        }
        else
        {
            return a.equals(b);   } } }
```

EJB-komponentin ja asiakasovelluksen lähdekoodit

//Client.java

```
package gradu.jca.test;

import java.util.Properties;
import java.rmi.RemoteException;
import javax.ejb.CreateException;
import javax.naming.Context;
import javax.naming.InitialContext;
import javax.naming.NamingException;
import javax.rmi.PortableRemoteObject;
import java.io.*;
import java.net.*;

public class Client
{
    private String url;
    private GraduPapuHome home;

    public static void main(String[] args) throws NamingException
    {
        String url = "t3://localhost:7001";
        if (args.length > 1)
        {
            url = args[0];
        }

        Client client = null;

        try
```

```
        {
            client = new Client(url);
            client.doIt();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }

    public Client(String url) throws NamingException
    {
        this.url = url;
        home = lookupHome();
    }

    public void doIt() throws CreateException, RemoteException
    {
        GraduPapu graduPapu =
            (GraduPapu)PortableRemoteObject.narrow(
                home.create(),GraduPapu.class);

        System.out.println(graduPapu.haeAsiakas(
            "etunimi", "Matti"));
    }

    private GraduPapuHome lookupHome() throws NamingException
    {
        Context ctx = getInitialContext();
        Object home = ctx.lookup("GraduPapuEJB");

        return (GraduPapuHome)PortableRemoteObject.narrow(
                                home, GraduPapuHome.class);
    }

    private Context getInitialContext() throws NamingException
    {
        Properties h = new Properties();

        h.put(Context.INITIAL_CONTEXT_FACTORY,
            "weblogic.jndi.WLInitialContextFactory");
        h.put(Context.PROVIDER_URL, url);

        return new InitialContext(h);
    }
}
```

//GraduPapu

```
package gradu.jca.test;

import java.rmi.RemoteException;
import javax.ejb.EJBObject;


/**
 *
```

```
 *  Etärajapinta GraduPapuBean-komponentille
 *
 */
public interface GraduPapu extends EJBObject
{

  /*
   * @param     String hakuehto
   *            - kentän nimi jolla asiakasta haetaan:
   *              "sukunimi|etunimi"
   * @param     String arvo
   *            - arvo jolla asiakasta haetaan, esim. "Pentti"
   */
  public String haeAsiakas(String hakuehto, String arvo)
    throws RemoteException;
}
```

//GraduPapuBean.java

```
package gradu.jca.test;

import javax.ejb.CreateException;
import javax.ejb.SessionBean;
import javax.ejb.SessionContext;
import javax.naming.InitialContext;
import javax.naming.NamingException;

/**
 *
 *  Tämä EJB-komponentti simuloi J2EE-sovellusta,
 *  joka käyttää JCA-sovitinta ja CCI-rajapintaa
 *  tiedonhakemiseen integroidusta tietojärjestelmästä
 *
 */
 public class GraduPapuBean implements SessionBean
 {

    private SessionContext ctx;

    public void ejbCreate () throws CreateException
    {
    }

    public void ejbRemove()
    {
    }

    public void ejbActivate()
    {
    }

    public void ejbPassivate()
    {
    }

    public void setSessionContext(SessionContext ctx)
    {
        this.ctx = ctx;
```

```
    }

    public String haeAsiakas(String hakuehto, String arvo)
    {
        String retVal = "";

        System.out.println("GraduPapu.haeAsiakas()");

        InitialContext initCtx = null;

    try
    {
        //luodaan InitialContext
        initCtx = new InitialContext();

        //Haetaan JNDI:llä ConnectionFactory
        javax.resource.cci.ConnectionFactory cf =
            (javax.resource.cci.ConnectionFactory)
                initCtx.lookup("eis/GraduAdapter");

        //luodaan yhteys
        javax.resource.cci.Connection connection =
            cf.getConnection();

        javax.resource.cci.Interaction interaction =
            connection.createInteraction();

        javax.resource.cci.MappedRecord recordIn =
            cf.getRecordFactory().createMappedRecord("");

        //tallennetaan hakuehdot
        recordIn.put("key", hakuehto);
        recordIn.put("value", arvo);

        //suoritetaan haku
        javax.resource.cci.MappedRecord recordOut =
            (javax.resource.cci.MappedRecord)interaction.execute(
                null, (javax.resource.cci.Record)recordIn);

        retVal = (String)recordOut.get("response");

        connection.close();

    }
    catch(Exception e)
    {
        e.printStackTrace();
    }

    //palautetaan asiakassovellukselle hakutulos
    return retVal;

    }
}
```

//GraduPapu.java

```
package gradu.jca.test;
```

```
import javax.ejb.CreateException;
import javax.ejb.EJBHome;
import java.rmi.RemoteException;

/**
 *
 *  Kotirajapinta GraduPapuBean-komponentille
 *
 */
public interface GraduPapuHome extends EJBHome
{
  public GraduPapu create() throws
    CreateException, RemoteException;
}
```

Resurssisovittimen asennustiedostot

```
<!- ra.xml -->

<?xml version="1.0" encoding="UTF-8"?>

<!DOCTYPE connector PUBLIC '-//Sun Microsystems, Inc.//DTD Connector
1.0//EN' 'http://java.sun.com/j2ee/dtds/connector_1_0.dtd'>

<connector>
    <display-name>GraduAdapter</display-name>
    <vendor-name>Markus</vendor-name>
    <spec-version>1.0</spec-version>
    <eis-type>Socket Server</eis-type>
    <version>1.0</version>
    <resourceadapter>
        <managedconnectionfactory-
class>gradu.jca.GraduManagedConnectionFactory</managedconnectionfactor
y-class>
        <connectionfactory-
interface>javax.resource.cci.ConnectionFactory</connectionfactory-
interface>
        <connectionfactory-impl-
class>gradu.jca.GraduConnectionFactory</connectionfactory-impl-class>
        <connection-
interface>javax.resource.cci.Connection</connection-interface>
        <connection-impl-class>gradu.jca.GraduConnection</connection-
impl-class>
        <transaction-support>NoTransaction</transaction-support>
        <config-property>
            <config-property-name>ConnectionURL</config-property-name>
            <config-property-type>java.lang.String</config-property-
type>
            <config-property-value>127.0.0.1</config-property-value>
        </config-property>
        <config-property>
            <config-property-name>ConnectionPort</config-property-
name>
            <config-property-type>java.lang.Integer</config-property-
type>
```

```
            <config-property-value>9696</config-property-value>
        </config-property>
        <authentication-mechanism>
            <authentication-mechanism-
type>BasicPassword</authentication-mechanism-type>
            <credential-
interface>javax.resource.security.PasswordCredential</credential-
interface>
        </authentication-mechanism>
        <reauthentication-support>false</reauthentication-support>
    </resourceadapter>
</connector>
```

<!- weblogic-ra.xml -->

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-connection-factory-dd PUBLIC '-//BEA Systems,
Inc.//DTD WebLogic 6.0.0 Connector//EN'
'http://www.bea.com/servers/wls600/dtd/weblogic600-ra.dtd'>

<weblogic-connection-factory-dd>

    <connection-factory-name>GraduConnectionFactory</connection-
factory-name>
    <jndi-name>eis/GraduAdapter</jndi-name>
    <pool-params>
                        <initial-capacity>1</initial-capacity>
    </pool-params>
    <map-config-property>
        <map-config-property-name>ConnectionURL</map-config-property-
name>
        <map-config-property-value>127.0.0.1</map-config-property-
value>
    </map-config-property>
    <map-config-property>
        <map-config-property-name>ConnectionPort</map-config-property-
name>
        <map-config-property-value>9696</map-config-property-value>
    </map-config-property>
<security-principal-map>
   <map-entry>
           <initiating-principal>*</initiating-principal>
              <resource-principal>
                  <resource-username>g_user</resource-username>
                      <resource-password>g_password</resource-password>
              </resource-principal>
   </map-entry>
</security-principal-map>
</weblogic-connection-factory-dd>
```

EJB-komponentin asennustiedostot

<!- ejb-jar.xml -->

```
<?xml version="1.0"?>
```

```
<!DOCTYPE ejb-jar PUBLIC '-//Sun Microsystems, Inc.//DTD Enterprise
JavaBeans 1.1//EN' 'http://java.sun.com/j2ee/dtds/ejb-jar_1_1.dtd'>

<ejb-jar>

        <enterprise-beans>
        <session>
                <ejb-name>GraduPapu</ejb-name>
                <home>gradu.jca.test.GraduPapuHome</home>
                <remote>gradu.jca.test.GraduPapu</remote>
                <ejb-class>gradu.jca.test.GraduPapuBean</ejb-
class>
                <session-type>Stateless</session-type>
                <transaction-type>Container</transaction-type>
                <resource-ref>
                        <res-ref-name>eis/GraduAdapter</res-
ref-name>
                        <res-type>javax.sql.DataSource</res-
type>
                        <res-auth>Container</res-auth>
                </resource-ref>
        </session>
        </enterprise-beans>

        <assembly-descriptor>
        <container-transaction>
                <method>
                        <ejb-name>GraduPapu</ejb-name>
                        <method-intf>Remote</method-intf>
                        <method-name>*</method-name>
                </method>
                <trans-attribute>Supports</trans-attribute>
        </container-transaction>
        </assembly-descriptor>

 </ejb-jar>
```

<!- weblogic-ejb-jar.xml -->

```
<?xml version="1.0"?>

<!DOCTYPE weblogic-ejb-jar PUBLIC '-//BEA Systems, Inc.//DTD WebLogic
5.1.0 EJB//EN' 'http://www.bea.com/servers/wls510/dtd/weblogic-ejb-
jar.dtd'>

<weblogic-ejb-jar>
        <weblogic-enterprise-bean>
        <ejb-name>GraduPapu</ejb-name>
             <caching-descriptor>
                  <max-beans-in-cache>10</max-beans-in-cache>
             </caching-descriptor>
             <reference-descriptor>
                       <resource-description>
                            <res-ref-
name>eis/GraduAdapter</res-ref-name>
                            <jndi-name>eis/GraduAdapter</jndi-
name>
```

```
                </resource-description>
            </reference-descriptor>
        <jndi-name>GraduPapuEJB</jndi-name>
        </weblogic-enterprise-bean>
</weblogic-ejb-jar>
```