Kai Lahti

ORGANISAATION JÄRJESTELMÄINTEGRAATIOT

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

6.10.2003

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Kai Lahti

Yhteystiedot: Survontie 46 B 82, 40520 Jyväskylä

Työn nimi: Organisaation järjestelmäintegraatiot

Title in English: Enterprise Application Integration

Työ: Pro gradu -tutkielma

Sivumäärä: 69

Linja: Ohjelmistotekniikka

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: EAI, sovellus, järjestelmä, integraatio, väliohjelmisto

Keywords: EAI, application, system, integration, middleware

Tiivistelmä: Tämän pro gradu -tutkielman aiheena ovat organisaation järjestelmäintegraatiot. Tutkielmassa on esitetty integroinnin vaatimukset, tavoitteet ja perusratkaisut. Integroinnin teoriaa on käsitelty mahdollisten liityntöjen kautta sekä arkkitehtuuriselta kannalta. Lähempänä käytännön ratkaisuja tässä tutkielmassa ovat väliohjelmistot ja niiden eri tyypit sekä suunnittelumallit. Tutkielmassa tuodaan myös esille kuhunkin teoriaosuuteen liittyviä käytössä olevia teknologioita ja standardeja. Lopuksi käydään läpi Metso Paper Oy:lle tehty integrointiratkaisu.

Abstract: The subject of this master thesis is enterprise application integration. This thesis is an introductory to the requirements, objectives and basic solutions of integration. The theory of integration is considered through integration levels and architectural point of view. Closer to practical solutions in this thesis are the middleware types and models. The integration technologies and standards related to these theories are also discussed. As an example, the integration solution made to Metso Paper Inc. is introduced.

Esipuhe

Tämän pro gradu -tutkielman kirjoittaminen on ollut hyvin opettavaista. Alussa oli vain otsikko ja siitä se sitten lähti. Nyt työn viime hetkillä olo on huojentunut ja tyytyväinen. Haluaisin kiittää Metso Paper Oy:tä tämän tutkielman mahdollistamisesta. Kiitokset myös Metso Paper Oy:n työntekijöille ajastanne ja erityiset kiitokset Jari P. Hämäläiselle sekä Jorma Salliselle hyvästä yhteistyöstä. Kiitos professori Tommi Kärkkäiselle ja Sami Äyrämölle tarjoamastaan ohjauksesta sekä Turo Kilpeläiselle hyvästä yhteistyöstä projektin parissa. Kiitos myös ystäville, kotiväelle sekä erityisesti Marialle tuesta ja avusta tutkielman edetessä.

Syyskuussa 2003 *Kai Lahti*

Termiluettelo

API	Application Programing Interface Sovelluksen yleinen ohjelmointirajapinta
CLI	Call-Level Interface Komentorajapinta, jonka avulla voidaan liittyä esimerkiksi tietokantaan
COM	Component Object Model Microsoftin objektiteknologia
CORBA	Common Object Reguest Broker Architecture Hajautettujen objektien spesifikaatio
DCOM	Distributed Component Object Model Microsoftin hajautettujen objektien teknologia
DLL	Dynamic Link Library Windowsissa toimiva funktio- ja proseduurikirjasto
EAI	Enterprise Application Integration Organisaation sovellusten/järjestelmien integrointi
EJB	Enterprise Java Beans SUN Microsystemsin komponenttiarkkitehtuuri
IDL	Interface Definition Language Hajautettujen objektien rajapinnanmäärittelykieli
J2EE	Java 2 Enterprise Edition Javan sovellusarkkitehtuuri hajautettujen ja monitasoisten sovellusten suunnitteluun

JDBC	Java Database Connectivity Java-pohjainen komentorajapinta tietolähteiden lukemiseen SQL:n avulla
JMS	Java Message Service Java-ratkaisu sanomapohjaiseen viestien välitykseen
JNDI	Java Naming and Directory Interface Javan nimi- ja hakemistopalvelu
JTA	Java Transaction API Mahdollistaa transaktioiden käytön J2EE-arkkitehtuurissa
MOM	Message Oriented Middleware Sanomapohjaiseen viestien välitykseen erikoistuneet väliohjelmistot
ODBC	Open Database Connectivity Microsoftin komentorajapinta erilaisten tietolähteiden käyttöön SQL:n avulla
OLE DB	Object Linking and Embedding to Database Oliomainen lähestymistapa tietolähteiden lukemiseen
OMG	Object Management Group Oliopohjaisten tekniikoiden standardointi-järjestö, joka vastaa esimerkiksi CORBA:n spesifikaatioista
ORB	Object Reguest Broker Hajautettujen objektien kommunikoinnin verkon yli mahdollistava komponentti

RMI	Remote Method Invocation Hajautettujen objektien tekniikka joka on puhdas Java-ratkaisu
RPC	Remote Procedure Call Tekniikka, jolla toisen sovelluksen proseduureja voidaan kutsua verkon yli aivan kuin omia
SQL	Structured Query Language Yleinen tietokantojen kyselykieli
TPM	Transaction Processing Monitors Tapahtumamonitorit, jotka hallitsevat kahden tai useamman sovelluksen välisiä tapahtumia
XML	Extensible Markup Language Metakielistandardi, jolla voidaan määritellä dokumenttien rakenne

Sisältö

1 Johdanto

Nykyään erilaiset tietokonejärjestelmät ovat olennainen osa kaikkien organisaatioiden toimintaa. Jopa pienimmilläkin yrityksillä on yleensä jonkinlainen järjestelmä käytössään. Ajan myötä järjestelmät käyvät vanhanaikaiseksi, eivätkä oikein vastaa kaikkia syntyneitä tarpeita. Ratkaisuksi tähän päätetään hankkia uusi järjestelmä vanhan tilalle tai sen rinnalle. Tämä tuo mukanaan myös tarpeen järjestelmien integroinnille. Tässä pro gradu -tutkielmassa selvitetään erilaisia mahdollisuuksia toteuttaa tämä integraatio.

1.1 Tausta

Lähtökohdan tälle pro gradu -tutkielmalle loi Metso Paper Oy:n tarve eräänlaiselle integrointiratkaisulle sekä selvitykselle järjestelmien hyötykäyttömahdollisuuksista. Tämän ongelmakentän ympärille syntyi ja kehittyi tämä tutkielma.

1.2 Tavoitteet

Tämän tutkielman tavoitteena on kuvata, mitä tarkoitetaan organisaation järjestelmäintegraatiolla ja mitkä ovat sen vaatimukset ja tavoitteet. Tarkoituksena on myös tuoda esille, miten erilaiset integrointiratkaisut rakentuvat, minkälaisia ne ovat arkkitehtuureiltaan ja minkälaisia teknologioita apuna käyttäen niitä on mahdollista toteuttaa. Lopuksi tavoitteena on tutkia Metso Paper Oy:n järjestelmien päivityksessä syntynyttä tilannetta, pohtia erilaisia integrointimahdollisuuksia sekä suunnitella ja toteuttaa oma ratkaisu.

1.3 Tutkielman rakenne

Tutkielma etenee johdannon jälkeen järjestelmäintegraatioiden yleisellä kuvauksella luvussa kaksi. Luvussa tuodaan myös esille järjestelmäintegraatioiden vaatimuksia ja tavoitteita. Tämän jälkeen luvussa kolme pohjustetaan integroinnin teoriaa kertomalla väliohjelmistoista ja niiden suunnittelumalleista. Luvussa kerrotaan yleisesti, mitä väliohjelmistot ovat, ja käydään läpi erilaisia väliohjelmistojen suunnittelumalleja. Jatkona edelliseen luvussa neljä käydään läpi eri väliohjelmistotyyppejä ja niihin liittyviä

toteutusteknologioita. Tarkoituksena on kuvata, mihin tarkoitukseen mikäkin teknologia parhaiten soveltuu. Luvussa viisi siirrytään tarkastelemaan eri integrointitasoja, mikä auttaa valitsemaan oikeanlaisen lähestymisen integrointikohteisiin. Luvussa kuusi käsitellään erilaisia integrointiarkkitehtuureja. Tämä luku antaa lähtökohdan integrointiratkaisun suunnittelulle ja toteutukselle. Luvussa seitsemän kerrotaan integroinnin teknologioista ja standardoinneista. Luvussa kahdeksan esitellään yleisesti tiedonvarastointia ja sen integrointimahdollisuuksia. Kokeellista osuutta käsitellään luvussa yhdeksän. Luvussa tutkitaan Metso Paper Oy:n uuden ja vanhan järjestelmän integrointimahdollisuuksia. Lisäksi esitellään toteutetun väliohjelmiston toimintaa. Lopuksi luvussa kymmenen tehdään yhteenveto ja pohditaan tutkielman sisältöä.

2 Järjestelmäintegraatiot

Tässä luvussa kerrotaan hieman järjestelmäintegraatioista yleensä; mitä ne ovat, mihin yritykset tarvitsevat niitä, mitä niillä halutaan saavuttaa ja millä tavoin integrointiongelmat ratkaistaan?

2.1 Mitä ovat organisaation järjestelmäintegraatiot?

Organisaation järjestelmäintegraatio (*EAI*) tarkoittaa eri tekniikoilla ja alustoilla toteutettujen sovellusten yhteistoiminnan mahdollistamista. Stonebraker määrittelee EAI:n olevan erillisten tietosaarekkeiden (*information island*) yhdistämisen mahdollistava teknologia [Sto99]. Linthicum puolestaan tarkentaa tätä sanomalla EAI:n olevan rajoittamatonta tiedon ja prosessien jakamista minkä tahansa toisiinsa kytkettyjen sovellusten ja tietolähteiden välillä organisaation sisällä [Lin00b].

Integroinniksi voidaan yksinkertaisimmillaan lukea kahden samalla koneella olevan sovelluskomponentin yhdistäminen niin, että ne muodostavat yhdessä toimivan kokonaisuuden. Yhteistoiminnan mahdollistamiseksi täytyy sovelluksilla olla joku yhteydenpitokanava sekä yhdenmuotoinen tietosanoma. Nämä tietosanomat voivat olla esimerkiksi tapahtumaviestejä komponenttien välillä tai tiedostoja kiintolevyllä. Tietosanomat voivat sisältää sovellusten välillä siirrettävää tietoa, mutta myös komentoja sovellukselta toiselle. [Lin00a, Lin00b]

2.2 Integrointitarpeet

Nykyään organisaatioissa on käytössä yhä enemmän sovelluksia ja erilaisia järjestelmiä. Mitä isompi ja vanhempi yritys on, sitä enemmän tietoa on hajallaan ympäri organisaatiota. Yrityksen kannalta on kallista pitää esimerkiksi useampia henkilörekistereitä erilaisiin järjestelmiin liittyen. Jonkun tiedon saanti ei myöskään aina ole käytännön syistä mahdollisia, vaikka periaatteessa tieto olisikin tallella yrityksen järjestelmissä. Tällainen ongelma syntyy, kun toisen järjestelmän tallentama tieto ei olekaan luettavissa toisella järjestelmällä. Toisin sanoen järjestelmien ymmärtämät tallennusmuodot eroavat liikaa toisistaan. [Lin00b, Lee03]

2.3 Integroinnin tavoitteet

Laajassa mittakaavassa organisaation järjestelmäintegraatio on isojen järjestelmien tietojen ja toiminnan vuorovaikutuksen mahdollistamista. Tarkoituksena on varmistaa tiedon kulku ja helppo liittyminen toisiin sovelluksiin ja järjestelmiin. Helpolla liittymisellä tarkoitetaan, ettei liittyminen vaadi isoja muutoksia integroitaviin tai integroinnin kohteena oleviin sovelluksiin. Jos vain on mahdollista, niin muutoksien tarve pyritään poistamaan kokonaan. Järjestelmäintegraatioita tehtäessä pyritään ottamaan huomioon myös organisaatiossa vallitsevat toimintaprosessit ja mahdollisesti yhdistämään nämä uuden toimintamallin mukaiseksi tai sulauttamaan uusi prosessi vanhojen prosessien mukaan. Organisaation järjestelmäintegraatio ei ole pelkästään olemassa olevien sovellusten ja prosessien yhdistämistä, vaan myös varautumista tulevaisuuden tarpeisiin. Tämän vuoksi ratkaisuista pyritään tekemään mahdollisimman yleiskäyttöisiä, mikä onnistuu käyttämällä hyväksi havaittuja, standardoituja rajapintoja. [Lin00a, Lin00b]

2.4 Integroinnin ratkaisukeinot

Mietittäessä integroinnin ratkaisukeinoja on hyvä aloittaa suunnittelu ylemmän tason ratkaisuista. Tällöin järjestelmän kokonaisrakenne suunnitellaan ensin toimivaksi ja sen jälkeen tarkennetaan tätä aina varsinaiseen toteutukseen asti. Seuraavassa on esitetty integrointitehtävien ratkaisuprosessi:

1. Kartoitetaan ongelma-alue ja integroinnin piiriin kuuluvat nykyiset ja mahdolliset tulevat järjestelmät. Tähän voisi käyttää esimerkiksi genre-pohjaista tietovirta-analyysia [Päi01].
2. Suunnitellaan integrointiarkkitehtuuri luvussa kuusi esitettyjen mallien pohjalta.
3. Tutkitaan, millä tasolla integrointi on mahdollista toteuttaa kuhunkin järjestelmään. Tässä voi käyttää ohjeena lukua viisi.
4. Suunnitellaan väliohjelmisto yllä olevia valintoja ja väliohjelmistojen suunnittelumalleja sekä tyyppejä soveltaen. Näistä löytyy lisää tietoa luvuissa kolme ja neljä.
5. Toteutetaan ja testataan integrointiratkaisu.

3 Väliohjelmistot ja niiden suunnittelumallit

Väliohjelmistot ovat tärkeä osa järjestelmäintegrointia. Lähes kaikki integrointiratkaisut käyttävät jonkinlaista väliohjelmistoa. Väliohjelmistoja on monenlaisia ja tässä luvussa esitellään, mitä väliohjelmistot ovat ja minkälaisia suunnittelumalleja niille käytetään.

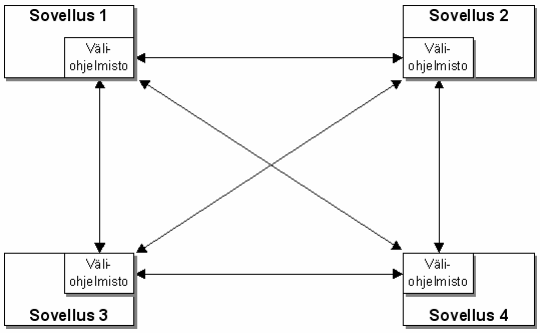
3.1 Mitä ovat väliohjelmistot?

Väliohjelmistot (*middleware*) ovat komponentteja, jotka ovat ohjelman käyttäjien ja tietolähteiden välissä [Rit98]. Käyttäjänä voi tässä olla jokin sovellus tai jopa loppukäyttäjä, joka haluaa käyttää tietolähdettä. Toisin sanoen väliohjelmistot ovat komponentteja, jotka mahdollistavat erilaisten kokonaisuuksien kommunikoinnin toistensa kanssa [Lin01]. Väliohjelmistot ovat siis se yhdistävä tekijä, jolla voimme muuntaa esimerkiksi yhden tietokannan käyttöön tarkoitetun ohjelman käyttämään toista tietokantaa. Väliohjelmistot myös osaltaan yksinkertaistavat organisaation järjestelmän rakennetta. Niiden avulla voidaan piilottaa käyttöjärjestelmä ja verkkoteknologia sovellustenkehittäjiltä. Väliohjelmistot liittyvät sovelluksiin jonkin rajapinnankautta. Rajapintoja voivat olla sovelluksen varsinainen ohjelmointirajapinta, mutta myös esimerkiksi yhteisesti käytettävä tiedosto tai vaikka sovelluksen oma käyttöliittymä. Nämä rajapintojen eri tasot on esitelty integrointi tasoista kertovassa luvussa viisi. [Agh02]

3.2 Väliohjelmistojen suunnittelumallit

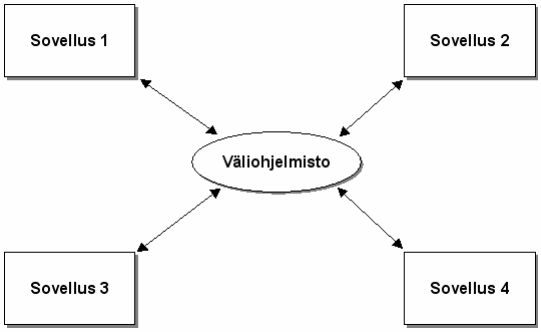
Tässä osiossa käsitellään väliohjelmistojen suunnittelumalleja. Suunnittelumallien on tarkoitus helpottaa ohjelmistokehittäjän työtä. Mallien avulla ohjelmistonkehittäjä näkee erilaisia ratkaisuja yleisiin ongelmiin. Väliohjelmistojen tapauksessa on tarkoitus hahmottaa toimintaperiaatteita kahden tai useamman sovelluksen väliseen kommunikointiin. Tässä esitettyjä suunnittelumalleja voi käyttää myös yhdistelminä toisistaan, eli voidaan tehdä esimerkiksi synkronisia pisteestä pisteeseen tai asynkronisia jonotusyhteydellisiä väliohjelmistoja. [Lin00b]

3.2.1 Pisteestä pisteeseen ja monesta moneen



Kuva 3-1: Pisteestä pisteeseen -malli [Lin00b].

Pisteestä pisteeseen (*point-to-point*) -mallissa väliohjelmistot tarjoavat sovelluksille suoran yhteyden toisiinsa. Väliohjelmistot voivat olla pieniä yhteysohjelmia tai ne voivat sisältyä itse sovelluksiin. Tämä on yksinkertainen ja nopea tapa yhdistää kaksi sovellusta toisiinsa. Mallilla on myös varjopuolensa, sillä sovellusten lukumäärän (n) kasvaessa kasvaa yhteyksien määrä neliöllisesti (n^2) ja tästä seuraa vaikeasti hallittava järjestelmä. [Lin00b]



Kuva 3-2: Monesta moneen -malli [Lin00b].

Monesta moneen (*many-to-many*) -mallissa kahden tai useamman sovelluksen välinen kommunikointi hoidetaan erillisellä väliohjelmistolla. Vaikka kahden sovelluksen välinen yhteys ei olekaan tässä mallissa aivan yhtä nopea kuin pisteestä pisteeseen mallissa, on sen etuna hyvä laajennettavuus, sillä yksi uusi sovellus vaatii vain yhden uuden yhteyden. [Lin00b]

3.2.2 Synkroninen ja asynkroninen

Synkronisessa (*synchronous*) mallissa toimintajärjestys on määrätty. Kun kutsumme jotain toista sovellusta, niin oma sovelluksemme jää odottamaan kunnes kutsu on suoritettu. Tämän mallin mukaan toimiminen yksinkertaistaa integroinnin suunnittelua. Koska malli on helposti ymmärrettävissä, on sitä myös helppo ylläpitää. David Linthicum [Lin00b] kuitenkin sanoo: ”*They suck the life from the system*”, eli synkroniset integroinnit hidastavat tehokkaimmatkin koneet, jos siirtotiet eivät ole kunnossa tai jos kutsuja käytetään paljon. Synkronista mallia käytetään kuitenkin esimerkiksi pankkiautomaattisovelluksissa tai muissa järjestelmissä, joissa suoritusjärjestyksellä on suuri merkitys. [Lin01]

Asynkroninen (*asynchronous*) malli on edellisen vastakohta, eli tässä mallissa ei jäädä odottamaan vastausta toiselta osapuolelta. Vaikka vastaus suoritettavana olevaan kutsuun ei tule juuri synkronista kutsua nopeammin, on mallin etuna se, että odottelun aikana voidaan suorittaa jotain muuta tehtävää. Varjopuolena ovat kuitenkin suunnittelun ja toteutuksen monimutkaistuminen. Erityisesti on kiinnitettävä huomiota siihen missä tilassa sovellus voi mahdollisesti olla vastauksen saapuessa. Kun kyselyjä lähetetään useille sovelluksille yhtä aikaa, voivat vastaukset kyselyihin tulla eri järjestyksessä kuin ne on lähetetty. [Lin01]

3.2.3 Yhteydellinen ja yhteydetön

Yhteydellisessä (*connection oriented*) mallissa jokaista tiedon välitystä varten luodaan oma yhteys. Yhteys katkaistaan vasta sitten, kun kaikki on saatu tehtyä. Malli toteutetaan yleensä käyttäen synkronisia kutsuja, mutta se voidaan toteuttaa myös asynkronisilla kutsuilla. [Lin99]

Yhteydettömässä (*connectionless*) mallissa tieto vain välitetään toiselle sovellukselle. Tietoa voidaan lähettää molempiin suuntiin yhtä aikaa, mutta tiedon perille saapuminen ei ole tässä mallissa turvattua. Sen varmistamiseen täytyy käyttää kuittauksia. [Lin99]

3.2.4 Suorat yhteydet ja jonotusyhteydet

Suoran yhteyden (*direct communications*) mallissa väliohjelmistokerros hyväksyy viestin kutsuvalta ohjelmalta ja välittää sen suoraan kohdeohjelmalle. Suorat yhteydet ovat yleensä synkronisia. Yleisimmät etäkutsuväliohjelmistot (*RPC*) käyttävät suoria yhteyksiä. [Lin00b]

Jonotusyhteysmalli (*queued communications*) toteutetaan yleensä jonotusmanagerin, ohjelman joka pitää yllä viestejä, avulla. Kutsuva ohjelma lähettää viestin jonotusmanagerille, joka laittaa sen jonoon odottamaan. Viesti lähetetään eteenpäin managerin parhaaksi katsomalla ajanhetkellä, ottaen huomioon aikarajat. Useimmat sanomapohjaiset väliohjelmistot käyttävät jonotusyhteyksiä. Jonotusyhteyksien asynkronisuuden ansiosta niillä on etuna suoriin yhteyksiin nähden se, ettei vastaanottavan sovelluksen tarvitse olla vielä aktiivinen viestin lähetyshetkellä. [Lin00b]

3.2.5 Julkaise ja tilaa

Julkaise ja tilaa (*publish and subscribe*) -mallissa julkaisijasovellus lähettää jonkin teeman (*topic*), josta se haluaa jakaa tietoa tätä mallia varten tehdylle väliohjelmistolle. Seuraavaksi väliohjelmisto lähettää kaikille siihen liittyneille tiedon otsikosta. Tämän jälkeen kaikki teemasta kiinnostuneet tilaajasovellukset voivat liittyä teemaan. Teemaan liittymisen jälkeen kaikki tilaajasovellukset saavat kaikki julkaisijasovelluksen lähettämät viestit. Tässä mallissa julkaisijasovelluksien ei tarvitse tietää mitään tilaajasovelluksista. Esimerkkinä tästä mallista voisi mainita Javan tapahtumakuuntelijat ja –käsittelijät, jotka toimivat saman periaatteen mukaisesti. [Lin00b]

3.2.6 Vaadi vastaus

Vaadi vastaus (*request response*) -mallissa kaksi tai useampia sovelluksia on liittyneenä johonkin väliohjelmistoon. Tällainen kyseistä väliohjelmistoa käyttävä sovellus voi vaatia vastausta toiselta samaan väliohjelmistoon liittyneeltä sovellukselta. Tämän jälkeen vaatimukseen kohteena ollut sovellus vastaa viestiin väliohjelmiston kautta. [Lin00b]

3.2.7 Lähetä ja unohda

Lähetä ja unohda (*fire and forget*) -mallissa sovellus voi lähettää viestin ja sen jälkeen unohtaa sen, ilman että sen tarvitsee huolehtia siitä kuka viestin ottaa vastaan vai ottaako kukaan. Mallin tarkoituksena on antaa sovelluksille mahdollisuus lähettää viestejä monille vastaanottajille sekä mahdollisuus suorittaa kevyesti joitain vähemmän tärkeitä kyselyjä muiden sovelluksien tiloista. [Lin00b]

3.2.8 Keskustelu

Keskustelu (*conversational mode*) -mallin mukaisen väliohjelmiston suurin etu on se, että sen avulla voidaan toteuttaa monimutkaisia sovelluslogiikoita. Väliohjelmisto ylläpitää tilat ja sen avulla voidaan selvitellä tapahtumia. Liikennöinti tässä väliohjelmistomallissa on kaksisuuntaista, jolloin molemmat keskustelun osapuolet voivat lähettää ja vastaanottaa viestejä. Tämä väliohjelmistomalli onkin tärkeä tapahtumapohjaisten ratkaisujen integroinnissa. Välitettävät tietoelementit sisältävät usein erilaisia alitapahtumia ja ne voidaan purkaa käymällä keskusteluja vastaanottajan kanssa. [Lin99, Lin00b]

4 Väliohjelmistotyypit

Tässä luvussa esitellään väliohjelmistojen teknistä puolta käymällä läpi erilaisten integrointiratkaisujen toteutusteknologioita. Luvussa käydään kukin teknologia yleisesti läpi teoriatasolla ja esitellään sen jälkeen siihen liittyviä käytännön toteutuksia.

4.1 Etäkutsut

Etäkutsut (*RPC, Remote Procedure Call*) ovat väliohjelmistojen vanhin tyyppi. Niiden rakenne on myös yksinkertaisin ja ne ovat helpoimpia käyttää. Etäkutsu suoritetaan siis aivan kuin kutsuttaisiin jotain oman sovelluksen sisäistä proseduuria tai funktiota. Mahdollinen näkyvä toiminta ilmenee sovelluksessa, jonka proseduuria tai funktiota on kutsuttu. Lopuksi mahdollinen paluuarvo välittyy kutsuvalle sovellukselle samoin kuin perinteisissä funktionkutsuissa. Etäkutsujen varjopuolena on synkronisesta kommunikoinnista johtuva tehottomuus. Kun ohjelma tekee etäkutsun, se jää odottamaan, että kutsuttu ohjelma on saanut suorituksensa loppuun. Kommunikointiin etäkutsut käyttävät standardoituja rajapintoja sekä protokollia. [Lin01]

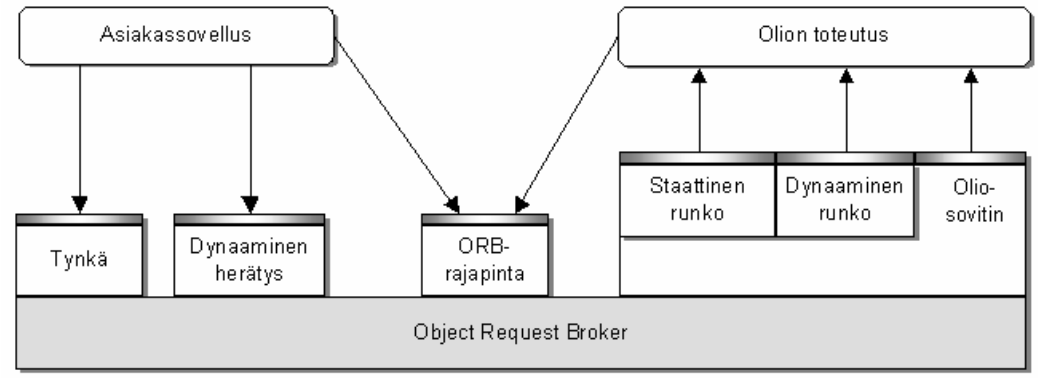
4.2 Hajautetut objektit

Hajautetuilla objekteilla (*distributed objects*) tarkoitetaan tekniikkaa, jossa toisen sovelluksen jotain komponenttia voidaan käyttää aivan kuin se olisi samaa sovellusta. Kutsut voivat tapahtua saman koneen sisällä tai verkon yli. Tekniikaltaan tämä kuulostaa lähes samalta kuin etäkutsut. Hajautetut objektit onkin usein toteutettu etäkutsujen päälle. Tosin nykyään käytetään myös hyvin usein sanomapohjaisia välitystapoja. Erona etäkutsuihin on, että hajautetut objektit tuovat hajautettujen sovellusten kehittäjien käyttöön kaikki oliopohjaisen ohjelmistokehityksen tekniikat, kuten kapseloinnin, perinnän ja monimuotoisuuden. [Lin00b]

4.2.1 CORBA

CORBA (*Common Object Request Broker Architecture*) on OMG:n spesifikaatio hajautetuille objekteille. CORBA on siis tarkkaan ottaen vain spesifikaatio. Tämä on

sinänsä hyvä asia, sillä se tekee CORBA:sta ohjelmointikieliriippumattoman. CORBA 1.0 julkaistiin 1991 ja siitä lähtien sitä on kehitetty kokoajan eteenpäin. Pitkän tuotekehityksen aikana virheitä on saatu korjattua ja tällä hetkellä on käytössä versio 3.0.2. Toteutuksia CORBA:sta löytyy monille eri kielille ja siitä onkin tullut eräs varteenotettavimmista hajautusteknologioista. [OMG02a, OMG02b]



Kuva 4-1: CORBA:n arkkitehtuuri [Mow97 s.55].

CORBA:n arkkitehtuuri on kuvattu tarkemmin kuvassa 4-1. CORBA rakentuu siis ORB:in päälle, joka hoitaa olioiden välityksen verkon yli. Asiakassovelluksen päässä, eli hajautetun sovelluksen kutsuvassa osassa, on tynkä (*Stub*), jota voimme kutsua kuten omaa olioamme. Tämän tyngän ansiosta voimme kutsua muita olioita verkon yli ORB:ia käyttäen. Toisella koneella, olion toteutuspäässä, on runko (*Skeleton*), jonka päälle voimme puolestaan tehdä oman toteutuksemme oliolle. Tämä toteutus on siis edellä kutsutun tynkäolion toteutus, eli siis periaatteessa kutsumme tynkäoliota, jonka koodi ajetaan toisella koneella. Tosin käytännössä asia monimutkaistuu hieman. Jotta CORBA saadaan ohjelmointikieliriippumattomaksi, täytyy meillä olla rajapinnanmäärittelykieli (*IDL*), jolla olioiden, tyngän ja rungon väliset rajapinnat on kuvattava. Lisäksi täytyy ottaa huomioon se, saavatko rajapinnat muuttua ajonaikana. CORBA:ssa on myös mahdollisuus käyttää dynaamista herätystä (*Dynamic Invocation*), jolloin oliota kutsutaan yleistä rajapintaa käyttäen. Tämän avulla voimme toteuttaa sovelluksia, vaikka emme tietäisikään hajautettujen olioiden ajonaikaisia rajapintoja. [Mow97, Ben95, OMG02a, OMG02b]

4.2.2 COM ja DCOM

COM (*Component Object Model*) on Microsoftin kehittämä teknologia komponenttien väliseen vuorovaikutukseen. Se määrittelee yleiset rajapinnat komponenttien välille, mikä puolestaan mahdollistaa komponenttien jakelun muille ja myös muiden tekemien komponenttien käyttämisen omassa sovelluksessa. Komponenttien vaihdot voidaan tehdä jopa ilman koodien muuttamista. COM:lla on kaksi periaatteellista toimintatapaa: prosessin sisäinen tai prosessien välinen. Tässä tapauksessa prosessit tarkoittavat yhden koneen sisällä käynnissä olevia erillisiä prosesseja. Prosessin sisäinen toiminta tarkoittaa, että jonkin komponentin päälle tehdään COM-rajapinta. Tämän avulla sovelluksesta saadaan muokattavampi. Toinen toimintatapa, prosessien välinen, tarkoittaa yleensä yleisesti hyväksi havaittujen komponentien ulkoistamista DLL-kirjastoiksi. DLL-kirjastot ovat Windowsissa toimivia funktio- ja proseduurikokoelmia. Näitä DLL-kirjaston sisältämiä funtioita tai proseduureja voivat kutsua kaikki samalle koneelle asennetut sovellukset. Tämän ansiosta niitä ei tarvitse tehdä jokaiselle sovellukselle erikseen ja näin voidaan myös hyötykäyttää olemassa olevia ja yleisessä levityksessä olevia DLL-kirjastoja. [Lin01, Tho97]

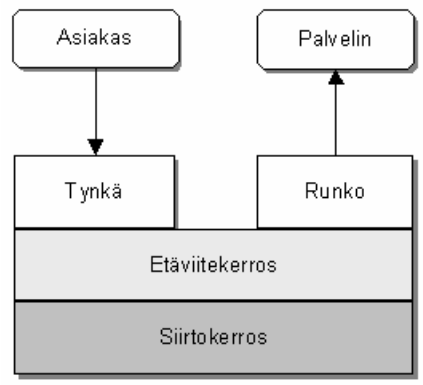
Microsoftin varsinainen ratkaisu hajautettuihin objekteihin on DCOM (*Distributed component object model*), joka kehitettiin ja julkaistiin COM-teknologian jälkeen. DCOM laajentaa COM:n toimintaa mahdollistamalla tietokoneiden välisen hajauttamisen. DCOM siis paketoi COM:n objektit siten, että ne soveltuvat lähetettäväksi yleisen verkkoliikenteen sekaan. Kuten CORBA:n niin myös DCOM:n objektien rajapinnat määritellään käyttäen IDL-kieltä. Vaikka DCOM:lle löytyy toteutuksia muillekin alustoille kuin Windowsille, käytetään DCOM:a yleensä vain Windows-ympäristöön toteutettavissa projekteissa. DCOM:n vahvuuksia ovat:

- Jaettu muistinhallinta useiden objektien kesken.
- Tapa jolla verkkoliikennöinti piilotetaan ja hoidetaan toimivasti.
- Mahdollisuus luoda ja tuhota objekteja aina tarvittaessa.
- Monipuoliset virhe- ja tilaviestit sovellustenkehittäjille.

[Mic03, Tho97]

4.2.3 Java RMI

Java RMI (*Remote Method Invocation*) on puhtaasti Java-kieltä varten suunniteltu olioiden hajautustekniikka. RMI pyrittiin suunnittelemaan heti alusta alkaen mahdollisimman läpinäkyväksi. Tällä tarkoitetaan sitä, että ohjelmoijan tarvitsee ottaa mahdollisimman vähän kantaa RMI:n alla olevaan tietoliikenneverkkoon. Myös helppokäyttöisyys on otettu huomioon. [Nis00]



Kuva 4-2: RMI:n arkkitehtuuri [Nis00].

Java RMI rakentuu asiakaspäässä toimivasta tyngästä (*Stub*) ja palvelinpäässä toimivasta rungosta (S*keleton*). Näiden toiminta on vastaava kuin CORBA:n tapauksessa eli ne muodostavat perustan asiakas- ja palvelinpäissä toimiville olioille. Tyngän ja rungon alla toimii etäviitekerros (*Remote Reference Layer*), joka tulkkaa tyngän ja rungon tiedot edelleen siirtokerrokselle (Transport Layer). Lopuksi siirtokerros hoitaa varsinaisen siirron verkon yli. [Nis00]

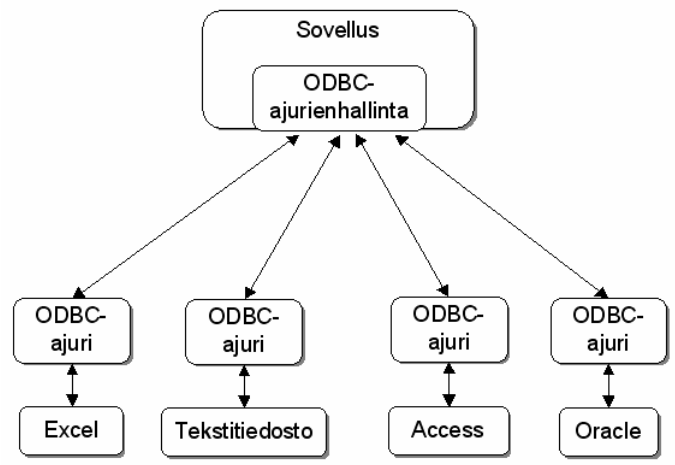
Suurin ero CORBA:an ja DCOM:iin nähden on se, että RMI:ssä luokat kuvataan Javan omia rajapintoja käyttäen eikä erillistä rajapinnanmäärittelykieltä tarvita. Javan ominaisuuksista myös automaattinen roskienkeruu on saatu mukaan toteutukseen. Näistä ominaisuuksista on sekä hyötyä että haittaa. Esimerkiksi automaattinen roskienkeruu helpottaa ohjelmoijan työtä ja vähentää huolimattomuudesta johtuvia muistivuotoja. Huonona puolena on, että roskienkeruun ajankohtaa ei voi itse määrätä, joka puolestaan voi aiheuttaa ylimääräistä kuormaa sovellukselle juuri sinä ajanhetkenä jolloin sitä vähiten kaivattaisiin. RMI:n käyttö on yksinkertaista eikä vaadi paljoa uuden opettelua Java-

kokemusta omaavalle, mutta toisaalta RMI:n käyttö pakottaa sovellusten toteutuskieleksi Javan. [Nis00]

4.3 Tietokantaväliohjelmistot

Tietokantaväliohjelmistot (*database-oriented middleware*) helpottavat tiedon välittämistä ohjelman ja tietokannan välillä tai eri tietokantojen välillä. Ne voivat keskustella tietokantojen kanssa standardoituja komentorajapintoja (*CLI, Call-level Interface*) käyttäen. Näitä komentorajapintoja ovat esimerkiksi JDBC, ODBC sekä OLE DB. Toinen keino keskustella on käyttää tietokannan alkuperäistä väliohjelmistoa. Nämä tietokantakohtaiset väliohjelmistot tarjoavat tietokannan käyttäjälle kaikki erikoisfunktiot ja ovat usein nopeampia kuin yleiset komentorajapinnat. Huonona puolena on tietenkin se, että ne toimivat vain yhden valmistajan tietokannoissa. Lisäksi jopa saman valmistajan eri tietokantaversioidenkin välillä voi kommentorajapinnoissa olla eroja. [Gia01]

4.3.1 ODBC

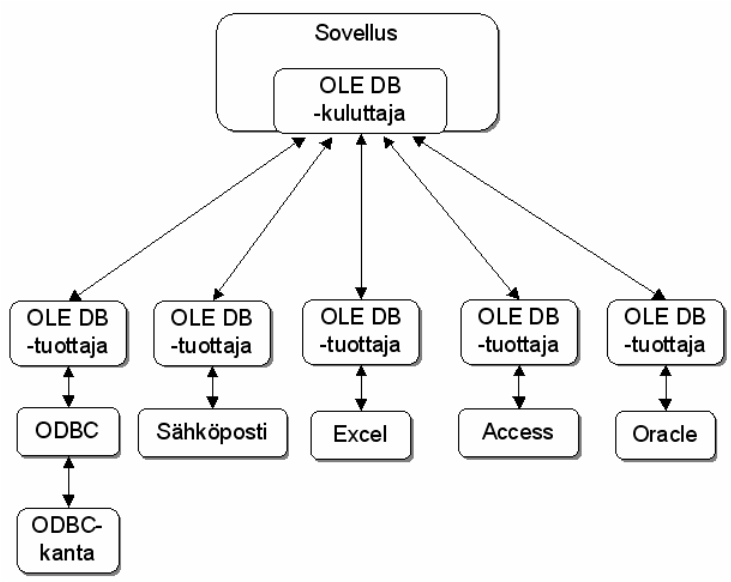


Kuva 4-3: ODBC-komentorajapinnan arkkitehtuuri [Mic03].

ODBC (*Open Database Connectivity*) on Microsoftin julkaisema komentorajapinta tietokantojen käyttöön. ODBC on kehitetty Open Group:n [Ope95] ja ISO/IEC:n [ISO95] standardien pohjalta ja se käyttää SQL:ää kyselykielenään. ODBC on pyritty tekemään sovelluksenkehittäjän näkökulmasta mahdollisimman helppokäyttöiseksi. ODBC toimii

käytännössä siten, että se sisältää ajurit joka tietolähteelle kuvan 4-3 mukaisesti. Nämä ajurit sijaitsevat samassa paikassa tietolähteen kanssa ja ovat ennalta asennettu käyttämään tätä tietolähdettä. Tästä on se hyöty, että ajureita ei tarvitse kääntää sovellusta käännettäessä. Koska kaikkien ajurien rajapinnat sovellukseen päin ovat samat, voidaan ajureita ja sitä kautta myös tietolähteitä vaihtaa tekemättä mitään muutoksia itse sovellukseen. Sovelluksen päässä on ajurienhallintakomponentti (*driver management*), joka hoitaa yhteyden yksittäisiin ODBC-ajureihin. Sovelluskehittäjän työksi jää ainoastaan ajurienhallitsijan liittäminen valmisteilla olevaan sovellukseen. ODBC:n käyttämien tietolähteiden ei välttämättä tarvitse olla tietokantoja, vaan ne voivat olla vaikka Excel- tai tekstitiedostoja. Ajurit osaavat tulkita sovelluksien tekemät SQL-kyselyt siten, että tieto luetaan halutuista tiedoston kohdista. Tietokantojen tapauksessa SQL-kyselyt tietenkin vain välitetään eteenpäin. [Mic03]

4.3.2 OLE DB



Kuva 4-4: OLE DB -komentorajapinnan arkkitehtuuri [Mic03].

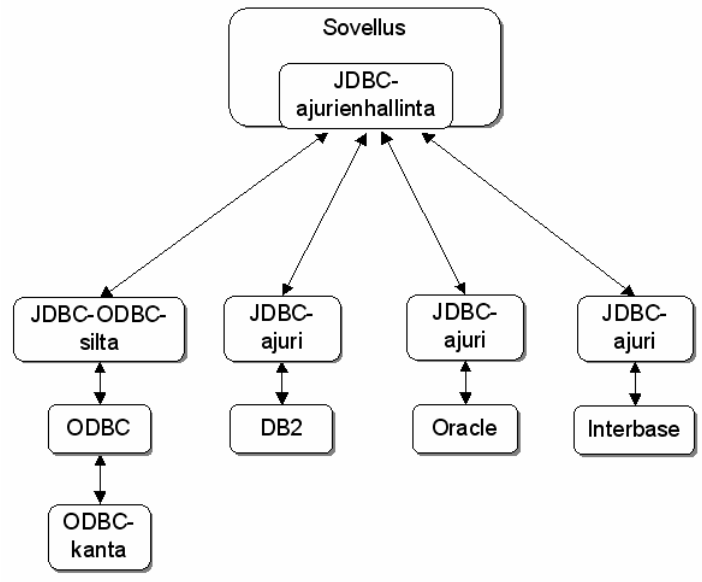
OLE DB (*Object Linking and Embedding to Database*) on myös Microsoftin suunnittelema tekniikka tiedon hakemiseen ja se luotiin laajentamaan ODBC:n toiminnallisuutta. OLE DB on oliomainen lähestymistapa tietolähteisiin liittymiseen.

Tiedon kulkemisen mahdollistavat komponentit ovat tiedontuottaja ja tiedonkuluttaja. [Mic03]

Tiedontuottaja (*dataprovider*) on se komponentti, joka kätkee alleen varsinaisen tietolähteen. Näitä tietolähteitä voivat olla kaikki ne, jotka tukevat COM-rajapintaa. Tämän lisäksi OLE DB:ssä on myös tuki ODBC-rajapinnalle. Tämä mahdollistaa tiedon hakemisen myös vanhemmilta ODBC:tä varten suunnitelluilta sovelluksilta. Tiedontuottajakomponentit suunnitellaan aina jotain yhtä tiettyä tietolähdetyyppiä varten. Lisäksi yksi tiedontuottajakomponentti asetetaan toimimaan aina vain yhden tietolähteen kanssa. Tällä tavalla saadaan peitettyä tietolähdekohtaiset tiedon lukuun liittyvät eroavaisuudet. [Mic03]

Tiedonkuluttajan (*dataconsumer*) tehtävä on toimia yhteyden muodostajana yksittäisiin tiedontuottajiin. Tiedonkuluttaja liitetään itse valmisteilla olevaan sovellukseen ja se siis helpottaa tiedonhakemista. Kaikki tehdyt haut tietolähteille ja niiltä saadut tulokset kulkevat tämän sovelluksen kautta. [Mic03]

4.3.3 JDBC



Kuva 4-5: JDBC-komentorajapinnan arkkitehtuuri [SUN99].

JDBC (*Java Database Connectivity*) on Sun Microsystemsin vastine ODBC:lle. JDBC:n toiminta on hyvin samankaltainen kuin ODBC:n eli se hakee tietoa erilaisista lähteistä SQL-kyselyihin perustuen. JDBC sisältää ajurienhallintakomponentin, joka liitetään valmisteilla olevaan sovellukseen. JDBC sisältää myös tietolähteen päässä olevan ajurin, joka hoitaa tiedon lukemisen SQL-lauseen perusteella. [SUN99, Ell01]

JDBC-ajurit voidaan luokitella neljään kategoriaan [SUN99]:

1. JDBC-ODBC-silta (*JDBC-ODBC Bridge*), joka tarjoaa JDBC-rajapinnan ODBC-rajapinnan läpi.
2. Tietokantakohtainen API-JDBC-ajuri: ajuri tulkitsee JDBC-kutsut natiiviksi tietokantakutsuksi.
3. Verkko-orientoitunut JDBC-ajuri, joka tulkitsee JDBC-kutsut standardiksi tietokantariippumattomaksi verkkoprotokollaksi, joka sitten tietokantapäässä tulkataan takaisin tarvittavalle tietokanta-API:lle.
4. Tietokantakohtainen verkko-JDBC-ajuri, joka tulkitsee JDBC-kutsut tietylle tietokannalle sopivaan verkkoprotokollamuotoon.

4.4 Tapahtumapohjaiset väliohjelmistot

Tapahtumapohjaiset väliohjelmistot (*transaction-oriented middleware*) koordinoivat tiedon kulkua ja metodien jakamista monien erilaisten lähteiden kanssa. Tapahtumilla (*transactions*) käsitetään tässä tapauksessa toimintoa tai toimintosarjaa, jolla on jokin alku ja loppu. Metodien jakaminen on tällä tekniikalla tehokasta, mutta yksinkertainen tiedon välittäminen on turhan hidasta verrattuna muihin välitystapoihin. Tapahtumapohjaisten väliohjelmistojen etuja ovat laajennettavuus, vikasietoisuus ja arkkitehtuuri, joka keskittää sovellusten käsittelyn. Tapahtumapohjaisiin väliohjelmistoihin lasketaan kuuluvaksi myös tapahtumamonitorit ja sovelluspalvelimet, jotka esitellään myöhemmin tässä luvussa. [Lin01]

4.4.1 ACID-ominaisuudet

Tapahtumapohjaisten väliohjelmistojen halutaan usein täyttävän ACID (*atomicity, consistency, isolation, durability*) -ominaisuudet. Ensimmäinen ominaisuus on tapahtuman

atomisuus (*atomicity*), mikä tarkoittaa, että joko kaikki tapahtuman sisältämät tehtävät suoritetaan tai ei suoriteta mitään niistä. Toinen ominaisuus on yhtenäisyys (*consistency*), mikä osoittaa sen, että väliohjelman sovelluskohteena oleva järjestelmä on sallitussa tilassa ennen ja jälkeen tapahtuman. Sallittu tila voidaan määritellä olevan vaikka perustila, josta kaikki tapahtumat aloittavat ja johon ne päättävät toimintansa. Kolmas ominaisuus eli eristettävyys (*isolation*) tarkoittaa sitä, että tapahtuma toimii muista tapahtumista riippumattomasti. Neljäs ominaisuus eli säilyvyys (*durability*) tarkoittaa sitä, että kun tapahtuma on saatu suoritettua loppuun, säilyvät sen tekemät muutokset tallessa vaikka väliohjelmistossa sattuisi joku virhe. [Ben00, Lin01]

4.4.2 Tapahtumamonitorit

Tapahtumamonitorit (*TPM, Transaction Prosessing Monitors*) ovat oikeastaan ensimmäisen sukupolven sovelluspalvelimia. Tapahtumamonitorit hallitsevat kahden tai useamman sovelluksen välisiä tapahtumia. Jos kesken tapahtuman toimintosarjan tulee virhe, osaavat tapahtumamonitorit palauttaa tilanteen tiedon muokkauksen kannalta siihen tilanteeseen, missä se oli ennen toimintosarjan aloittamista. Toinen tapahtumamonitorien etu on, että niiden avulla voidaan sovellus jakaa helposti pienempiin osiin. Ohjelman osat voidaan edelleen hajauttaa vaikka eri koneille ilman että käyttäjän tarvitsee ottaa sitä huomioon. [Lin00b]

4.4.3 Sovelluspalvelimet

Sovelluspalvelimet (*application servers*) on kehitetty tapahtumamonitoreista erityisesti internetistä tulevia yhteydenottoja silmällä pitäen. Nämä kaksi ovat kuitenkin pikkuhiljaa mukautumassa samaksi asiaksi. Sovelluspalvelimet tarjoavat keskitetyn paikan sovelluslogiikalle sekä myös koordinoivat yhteyksiä eri resursseihin. Sovelluspalvelimia kutsutaan myös joissakin yhteyksissä WWW-sovelluspalveluiksi (*web service*). Esimerkkejä tästä teknologiasta ovat Sun Microsystemsin EJB- ja Microsoftin .NET-arkkitehtuurien mukaiset toteutukset. [Lin00b]

4.5 Sanomapohjaiset väliohjelmistot

Sanomapohjaiset väliohjelmistot (*MOM, Message Oriented Middleware*) välittävät tietyn viestin sovellukselta toiselle ottamatta kantaa sen sisältöön. Koska sanomapohjaiset väliohjelmistot ovat myös tyypillisesti asynkronisia, ovat ne suhteellisen nopeita toiminnassaan verrattuna moniin muihin väliohjelmistoihin. Sanomapohjaiset väliohjelmistot käyttävät joko suoria yhteyksiä tai jonotusyhteyksiä. Nykyään on yhä enemmän siirrytty käyttämään jonotusyhteyksiä. Tähän on syynä se, että vastaanottavan sovelluksen ei tarvitse olla välttämättä käynnissä sanoman lähetyshetkellä. [Rit98]

4.5.1 JMS

JMS (*Java Message Service*) on Sun Microsystemsin ratkaisu sanomapohjaiseen viestien välitykseen. JMS on ollut osa Java-standardia versiosta 1.3 lähtien. Tämä standardiin yhdistäminen on toteutettu monin tavoin [Mat01]:

- J2EE-komponentit voivat lähettää ja vastaanottaa synkronisia viestejä.
- Sanomapohjaiset pavut (*message-driven beans*) voivat ottaa vastaan asynkronisia viestejä.
- JMS-asiakkaat voivat tutkia käynnissä olevan JMS-palvelimen objekteja JNDI:tä käyttäen.
- JMS-viestit voivat olla osallisia transaktioissa JTA:ta käyttäen.

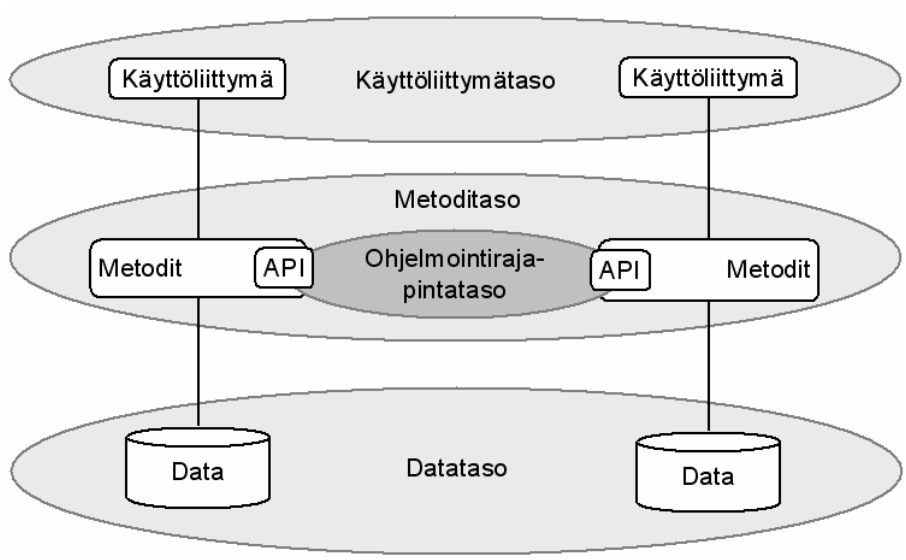
JMS tukee kahta viestien välitystapaa, jotka ovat pisteestä pisteeseen sekä julkaise ja tilaa. Nämä välitystavat on esitelty väliohjelmistojen suunnittelumalleja käsittelevässä luvussa kolme. JMS:n toteutustavoista pisteestä pisteeseen on se, jossa viestien välitys on myös jonotusyhteysmallin mukainen. Siinä viestit lähetetään erilliselle jonotusohjelmistolle, joka lähettää ne edelleen vastaanottajalle annettujen prioriteettien mukaisessa järjestyksessä. Julkaise ja tilaa -välitys toimii pääpiirteissään käyttäen kyseisen mallin perinteistä kaavaa, joka on selitetty väliohjelmistojen suunnittelumalleja käsittelevässä osiossa. Erona voidaan mainita se, että viesteihin on mahdollista lisätä prioriteetti sekä elossaoloaika. [Mat01, Hap02]

4.6 Sanomanvälittäjät

Sanomanvälittäjät (*message brokers*) ovat palvelimia, jotka voivat välittää tietoa kahden tai useamman sovelluksen välillä. Sovellukset voivat sijaita eri alustoilla ja ne voivat lähettää hyvin erimuotoisia viestejä. Sanomanvälittäjät voivat muuttaa viestien muotoa vastaanottajan ymmärtämään muotoon. Näiden toimintojen lisäksi sanomanvälittäjät osaavat älykkään viestien reitityksen (*intelligent routing*). Tämä tarkoittaa sitä, että sanomanvälittäjä osaa lähettää sanoman oikeaan paikkaan sen sisältämän tiedon perusteella. Sanomien välitystä voidaan ohjata myös annettujen sääntöjen perusteella, sillä sanomanvälittäjät voivat sisältää sääntökoneen (*rules engine*). Sääntökoneille voidaan syöttää ennakkoon säännöt, jotka on toteutettu esimerkiksi jollakin skriptikielellä. [Lin00b]

5 Integrointitasot

Tässä luvussa esitellään integrointiratkaisujen jakautumista eri kerroksille, joka määrittelee sen millä tasolla integroitavaan sovellukseen liitytään. Näiden tasojen käyttöä voi soveltaa myös siten, että integroitaviin sovelluksiin liitytään eri tasoilla. David Linthicum [Lin00b] on jakanut tasot kuvan 5-1 mukaisesti neljään eri osaan, jotka ovat datataso, ohjelmointirajapintataso, metoditaso ja käyttöliittymätaso.



Kuva 5-1: Integrointitasot [Lin00b].

5.1 Datataso

Integrointi datatasolla tarkoittaa sekä prosessia että tekniikkaa, joilla tietoa siirretään tietovarastojen välillä. Tätä kutsutaan myös ETL-tyyppiseksi (*Extract-Transform-Load*) tiedonkäsittelyksi: tietoa luetaan yhdestä paikasta, mahdollisesti muokataan ja tämän jälkeen siirretään toiseen paikkaan. Datatason integrointi on yleensä ensimmäinen keino, jolla yrityksen sovellusten integrointia aletaan toteuttaa. Vaikka tämän tyyppistä ratkaisua pidetään yleensä yksinkertaisena ja halpana toteuttaa, voi se laajojen tietovarastojen ja erityyppisten tietokantojen tapauksessa johtaa hyvinkin monimutkaisiin tilanteisiin. [Lin00a, Lin00b]

5.1.1 Tietokannasta tietokantaan

Tietokannasta tietokantaan (*database to database*) -integroinnissa tietoa siirretään nimensä mukaisesti kahden tai useamman tietokannan välillä. Tiedonsiirrot suoritetaan siis ottamatta mitenkään kantaa alkuperäisiin tietokantoja käyttäviin sovelluksiin. Helpointa tietokantojen integrointi on silloin, kun niiden rakenteet ovat samanlaiset. Tällöin tietokantoja käyttävät ohjelmistot voidaan jättää ennalleen. Väliohjelmisto voidaan myös tehdä muuntamaan alkuperäisen tiedon rakennetta tiedonsiirtokohteena olevaan tietokantaan soveltuvaksi. Ongelmakohta tietokannasta tietokantaan -integroinnilla on kuitenkin siinä, että hyvin monet olemassa olevat ohjelmistot ovat vahvasti sidoksissa tietokannassa esitetyn tiedon rakenteeseen. Lähes aina muutokset tietokantaan aiheuttavat muutoksia myös sitä käyttäviin ohjelmistoihin. [Lin00b]

5.1.2 Liitetyt tietokannat

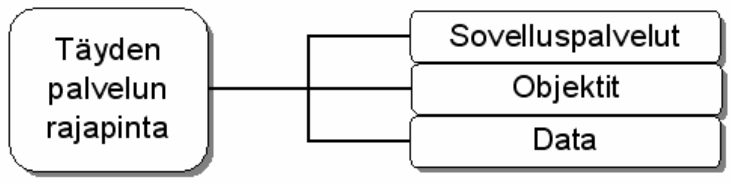
Liitetyt tietokannat (*federated database*) -integrointi tarkoittaa sitä, että käytössä olevat tietokannat liitetään toisiinsa esimerkiksi ODBC:n avulla toteutetun väliohjelmiston avulla. Väliohjelmisto piilottaa muut tietokannat alleen luomalla ikäänkuin virtuaalitietokannan, jota ohjelmat voivat käyttää. Tietoa hakevat sovellukset voivat siis liittyä tähän virtuaalitietokantaan ja hakea tietoa aivan kuten tavallisesta tietokannasta. Virtuaalitietokanta suorittaa tämän jälkeen haun kaikkiin siihen liitettyihin tietokantoihin. Haun tulokset se kokoaa yhdeksi kokonaisvastaukseksi, jonka se palauttaa alkuperäisen kyselyn tehneelle sovellukselle. Tämäkään integrointiratkaisu ei toimi silloin, kun ohjelmistot ovat liian sidoksissa valmistajan tietokantaan. Esimerkiksi eri valmistajien tietokantakohtaiset erikoisfunktiot ja liityntätavat aiheuttavat ongelmia. Erikoisfunktioita ovat esimerkiksi nykyaikaisten tietokantojen SQL-lauseiden joukkoon sijoitettavat komennot. Nämä erikoisfunktiot voivat määrätä esimerkiksi palautettavien tulosten olevan XML-muodossa. [Lin00b]

5.2 Ohjelmointirajapintataso

Ohjelmointirajapintataso (*application interface level*) on kaikkein yleisin integrointitaso. Tälle tasolle löytyy monenlaisia ratkaisuja ja ratkaisuille monenlaisia toteutuksia. Näitä ratkaisuja ovat yleisimmin erilaiset välitystiedostot ja sanomat sekä sovellusrajapinnat (*API*). Välitystiedostojen käyttö on yleistynyt XML:n myötä valtavasti. Tällä tavoitellaan parempaa yhteensopivuutta muiden ohjelmistojen kanssa. API:t tarjoavat puolestaan joustavamman lähestymisen sovellusten integrointiin, mutta ne tuovat mukanaan myös ongelmia. Yhteyden muodostamista vaikeuttavat ohjelmien erilaiset toteutuskielet sekä toisistaan eroavat kommunikointitekniikat, kuten COM, CORBA ja EJB. Suurimman ongelman aiheuttavat kuitenkin sellaiset ohjelmat, joilla ei yksinkertaisesti ole rajapintoja tai joiden rajapinnat vaihtelevat eri versioiden mukaan. Tämän tason etuna datatason integrointiin nähden on se, että tällöin päästään vaikuttamaan myös integroitavien ohjelmien toimintaan. [Lin00a, Lin00b]

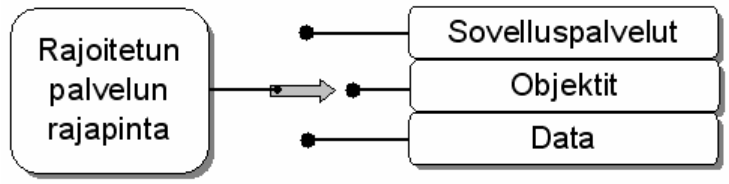
5.2.1 Rajapintatyypit

Integroitavia ohjelmia, jotka tarjoavat jonkinlaisen rajapinnan sovelluskehittäjien käyttöön, kutsutaan paketoiduiksi sovelluksiksi. Niiden tarjoamat rajapinnat voidaan jakaa kolmeen osaan sen mukaan, mille eri kerroksille niiden kautta on mahdollisuus päästä. Kerroksia ovat: sovelluspalvelut (*business services*), objektit (*objects*) ja data. Sovelluspalvelut pitävät sisällään sovelluslogiikan sekä mahdollisuuden lukea sovelluksen sisäistä tietoa. Objektit (*objects*) sisältävät tiettyyn tarkoitukseen läheisesti liittyvät metodit ja tiedon. Data on puhdasta sovelluksen tuottamaa tai käyttämää dataa. Rajapintatyypit on esitetty seuraavissa kappaleissa.



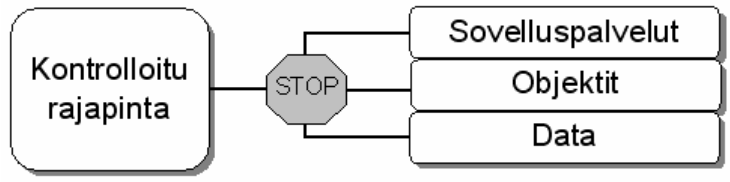
Kuva 5-2: Täyden palvelun rajapinta [Lin00b].

Täyden palvelun rajapinta tarjoaa ohjelmoijalle pääsyn kaikille kerroksille. Tämä onkin joustavin rajapinta vaikuttaa ohjelman tiedonkulkuun. Ohjelmien markkinoijat usein mainostavat, että heidän tuotteensa tarjoaa täyden palvelun rajapinnat käyttöön, vaikka oikeasti toteutukset ovat vajavaisia. Esimerkiksi Oracle on väittänyt, että sillä on täyden palvelun rajapinta. Todellisuudessa käytössä on vain monta rajoitetun palvelun rajapintaa, sillä minkään rajapinnan kautta ei päästä käsiksi kaikkiin sen palveluihin ja sisältämään tietoon. [Lin00b]



Kuva 5-3: Rajoitetun palvelun rajapinta [Lin00b].

Rajoitettu palvelurajapinta on yleisin toteutus mitä ohjelmat tarjoavat. Ohjelmoijalle annetaan käyttöön rajapinta yhdelle tasolle ja sitäkin mahdollisesti rajoitetaan. Tällainen toiminta voi tulla kyseeseen esimerkiksi silloin, kun integroitavien ohjelmien kehittäjät haluavat ensin myydä tuotteensa perusversioita ja tämän jälkeen tehdä uuden laajemman version, jossa on rajapinnat myös muille tasoille.



Kuva 5-4: Kontrolloitu rajapinta [Lin00b].

Kontrolloitu rajapinta tarjoaa vain minimimäärän ominaisuuksia ja funktioita. Tällaisella suljetulla ympäristöllä haetaan turvaa ulkopuolisia ohjelmia vastaan, mutta tarjotaan silti kaikkein välttämättömin toiminnallisuus.

5.3 Metoditaso

Metoditason integroinnissa voidaan käyttää yksinkertaistettuna kahta lähestymistapaa: metodien säilytys jollakin keskuspalvelimella käyttäen esimerkiksi sovelluspalvelimia (*application server*) tai metodien jakaminen sovellusten välillä esimerkiksi hajautettuja objekteja (*distributed objects*) käyttäen. Molemmissa vaihtoehdoissa yritetään löytää sovelluksista jotain samankaltaisuuksia. Tarkoituksena on löytää ja eristää nämä samankaltaiset toiminnot yhteen paikkaan. Tästä johtuen metoditason integrointi vaatii, että päästään käsiksi integroitavien sovellusten metodeihin. Tämä rajoite tietenkin sulkee useita sellaisia sovelluksia pois, joiden metodit eivät ole käytettävissä. Jos onnistutaan kuitenkin saamaan halutut samankaltaiset metodit hallintaan, voi niiden päälle lisätä omaa koodia. Silloin voidaan tehdä omasta koodista sellaista, että kun sitä kutsutaan, niin se kutsuu edelleen kaikkien integroitujen järjestelmien samankaltaisia metodeja. Näin kaikki samankaltaiset metodit tulevat suoritetuiksi yhdellä kertaa. Jokaiseen järjestelmään voidaan liitty niiden omalla toteuskielellä kunhan yhteydenpitokanavaksi valitaan joku yleinen teknologia esimerkiksi CORBA. [Lin00a, Lin00b]

Esimerkkinä metoditason integroinnista voidaan mainita uuden työntekijän tulo taloon. Hänet kuuluisi lisätä palkanlaskentaohjelmistoon, yrityksen sisäiseen kirjastojärjestelmään sekä lomakirjanpitoon. Jos kyseiset järjestelmät on integroitu metoditason ratkaisua käyttäen, voidaan kutsua tehtyä `lisaa_työntekija-`metodia. Tämä metodi kutsuu edelleen kunkin järjestelmän omaa `lisaa_työntekija-`metodia. Näin saadaan yhdellä kutsulla päivitettyä kaikki järjestelmät.

Käytettäessä metoditason integrointia saavutetaan hyvin yhteentoimivia sovelluksia, koska toinen sovellus voi reagoida toisen sovelluksen logiikassa tapahtuviin muutoksiin. Huonona puolena voidaan mainita se, että tämä integrointitapa on yleensä hyvin työläs, koska lähes kaikki yhteen sovellukseen tehdyt muutokset aiheuttavat muutoksia myös moniin muihin paikkoihin integroidussa ympäristössä. [Lin00b]

5.3.1 Metodienvarastointi

Metodienvarastointi (*method warehousing*) tarkoittaa yrityksen järjestelmien toiminnan kannalta olennaisten metodien keskittämistä yhteen paikkaan. Tämä voidaan toteuttaa ottamalla käyttöön palvelinsovellus, johon sovellukset voivat liittyä, ja luomalla metodit sinne. Metodit täytyy myös ohjelmoida uudelleen tätä palvelinsovellusta varten. Palvelinsovelluksen lisäksi tarvitsemme jonkin yhteydenpitotavan metodien ja niitä käyttävien sovellusten välille. Tämä yhteydenpitotapa voi olla esimerkiksi etäkutsut, jotka on esitelty kappaleessa 4.1. Näiden kaikkien toimenpiteiden suorittaminen on työlästä, mutta metodivarasto on hyvin ylläpidettävissä sekä metodit uudelleenkäytettävissä. Voimme asettaa esimerkiksi kaikki sovellukset, jotka muuttavat yrityksen työntekijän tietoja, käyttämään samaa metodia, joka hoitaa tehtävän. Tämän jälkeen on helppo muuttaa vain yhden varastoidun metodin toimintaa ja vaikutukset ilmenevät kaikissa sitä käyttävissä sovelluksissa. Metodienvarastoinnilla saavutetaan paras hyöty silloin, kun sitä käytetään mahdollisimman varhaisessa vaiheessa eli silloin, kun yrityksen järjestelmiä ollaan rakentamassa tai ollaan jonkin suuren muutoksen edessä. Silloin muut sovellukset voidaan suunnitella ja toteuttaa käyttämään tätä varastointiratkaisua.

5.4 Käyttöliittymätaso

Käyttöliittymätason integrointia käytetään yleensä vasta siinä vaiheessa, kun integrointi muilla tasoilla ei onnistu. Tässä tekniikassa sovellusten välinen tiedonkulku välitetään niiden omien käyttöliittymien kautta. Tätä tekniikkaa pidetään yleisesti hyvin epävarmana ja kalliina vaihtoehtona, sillä pienikin muutos käyttöliittymään saattaa aiheuttaa sen, että integrointisovellus ei enää toimi oikein. [Lin00a, Lin00b]

5.4.1 Staattinen tiedonluku

Staattinen tiedonluku tarkoittaa tekniikkaa, jonka avulla tieto luetaan käyttöliittymästä. Käyttöliittymä koostuu ikkunoista ja ikkunat koostuvat erilaisista komponenteista, kuten painikkeista sekä tekstikentistä. Näillä kaikilla komponenteilla on jokin tietty paikka ja tähän staattinen tiedonluku luottaa. Staattisessa tiedonlukemisessa tiedon hakua ja syöttämistä varten annetaan halutun kentän tarkka paikka ruudulla esimerkiksi

koordinaattien perusteella. Tämä johtaa siihen, että kaikki käyttöliittymän muutokset on otettava huomioon myös tiedonluku-komponentissa. [Lin00a, Lin00b]

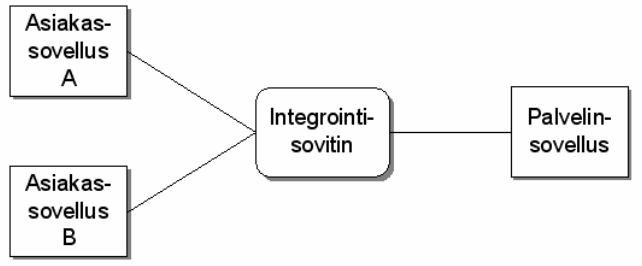
5.4.2 Dynaaminen tiedonluku

Dynaaminen tiedonluku on hieman kehittyneempi versio staattisesta tiedonlukemisesta. Siinä tietoa haetaan ensin jonkin itse tietoon liittyvän tekijän suhteen. Esimerkiksi jos halutaan lukea lomakkeelta työtekijän henkilötunnus, etsitään ensiksi otsikko henkilötunnus ja sen jälkeen luetaan tieto tämän alla olevasta tekstikentästä. Kun tiedonlukeminen tehdään näin, niin henkilötunnus-kentän siirtäminenkään ei aiheuta ongelmia, kunhan se pysyy samassa suhteessa otsikkoonsa nähden. [Lin00a, Lin00b]

6 Integrointiarkkitehtuurit

Tässä luvussa tuodaan esille erilaisia integrointiarkkitehtuureja. Arkkitehtuurit on esitelty yleisellä tasolla ja niiden on tarkoitus toimia ohjeena integrointiratkaisuja varten. Arkkitehtuurit kuvaavat järjestelmän loogista rakennetta, mutta eivät välttämättä toteutettavaa rakennetta. Esimerkiksi integrointisovitin voi teknisessä mielessä olla samaa sovellusta palvelinsovelluksen kanssa, mutta loogisessa mielessä erillinen ohjelma.

6.1 Integrointisovitin



Kuva 6-1: Integrointisovittimen arkkitehtuuri [Lut00].

Integrointisovitinarkkitehtuurissa (*integration adapter*) on jokin sovellus, jonka tietoihin haluamme päästä käsiksi muilla sovelluksilla. Tälle palvelinsovellukselle (*server application*) teemme sovittimen, joka liittyy palvelinsovelluksen rajapintaan. Sovitin tarjoaa edelleen rajapinnan palvelinsovellukseen liittyville asiakassovelluksille (*client application*). Integrointisovitin muokkaa rajapintoja siten, että palvelin- ja asiakassovellus ymmärtävät toisiaan. Nämä rajapinnat voivat olla mitä tahansa kommunikointivälineitä, kuten viestejä, metodikutsuja tai vaikka XML-tiedostoja. Joissakin yhteyksissä integrointisovittimesta puhutaan myös sovelluskääreenä (*application wrapper*). [Lut00]

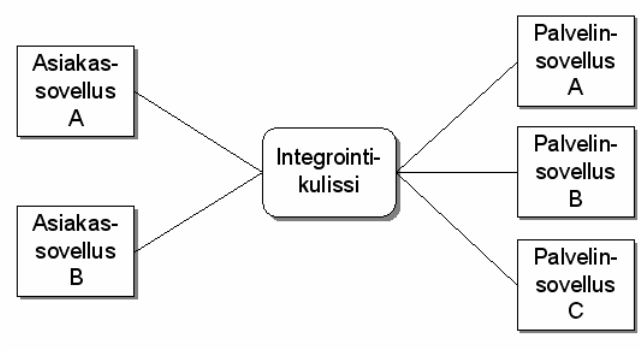
6.2 Integrointilähetti

Kuva 6-2: Integrointilähetin arkkitehtuuri [Lut00].

Integrointilähettiarkkitehtuuria (*integration messenger*) käytetään yleensä silloin, kun sovellusten ja niiden toimintalogiikan erottaminen on epäkäytännöllistä tai tarpeetonta. Tämä tarkoittaa sitä, että jokainen sovellus halutaan pitää itsenäisenä yksilönä eli mahdollisimman vähän riippuvaisena toisten sovellusten tiloista. Viestien välitys hoidetaan pitämällä integrointilähetin ja sovellusten välinen kommunikointi mahdollisimman yksinkertaisena. Toisin sanoen sovellusten välillä siirretään ainoastaan toiminnan takaamiseen pakollinen tieto. Integrointilähetin on siis tarkoitus toimia vain kommunikointialustana. [Lut00]

Tämän arkkitehtuurin mukaisessa ratkaisussa itse sovellusten välinen vuorovaikutusmalli (*application interaction model*) tehdään integrointilähetin päälle. Tämä malli sisältää sovellusten välisen kommunikointilogiikan ja se on liitetty itse sovelluksiin jääden myös niiden vastuulle. Näitä kommunikointilogiikoita, jotka esitellään yleensä väliohjelmistojen suunnittelumalleina, ovat esimerkiksi jonotusyhteydet ja julkaise ja tilaa. Nämä sekä muita väliohjelmistoja on esitelty luvussa kolme. [Lut00]

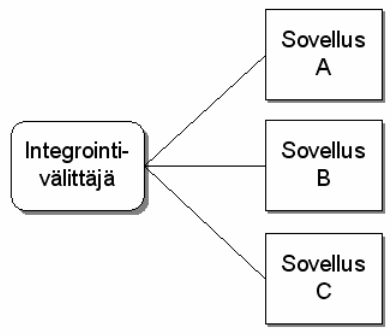
6.3 Integrointikulissi



Kuva 6-3: Integrointikulissin arkkitehtuuri [Lut00].

Integrointikulissiarkkitehtuurin (*integration facade*) ideana on luoda yksi väliohjelmisto, integrointikulissi, joka piilottaa taakseen kaikki palvelinsovellukset. Asiakassovellukset voivat tämän jälkeen käyttää palvelinsovelluksia integrointikulissin kautta. Asiakassovellukset näkevät vain yhden tietolähteen ja tämän rajapinnan. Integrointikulissi osaa muuntaa asiakassovelluksen tekemät pyynnöt palvelinsovelluksien ymmärtämään muotoon ja myös palvelinsovellusten palauttamat tiedot asiakassovellusten ymmärtämään muotoon. [Lut00]

6.4 Integrointivälittäjä

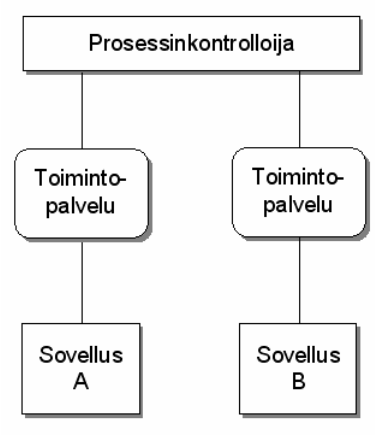


Kuva 6-4: Integrointivälittäjän arkkitehtuuri [Lut00].

Integrointivälittäjäarkkitehtuurissa (*integration mediator*) luodaan yksi piste, jonka sisältämän kommunikointilogiikan kautta kaikki integroitavat sovellukset kommunikoivat. Hyvä puoli tällä arkkitehtuurilla on se, että sovellusten väliset riippuvuudet on minimoitu.

Ylläpito on helppoa, koska pääasiallinen toiminnallisuus on yhdessä paikassa. Myös uudelleenkäytettäviä palveluja on helppo liittää mukaan, koska kommunikointilogiikka on integrointivälittäjässä itsessään mukana. Ero integrointilähettiin on siinä, että integrointivälittäjä tietää minkälaisia sovelluksia siihen on liittyneenä mutta integrointilähetti ei. [Lut00]

6.5 Prosessinautomatisoija



Kuva 6-5: Prosessinautomatisoijan arkkitehtuuri [Lut00].

Prosessinautomatisoija (*process automator*) -arkkitehtuurissa pyritään muodostamaan kokonaisuus, jonka avulla saamme suoritettua halutun tehtävän alusta loppuun. Arkkitehtuurissa pyritään siis luomaan toimintoketju, jonka mukaan annetaan suoritusvuoroja niin ihmisille kuin toimintoketjuun integroitaville sovelluksille. Tätä toimintoketjua ohjaa prosessinkontrolloija (*process controller*). Kaikki yksittäiset toiminnat on kapseloitu toimintopalveluiden (*activity service*) taakse. Jokainen joko ihmisen ja sovelluksen tai sovelluksien välinen vuorovaikutus vastaa yhtä toimintoa. Seuraavassa on kuvattu esimerkki toimintoketjusta tavallisen ostostilanteen avulla:

1. Asiakas kysyy tuotetta.
2. Kauppias syöttää tuotteen tiedot järjestelmän hakutoimintoon.
3. Järjestelmä hakee tiedot ja ilmoittaa varastossa olevien tuotteiden määrän.
4. Asiakas ostaa tuotteen.
5. Kauppias kirjaa oston järjestelmään.

6. Järjestelmä vähentää varastotiedoista tuotteen.
7. Järjestelmä merkitsee oston kirjanpitoon.
8. Järjestelmä tulostaa kuitin.

Prosessi koostuu siis useista eri toiminnoista, jotka prosessin automatisoija-arkkitehtuurissa yhdistetään prosessinkontrolloijaan. Arkkitehtuurin hyviä puolia ovat [Lut00]:

- Hyvin toimivia prosessin automatisointiratkaisuja voidaan toteuttaa kustannustehokkaasti.
- Mahdollistaa hyvän prosessin seurannan ja ongelmien tunnistamisen.
- Prosessiin liittyviä sovelluksia on helppo muuttaa tai jopa vaihtaa.
- Prosessin kuvaus on lähellä asiakkaan näkökulmaa asiasta, joten se helpottaa yhteisymmärryksen löytymistä asiakkaan ja sovelluskehittäjän välillä.
- Kuten integrointikulissin ja integrointivälittäjän tapauksessa, integrointilogiikka on kapseloitu, joten sitä voidaan käyttää uudelleen toisen sovelluksen integroimiseen.

7 Integroinnin teknologiat ja standardit

Tässä luvussa perehdytään yleisesti integroinnissa käytettyihin teknologioihin ja standardeihin. Aluksi tarkastelemme kahta integroinnin mahdollistavaa tekniikkaa: XML:ää, joka on paljon suosiota saanut rakenteinen tietomuoto, sekä Apache Jakarta POI-komponentteja joiden avulle voimme lukea ja kirjoittaa Mirrosoftin Office-tiedostoja. Lopuksi esitetään lyhyt kuvaus eri yritysten sovellusten välisessä integroinnissa käytetyistä standardeista: RosettaNet:stä ja ebXML:stä.

7.1 Rakenteinen tieto ja XML

Tiedon rakenteistaminen ja yhteisten tietomuotojen löytäminen on erittäin tärkeää tiedon välittämisessä. Ilman yhteistä ja tarkkaan määriteltyä tiedonvälitystapaa eivät sovellukset voi keskustella keskenään tai ne toimivat väärin. Esimerkiksi niinkin yksinkertainen tiedon siirtomuoto kuin tekstitiedosto voi aiheuttaa ongelmia. Kun Windows-maailmassa tekstitiedoston rivinvaihtoja merkitään `CR-LF`-merkeillä, niin Unix-maailmassa rivinvaihtoa merkitään pelkällä `LF`-merkillä. Tätä sekoittamassa on vielä MacOS, jossa rivinvaihtoa merkitään puolestaan `CR`-merkillä. Toinen ongelmakohta on esimerkiksi ohjelmointikielissä käytettävät lohkojen erotusmerkinnät. Joissakin kielissä käytetään aaltosulkuja ja toisissa `begin` ja `end` merkintöjä. [Mar00]

Tämäntapaisia tiedon kuvauksen ongelmia varten julkaistiin 1969 GML-kieli (*Generalized Markup Language*). GML oli kieli, jolla voitiin viitata itseensä tai kuvata toisia kieliä. Tämä oli ensimmäinen moderni merkintäkieli. 1986 GML-kieli laajeni SGML-kieleksi (*Standard Generalized Markup Language*), joka oli ilmaisuvoimaltaan todella vahva, mutta samalla mutkikas. SGML-kieltä käytettiin laajalti tiedon tallennukseen ja välittämiseen. Vihdoin 1998 W3C (*World Wide Web Consortium*) julkaisi suosituksensa XML 1.0:n käyttöön. XML-kieli kehitettiin yksinkertaistamaan monimutkaista SGML-kieltä. [Mar00]

7.1.1 XML ja integrointi

XML:ää käytetään yleisesti integroinnissa, jonka ei tarvitse olla reaaliaikaista. Integrointi sijoittuu yleensä datatasolle. Tämän tason kohteita voivat olla esimerkiksi sovellukset, jotka jalostavat tietoa ja asettavat sen tarjolle muiden sovellusten myöhempää käyttöä varten. XML:ää käytetäänkin yleisesti tiedonjakelutyyppinä. Nykyään onkin standardoitu monien liiketoiminta-alueiden tiedonjakeluformaatteja ja näissä XML on usein avaintekijänä. Esimerkkejä standardoinneista ovat RosettaNet ja ebXML (*Electronic Business using eXtensible Markup Language*), jotka esitellään myöhemmin tässä luvussa. [Lea99, Ros03, ebX03]

7.1.2 Syntaksi

Syntaksiltaan XML muistuttaa HTML:ää. XML:ssä käytetään kuitenkin tageja, jotka ovat <>-suluissa olevia merkintöjä, joita käytetään tiedon rakenteen kuvaamiseen. Nämä merkinnät voivat olla itse keksittyjä. HTML:ssä tagit ovat ennalta määrätyn muotoisia ja niitä käytetään kuvaamaan tiedon ulkoasua. XML määrittelee tiettyjä muotosääntöjä, jotka dokumenttien on täytettävä ollakseen niin sanotusti hyvin muodostettuja (*well-formed*). Seuraavassa on lista XML-dokumentin tärkeimmistä muotosäännöistä [Bra00]:

- Jokaisessa XML-dokumentissa on oltava juuritason elementti, eli elementti, jonka sisässä kaikki muut elementit ovat. Juuritason elementtejä voi olla vain yksi.
- Jokaisella elementillä on oltava alku- ja lopputagi.
- Tyhjille tageille on oma erikoismerkintä `<element/>`, joka vastaa merkintää `<element></element>`.
- Elementtien ja attribuuttien nimet ovat case-sensitiivisiä, eli niissä isot ja pienet kirjaimet ovat merkitseviä.
- Attribuuteilla täytyy olla aina arvo ja sen on oltava lainausmerkkien sisällä.
- Elementtien on oltava oikeassa järjestyksessä eli sisempi elementti on suljettava ennen kuin ulompi suljetaan.

7.1.3 DTD

DTD:llä (*document type definition*) eli dokumenttityypin määrittelykielellä kuvataan XML-tiedoston kielioppia. DTD antaa XML-tiedoston sisällölle jonkin muodon. DTD-määrittelyn avulla voidaan siis muodostaa yhdenmuotoisia XML-tiedostoja. Näitä tiedostoja voi sitten edelleen lukea tai luoda jollakin yhteisellä ohjelmalla. Jos XML-tiedosto täyttää sekä XML-syntaksin että DTD-määrittelyn tuomat rajoitteet, voidaan se todeta validiksi (*valid*). Seuraavassa luettelossa on esitelty yleisimpiä DTD:n määrittelemiä osioita XML-dokumentille [Bra00]:

- XML-tiedoston sisältämät elementit
- Elementtien sisältämät tietotyypit
- Elementin attribuutit
- Elementtien sijainnit suhteessa toisiinsa
- Elementtien pakollisuus

7.1.4 Skeema

DTD:n lisäksi on toinen tapa määritellä XML-dokumentin rakenne. Tämä tapa on nimeltään XML-Skeema (XML Schema). XML-Skeema luotiin korjaamaan DTD:n puutteita. XML-Skeema muun muassa yksinkertaistaa DTD:n syntaksia ja antaa paremmat mahdollisuudet tyypinmäärittelyyn. XML-Skeema sisältää myös oliomaisen tavan kirjoittaa määrittelyä. Tämä tarkoittaa sitä, että kerran dokumentissa kirjoitettua määrittelyä voi uudelleenkäyttää jossain toisessa dokumentin kohdassa. [Mar00]

7.1.5 Parseri

Parserin eli jäsentimen tehtävä on lukea tieto XML-tiedostosta sovellukselle. Parserit tarkastavat myös hyvin usein, onko XML-tiedosto hyvin muodostettu ja validi. Oikeamuotoisuuden parseri voi tutkia itse XML-dokumentista, mutta validiuden tarkastamiseen parserit käyttävät DTD- tai skeema-tiedostoja. Parsereita on saatavilla runsaasti ja useille eri kielille. Yleisesti ottaen on olemassa kahden tyyppisiä parsereita: tapahtumiin (*event-driven*) sekä puurakenteeseen (*tree-based*) perustuvia. Tapahtumiin perustuvat parserit käyvät XML-tiedostoa läpi ja antavat tapahtuman aina kun yksi

kokonainen elementti on luettu. Elementin tiedot välitetään tapahtuman mukana. Puurakenteeseen perustuvat ratkaisut muodostavat XML-tiedostosta puurakenteen, jota sovellus voi sitten käyttää. [Mar00]

7.1.6 Xpath

Xpath on XML-tiedostojen sisältämien tietojen hakutekniikka. Tietoa voidaan hakea aivan kuin se olisi käyttöjärjestelmän hakemistopuussa. Esimerkiksi lauseke `/element1/element2` palauttaisi kaikki `element2`:n alla olevat tiedot. Vastaavasti pelkkä `/element1` palauttaisi kaikki sen alla olevat tiedot, joihin kuuluu myös `element2`:n. Tietoa voidaan hakea myös elementtien sisältämien attribuuttien arvojen perusteella, sekä sen mukaan, mitä muita elementtejä kyseinen elementti sisältää. [Cla99a]

7.1.7 XSLT

XSLT (*XSL Transformations*) on kieli, joka on tarkoitettu XML-dokumenttien muuntamiseen toiseen muotoon. XML-dokumentti voidaan muuntaa siis XSLT-tyylitiedoston avulla esimerkiksi WWW-sivuksi. Tämän hyödyllisen ominaisuuden johdosta XSLT on saanut paljon huomiota osakseen sähköisen julkaisutoiminnan parissa. Tiedon julkaisu tapahtuu seuraavasti: kirjoitetaan julkaistava asia XML-tiedostoon ja jokaiselle julkaistavalle formaatille oma XSLT-tyylitiedosto. Ylläpito helpottuu huomattavasti, kun ei tarvitse muuttaa kuin XML-tiedostoa. Tämä XML-tiedosto voidaan sen jälkeen generoida kaikille halutuille formaateille, kuten WWW- tai WAP-sivuksi. [Cla99b]

7.1.8 DOM

DOM (*Document Object Model*) on ohjelmointirajapintamäärittely XML:ää varten. DOM määrittelee toiminnan ohjelmointikieliriippumattomasti. Määritelmässä kerrotaan miten ohjelma lukee XML-dokumentin ja muodostaa siitä loogisen rakenteen. Tämä tehdään lukemalla yhdellä kertaa koko XML-dokumentti. Vasta tämän jälkeen muodostettua rakennetta voi alkaa käyttämään. DOM:n avulla sovellus voi siis helposti selata XML-dokumentin rakennetta sekä lisätä, muokata ja poistaa elementtejä ja tietosisältöä. DOM:n

muodostamaa oliohierarkiaa on helppo käyttää, mutta tämä malli ei sovi kaikille XML-dokumenteille. Hyvin suurien XML-dokumenttien tapauksessa muistinkulutus voi kasvaa liian suureksi johtuen DOM:n ominaisuudesta ladata koko dokumentti kerralla muistiin. [App98]

7.1.9 SAX

SAX (*The Simple API for XML*) on ohjelmointirajapintamäärittely XML:n käyttöön aivan kuten DOM. Suurimpana erona on se, että SAX ei lataa XML-dokumenttia kokonaan muistiin, vaan käy sitä läpi kohta kerrallaan. Tiedot välitetään SAX:ia käyttävälle sovellukselle tapahtumien (*event*) avulla. Jokaisen elementin alku- ja loppukohdassa aiheutetaan tapahtuma, joka kertoo elementin nimen ja tiedon siitä, kumpi kohta oli kyseessä. Myös jokaisen varsinaisen tiedon kohdalla aiheutetaan tapahtuma. SAX kuluttaa huomattavasti vähemmän muistia kuin DOM, koska dokumentista käsitellään aina vain pieni osa kerrallaan. Huonona puolena SAX:lla on DOM:iin verrattuna se, että monipuoliset haut ovat hankalia toteuttaa tai niistä tulee hitaita. Tämän aiheuttavat mahdolliset ristiviittaukset XML-dokumentin sisällä.

7.2 Apache Jakarta POI

Apache Jakarta POI on vapaan lähdekoodin projekti Microsoftin Office -tiedostojen lukuun. Tarkemmin sanottuna se on Apache-ohjelmistosäätiön (*The Apache Software Foundation*) Java-ohjelmia sisällään pitävän Jakarta-projektin POI-alaprojekti. POI tulee sanoista *Poor Obfuscation Implementation*, joilla viitataan Office-tiedostojen huonoon ja sekavaan toteutukseen. Humorisesta nimestään huolimatta POI-projekti on varteenotettava vaihtoehto Office-tiedostojen lukuun. Hyvinä puolina sillä on ilmaisuus sekä puhdas Javan käyttö toteutuskielenä ja sen seurauksena käyttöjärjestelmäriippumattomuus. POI-projektin komponentteja käytetään muun muassa Open Office:ssa. Tämä jo nykyään laajalle levinnyt ilmainen ja useissa käyttöjärjestelmissä toimiva toimistosovelluspaketti käyttää POI-projektin komponentteja yhteensopivuuden takaamiseksi Microsoftin Office-tuotteisiin. POI-projektin komponenteissa on vielä kuitenkin hieman virheitä, eikä kaikkia Microsoftin Office-tiedostojen ominaisuuksia ole vielä toteutettu. POI-projekti on kuitenkin jatkuvan

kehityksen alla ja uusia päivityksiä tulee tiheään. Tämän gradun käytännön osuus on toteutettu POI:n versiolla 2.0 pre release 3. Seuraavissa kappaleissa esitellään POI-projektin komponentteja, joilla kullakin on oma tärkeä tehtävänsä. [Apa03]

7.2.1 POIFS

POIFS (*POI Filesystem*) on tehty vastaamaan Microsoftin OLE 2 Compound -formaattia. OLE 2 toimii yhteisenä pohjana kaikille Microsoftin Office-tiedostoille. Tämän pohjan päälle on sitten rakennettu eri sovellusten omat tiedostomuodot. OLE 2:hta voidaan ajatella ikään kuin zip-pakettina, joka pitää ensiksi purkaa päästäksemme kiinni itse sisällä olevaan dataan. POIFS tarjoaa siis lähtökohdan POI-projektin muille komponenteille. Itse POIFS-komponenttia käytetään harvemmin suoraan, mutta se sisältyy projektin muihin komponentteihin. [Apa03]

7.2.2 HSSF

HSSF (*Horrible Spreadsheet Format*) on POI:n komponentti Excel-tiedostojen käsittelyyn. Se tarjoaa menetelmät tiedostojen lukuun, kirjoittamiseen ja muokkaukseen. Komponentin avulla voidaan muotoilla Excelin kenttien erilaisia ulkoasullisia parametreja sekä eri tietotyyppejä. Kaavojen käsittely on tosin tämän tutkielman kirjoitushetkellä hieman keskeneräinen. Komponentti toimii kaikkien tiedostojen kanssa Excel 97 -versiosta eteenpäin. Tämän tutkielman kokeellisessa osuudessa Excel-tiedostojen lukuun käytetään juuri tätä komponenttia. [Apa03]

7.2.3 HDF

HDF (*Horrible Document Format*) on Java-toteutus Word-dokumenttien lukemiseen. Tämä POI:n komponentti on vielä kehityksensä alkupuolella, mutta edistyy jatkuvasti. Makrot ja erilaiset automaattiset kentät on jätetty vielä taka-alalle ja keskitytty varsinaisen tekstin ja sen muotoilujen lukemiseen. [Apa03]

7.2.4 HPSF

HPSF (*Horrible Property Set Format*) -komponentti on tarkoitettu Microsoftin Office-dokumentteja kuvailevien tietojen (*properties*) lukemiseen ja kirjoittamiseen. Näitä tietoja ovat esimerkiksi dokumentin otsikko, tekijä sekä viimeisin muokkauskerta. Myös tämä komponentti perustuu Microsoftin OLE 2 -formaattiin. Ja koska Microsoft käyttää tätä formaattia myös monissa muissa tiedostoissaan, on HPSF-komponentilla mahdollista muokata myös näiden tiedostojen tietoja. [Apa03]

7.3 RosettaNet ja ebXML

RosettaNet ja ebXML ovat yritysten välisen tiedonvälityksen standardeja, joista molemmat ovat keskittyneet lähinnä elektroniseen kaupankäyntiin. Näiden XML:ään perustuvien standardointien on tarkoituksena luoda yhteinen muoto välitettävälle tiedolle molempien kommunikointiosapuolten kesken. Tämä tarkoittaa esimerkiksi sitä, että myytäessä jokin tuote toiselle yritykselle, voidaan sen sähköiset tiedot siirtää ostaja yrityksen tietojärjestelmiin vain napin painalluksella. [Ros03, ebX03]

RosettaNet on voittoa tavoittelematon organisaatio, joka on levinnyt laajalle erityisesti elektroniikkateollisuudessa saamansa suosion avulla. RosettaNet tukee elektroniikan komponenttien ja puolijohdevalmistuksen lisäksi myös informaatio- sekä mobiiliteknologiaa. RosettaNet:n standardien mukaan jokaiselle yritykselle, samaten kuin jokaiselle yrityksen tuotteelle, annetaan yksilöivä koodinumero. Tuotteilla on myös yhteisiä parametreja joiden perusteella niiden vertailu on helppoa. [Ros03]

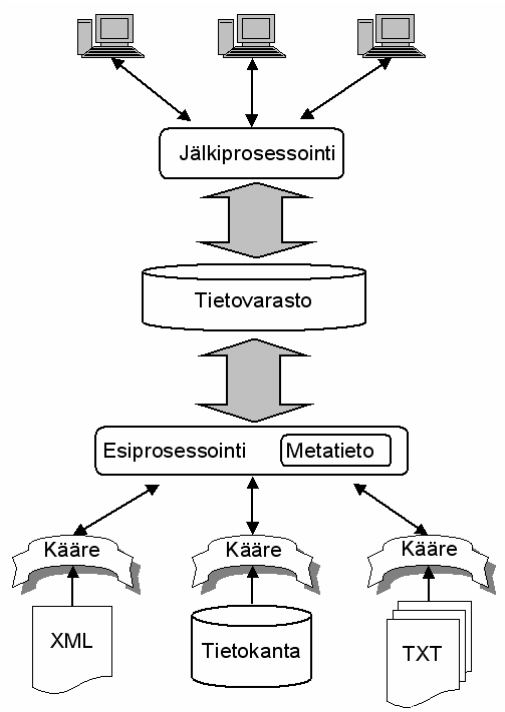
ebXML on kokoelma standardeja, joilla mahdollistetaan kaiken kokoisten yrityksien maailman laajuinen kaupankäynti. ebXML on luotu nimenomaan mahdollisimman yleiskäyttöiseksi, ottaen huomioon myös vanhat järjestelmät. ebXML:n perustana ovat toimineet avoimet internetstandardit kuten XML, HTTP, SOAP, HTTPS. Yritysten välinen vuorovaikutus perustuu ebXML:ssä rekistereihin ja tietohakemistoihin. Yritykset luovat itselleen ebXML:n rekisteriin tietohakemiston ja tallentavat sinne omat ja tuotteidensa tiedot. Yritykset voivat tämän jälkeen hakea liikekumppaneita ja heidän tuotteitaan näiden tietojen perusteella. [ebX03]

Organisaation järjestelmien integroinnin ja yleensä kehityksen kannalta nämä edellä mainitut standardit on hyvä ottaa huomioon. Vaikka nämä standardit ovat suunnattuja yritysten väliseen kaupankäyntiin, on hyvä idea toimia niiden mukaan myös yrityksen sisällä. Tuotteista kannattaa merkitä ylös yrityksen sisäisiin järjestelmiin jommankumman standardin mukaiset tiedot. Näin ollaan myös valmistauduttu mahdolliseen tiedon ulkoistamiseen tulevaisuudessa. Näiden standardien seuraaminen voi myös tuoda uusia ideoita siihen mitä tietoja yritysten järjestelmissä kannattaa pitään yllä.

8 Tiedonvarastointi

Tiedonvarastointiratkaisut ja tietovarastot ovat hyvin usein integroinnin kohteena olevia järjestelmiä. Tässä luvussa esitellään tietovarastojen yleistä rakennetta ja niihin liittyviä käsitteitä. Luvun lopuksi esitellään vielä tiedonvarastoinnin suhdetta sovellusten integrointiin ja mahdollisia liityntätapoja järjestelmään. [Run00]

8.1 Mitä on tiedonvarastointi?



Kuva 8-1: Tietovarastojen arkkitehtuuri [Run00].

Tiedonvarastointi (*data warehousing*) on enemmänkin tiedonhakuprosessi kuin tuote. Tämä prosessimalli siis kuvaa sitä, miten tietoa voidaan hakea ja käsitellä useista erityyppisistä lähteistä. Tarkemmin ottaen tiedonvarastointiprosessi jakaantuu vielä kolmeen osaan: esiprosessointi (*beck-end processing*), tiedon tallennus ja jälkiprosessointi (*front-end processing*). [Gar98, Jon98]

Esiprosessointi on tiedon haun ensimmäinen vaihe. Siinä tieto luetaan kääreiden (*wrapper*) avulla esimerkiksi tietokannasta, XML- tai Excel-tiedostoista. Lukemisen jälkeen tieto

voidaan muokata tietovaraston edellyttämään muotoon. Esiprosessointi hoidetaan kullekin tietolähdetyypille erikseen suunnitellulla väliohjelmistolla.

Tiedonvarastoinnin seuraava vaihe on kaiken esiprosessoinnista saadun tiedon tallennus yhteen paikkaan. Tätä paikkaa kutsutaan tietovarastoksi (*data warehouse*). Tietovarastot sisältävät siis aina viimeisimmät esiprosessoidut tiedot. Tietovarasto tarjoaa edelleen rajapinnan, jonka avulla tietoa voidaan sieltä hakea. Tässä mielessä tietovarasto on kuin tietokanta, eli se tarjoaa tietyn paikan ja rajapinnan mistä tietoa voidaan hakea. Toisaalta tietovarasto eroaa tietokannasta siltä osin, että tietovaraston sisäinen tieto voi muuttua. Tämä muutos aiheutuu siis ilman tietovarastoa varsinaisesti käyttävien sovellusten osallistumista asiaan. Tietovaraston päivittyminen lähtee liikkeelle yhden tietolähteen muuttuessa, jolloin se esiprosessoidaan uudestaan ja lähetetään edelleen tietovarastoon.

Jälkiprosessointi on sitä, että haemme tarvittavat tiedot tietovarastosta ja mahdollisesti käsittelemme niitä tiedonlouhinta- tai päättelymenetelmillä. Kun tämä on tehty, siirrämme tiedot niitä pyytävälle sovellukselle. Jälkiprosessointi suoritetaan sellaisilla väliohjelmistoilla, jotka osaavat käyttää tietovaraston rajapintaa ja toisaalta niitä voi kutsua jokin tietty sovellus.

8.2 Metatieto

Metatieto on tietoa tiedosta. Metatieto kuvaa siis jonkin järjestelmän sisältämää tietoa siten, että voimme tämän metatiedon avulla löytää varsinaisen tiedon huomattavasti helpommin. Metatieto on esimerkiksi sitä, että jostain tietystä kirjasta on tiedot sen tekijästä, julkaisuvuodesta, kustantajasta ja sijainnista sekä muista olennaisista tiedoista tallennettuna esimerkiksi kirjaston tietokantaan. Nyt voimme hakea kirjan sen tietokannassa olevien tietojen eli metatietojen perusteella. Lopputuloksena saamme tiedon kirjan sijainnista ja voimme hakea koko kirjan eli varsinaisen tiedon. [Gar98, Jon98]

Tiedonvarastoinnissa metatieto tarkoittaa tietoa jostain käytössä olevasta objektista. Näitä objekteja voivat olla tietokantataulu, sarake, kysely, raportti, toimintasääntö tai muunnosalgoritmi. Metatiedon avulla siis löydämme varsinaisen tiedon tietovarastosta. Ja kuten kirjastoesimerkissäkin täytyy kirjaston tietokanta pitää ajan tasalla lisättyjen ja

poistettujen kirjojen suhteen, täytyy myös tietovaraston metatietoja päivittää riittävän usein. [Gar98]

8.3 Tiedonvarastointi ja integrointi

Tiedonvarastointiratkaisuja käytetään monissa yrityksissä ja usein ne myös jossain vaiheessa joutuvat osaksi sovellusten integrointia. Vaikka tiedonvarastointia ei varsinaisesti lasketa sovellusten integroinnin piiriin, sillä on monia hyvin samantapaisia piirteitä ja käytettyjä tekniikoita. Samoin kuin sovellusten integroinnissa käytetään väliohjelmistoja, on tiedonvarastoinnissa kerroksia, jotka toimivat väliohjelmistojen tapaan. Jokaisella kerroksella on oma tehtävänsä ja käytetyt rajapinnat ovat samoja kuin mitä käytetään sovellusten integroinnissa. Näitä rajapintoja ovat esimerkiksi ODBC, JDBC ja OLE DB.

Tiedonvarastointiratkaisut integroinnin kohteena tarjoavat sovellusten integroijalle yleensä hyvät liityntämahdollisuudet. Se, millä tasolla liityntä voidaan toteuttaa, riippuu tiedonvarastointiratkaisun valmistajasta ja käytettyjen tekniikoiden julkisuudesta. Jos kaikki käytetyt tekniikat ovat yleisiä standardoituja ratkaisuja, on liityntään edellisten lukujen perusteella viisi erilaista lähestymiskohtaa.

1. Liitytään tietovarastoon jälkiprosessoinnin jälkeen. Liityntä tapahtuu siis käyttämällä rajapintaa, jonka kautta tietovarasto on alunperin suunniteltu välittämään tietoa. Tämä vaihtoehto on suositeltavin, koska tietovarasto palvelee tällöin parhaiten omassa roolissaan, eikä sen tekemiä tehtäviä tarvitse tehdä uudelleen. Joskus voi kuitenkin tulla vastaan tilanne, jossa tietovaraston tarjoama tietomuoto tai tiedon määrä ei ole haluttu. Tällöin joudutaan turvautumaan muihin vaihtoehtoihin.
2. Liitytään suoraan varsinaiseen tietovarastoon, eli paikkaan, johon tieto on tallennettu. Tämä on yleensä jokin tietokanta, mikä tekee liitynnästä helpon. Tässä vaihtoehdossa kaikki tieto on kyllä saatavilla, mutta jälkiprosessoinnin tekemä tiedon jalostaminen jää pois.

3. Liitytään esiprosessoinnin jälkeen. Tällä tavalla saamme periaatteessa samat tiedot kuin edellisessä kohdassa, mutta liityntätapa on erilainen tai sitä ei ole.
4. Käytetään tiedonvarastointiin tarkoitettuja kääreitä hyväksi. Tämän ratkaisun ansiosta meidän ei tarvitse itse tehdä jokaiselle tietotyypille omaa käsittelijää. Huonona puolena on, että joudumme silti liittymään jokaiseen kääreeseen erikseen sekä hoitamaan tiedon prosessoinnin itse.
5. Luetaan tieto suoraan tietolähteistä. Tämä ratkaisu ei enää juurikaan käytä hyväkseen olemassa olevaa tietovarastoratkaisua. Ainoastaan voimme hyötyä siitä, että tietolähteet on mahdollisesti koottu tai ryhmitelty jotenkin loogisesti. Kaiken muun joudumme tekemään itse.

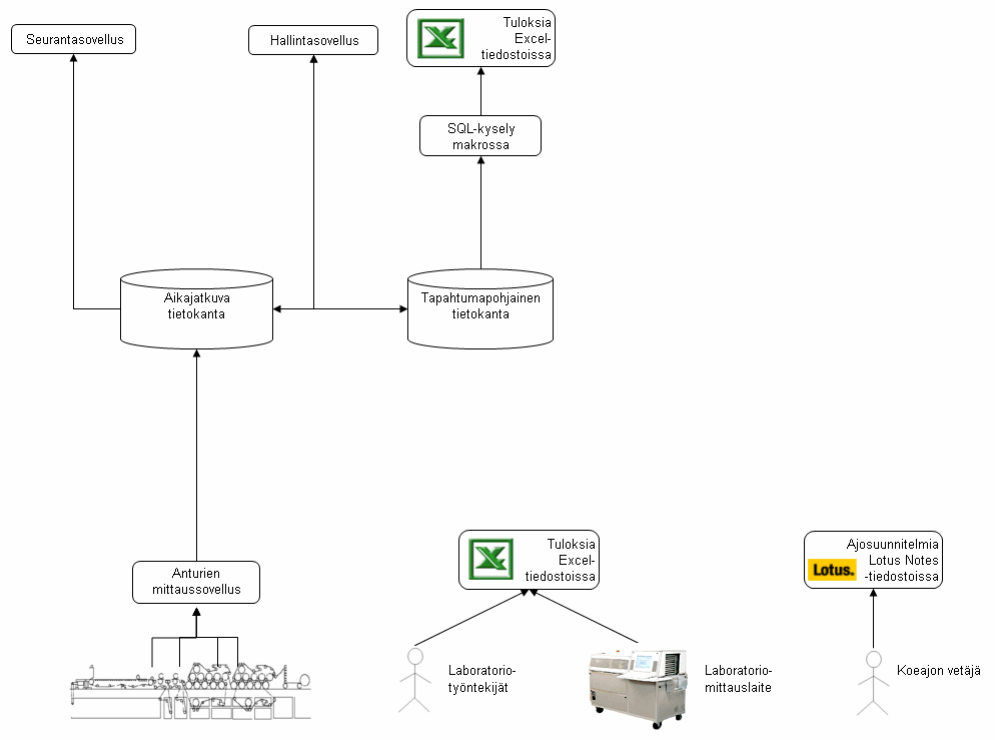
9 Esimerkkitapaus: Metso Paper

Tutkielman kokeellinen osuus sai alkunsa Metso Paper Oy:n aloitteesta. Metso Paper Oy on yksi maailman johtavia paperikoneen valmistajia ja teknologian kehittäjiä. Kokeellinen osuus käsittelee Metso Paper Oy:n paperikoneen ajoprosessin aikaisen mittausdatan keräämiseen ja seurantaan tarkoitettujen järjestelmien välisiä integrointiratkaisuja. Huomioitavaa on, että Metso Paper Oy:lla on tämän työn teon aikana käynnissä järjestelmien päivitys. Tässä luvussa perehdytään ensin vanhaan järjestelmään ja tutkitaan, mitä mahdollisia integrointipisteitä se sisältää. Integrointipisteillä tarkoitetaan kohtia, joihin jollain integrointimenetelmällä on mahdollista liittyä. Sen jälkeen tutkitaan uutta järjestelmää ja sen mukanaan tuomia integrointiratkaisuja sekä mahdollisia jatkokehitysideoita. Lopuksi esitellään toteutettu integrointiratkaisu, jolla vanhan järjestelmän tietovarastot integroidaan käyttäjän kannalta osaksi uutta järjestelmää.

9.1 Vanha järjestelmä

Metso Paper Oy:n vanha järjestelmä on ollut toiminnassa jo useita vuosia ja sen tarjoamat resurssit eivät enää vastaa tämän päivän tarpeita. Järjestelmä on kuitenkin ollut toiminnassa koko ajan ja sen keräämää tietoa on tallessa runsaasti. Pääasiassa järjestelmä toimii paperikoneella tehtävien koeajojen mittausdatan kerääjänä. Tietoa on tallennettu myös laboratoriossa tehdyistä kokeista.

9.1.1 Järjestelmän arkkitehtuuri ja toiminta



Kuva 9-1: Vanhan järjestelmän arkkitehtuuri.

Metso Paper Oy:n vanhan järjestelmän arkkitehtuuri on kuvan 8-1 mukainen. Koeajo alkaa siten, että koeajon vetäjä suunnittelee ja kirjaa koeajon parametrit Lotus Notes -kantaan. Tämä tehdään vanhassa järjestelmässä vain kirjanpidon ja jälkiseurannan takia. Seuraavaksi koeajon vetäjä syöttää parametrit hallintasovellukseen ja käynnistää koeajon. Järjestelmän pyöriessä anturien mittaussovellus kerää tietoa paperikoneeseen liitetyiltä antureilta. Nämä mittausarvot välitetään aikajatkuvaan tietokantaan. Tallennukset aikajatkuvaan tietokantaan tehdään parametreilla määriteltävin aikavälein. Koeajon ollessa käynnissä kerää hallintasovellus antureiden mittausarvoja aikajatkuvasta tietokannasta ja tallentaa halutulla hetkellä arvot tapahtumapohjaiseen tietokantaan. Tästä tallennushetkestä käytetään nimitystä koepiste. Koepisteitä otetaan esimerkiksi ennen ja jälkeen paperikoneen säätämistä uuteen tilaan. Kun koeajo on saatu päätökseen, voidaan järjestelmän keräämiä tuloksia seurata erillisellä seurantasovelluksella tai hakemalla tulokset Excel-tiedostoon SQL-makron avulla. Tämän lisäksi aina kun otetaan koepiste, otetaan samalla myös valmiista papereista näyte, joka viedään laboratorioon.

Laboratoriossa paperin laatua mitataan erilaisilla mittalaitteilla ja tulokset kirjoitetaan ylös Excel-taulukkoon. Excel-taulukoille on olemassa valmis pohja, joten ne ovat kaikki samanmuotoisia.

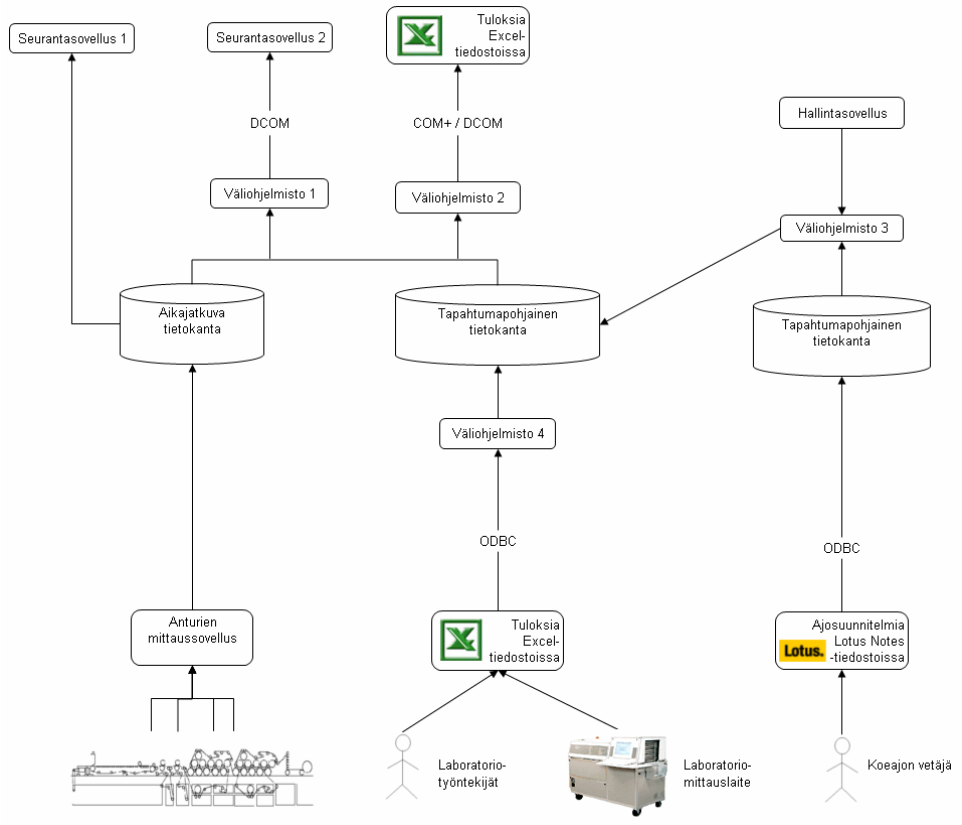
9.1.2 Integrointikohteita

Vanhassa järjestelmässä tietokannat olisivat hyviä integrointikohteita, jos ne sisältäisivät kaiken mahdollisen tiedon. Mutta koska tietokannoista häviää vanhaa tietoa automaattisen tilansäästötoiminnon vuoksi, ne eivät ole parhaita mahdollisia integrointikohteita. Prosessidata on tallennettu myös Excel-tiedostoon koeajojen päätteeksi ja laboratoriomittaukset menevät tässä järjestelmässä pelkästään Excel-tiedostoihin. Lisäksi koeajosuunnitelmat ovat Lotus Notes -tiedostoissa. Näihin tiedostoihin on mahdollista liittyä ODBC-yhteyden avulla, kuten uudessa järjestelmässä on tehty, tai käyttämällä joitakin niiden lukemiseen tarkoitettuja erikoiskomponentteja. Tämän gradun käytännön sovellus on tehty käyttäen erästä tällaista komponenttia. Nämä Apache Jakarta POI-projektin komponentit on esitelty luvussa seitsemän.

9.2 Uusi järjestelmä

Metso Paper Oy:n uusi järjestelmä tehtiin korjaamaan vanhassa järjestelmässä ilmenneitä puutteita. Tallennuskapasiteettia kasvatettiin huomattavasti. Koeajon suunnittelu- ja mittaustietojen keräys on hyvin pitkälle automatisoitu. Järjestelmässä onkin nähtävillä erilaisia integrointiratkaisuja sekä hieman tietovarastotyyppinen toteutus.

9.2.1 Järjestelmän arkkitehtuuri ja toiminta



Kuva 9-2: Uuden järjestelmän arkkitehtuuri.

Uuden järjestelmän toiminta on pääperiaatteiltaan samanlainen kuin vanhan järjestelmän. Koeajon suoritus alkaa koeajon vetäjän tekemillä suunnitelmilla. Suunnitelmat tallennetaan Lotus Notes –tiedostoihin, joista ne välittyvät ODBC-yhteyden avulla tapahtumapohjaiseen tietokantaan. Tämän jälkeen koeajon vetäjä voi halutessaan aloittaa koeajon varsinaisen suorituksen hyväksymällä ajosuunnitelmat yksinkertaisella hallintasovelluksella. Koeajosuunnitelmat siirtyvät eräajotyyppisesti toiseen tapahtumapohjaiseen tietokantaan. Koeajon ollessa käynnissä toimii anturien mittaussovellus samoin kuin vanhassakin järjestelmässä. Tiedot välittyvät ensin aikajatkuvaan tietokantaan ja sieltä valitulla tarkkuudella olevat keskiarvot tapahtumapohjaiseen tietokantaan. Laboratoriotulosten käsittely on uudessa järjestelmässä otettu hyvin huomioon. Tulokset siirtyvät väliohjelmiston 4 kautta ODBC-yhteyttä käyttäen tapahtumapohjaiseen tietokantaan. Väliohjelmistot 1 ja 2 on tehty tulosten

tarkasteluun. Näiden väliohjelmistojen avulla tulokset voidaan siirtää Excel-tiedostoon tai niitä voidaan tarkastella erillisellä seurantasovelluksella.

9.2.2 Käytetyt integrointiratkaisut

Uuden järjestelmän toteutuksessa on käytetty eräitä tyypillisiä integrointiratkaisuja. Koeajosuunnitelmat tapahtumapohjaisesta tietokannasta toiseen siirtävä väliohjelmisto on hyvä esimerkki datatason integroinnista. Tarkemmin ottaen väliohjelmisto käyttää tietokannasta tietokantaan -integrointia. Toinen integroinnin kohde on ollut laboratoriotuloksien lukeminen Excel-tiedostoista. Tämän hoitaa väliohjelmisto 4, jonka voisi sijoittaa ohjelmointirajapintatasolle. Väliohjelmisto 4 suorittaa Excelin lukemisen kutsumalla ODBC-rajapintaa. Tämän avulla se saa kaikki laboratoriotulokset käyttöönsä ja voi lähettää ne edelleen tapahtumapohjaiseen tietokantaan. Väliohjelmistot 1 ja 2 käyttävät puolestaan hyväkseen hajautettuja objekteja. Käytettävä teknologia, tässä tapauksessa DCOM, mahdollistaa tuloksien seurannan ja haun verkon yli tietokoneelta missä tietokannat fyysisesti sijaitsevat.

Arkkitehtuurisessa mielessä väliohjelmistot 1, 2 ja 4 ovat integrointisovittimia. Ne toimivat yhteyden muodostajana tietokantaan. Väliohjelmistot 1 ja 2 hakevat tietoa tietokannasta ja väliohjelmisto 4 tallentaa tietoa sinne. Näiden väliohjelmistojen tapauksessa asiakasosapuolia on vain yksi jokaista väliohjelmistoa kohden: väliohjelmisto 1:n kohdalla seurantasovellus, väliohjelmisto 2:n ja 4:n kohdalla Excelin makrot. Väliohjelmisto 3 käyttää puolestaan integrointilähettiarkkitehtuuria. Tässä tapauksessa väliohjelmisto 3 toimii vain yksinkertaisen tiedon välittäjänä kahden tietokannan välillä. Intengrointilähettiarkkitehtuurin mukaisesti toiminnanohjaus sekä aloituskäsky tulevat hallintasovellukselta.

9.2.3 Integrointikohteita

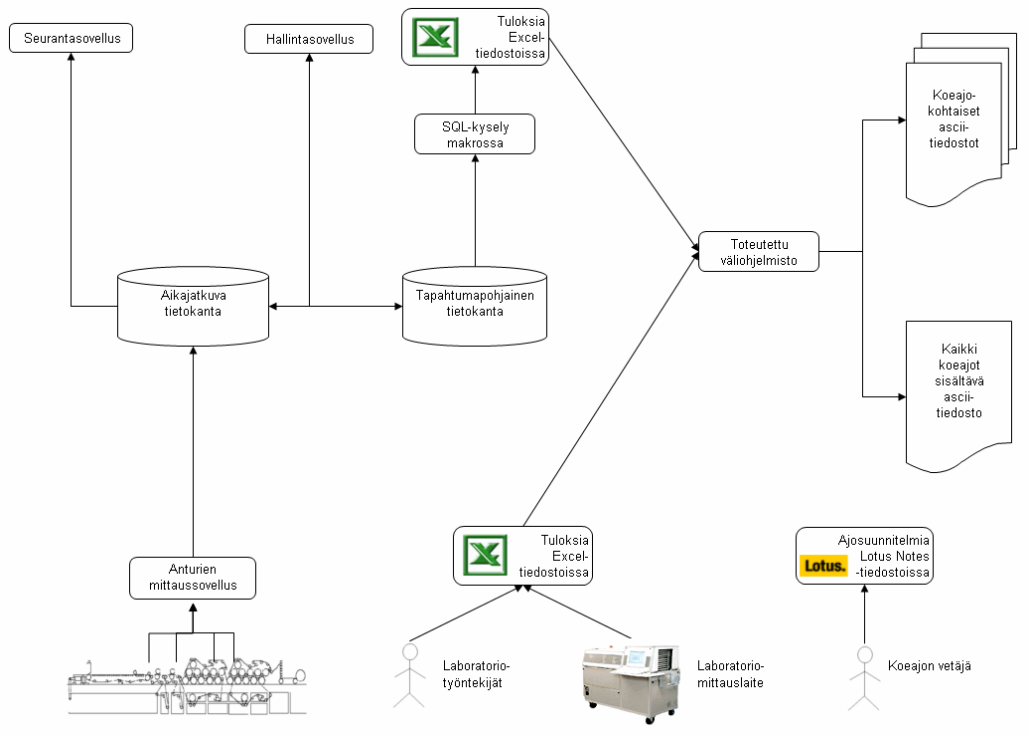
Metso Paper Oy:n uudessa tiedonkeruujärjestelmässä on monia integrointipisteitä, joihin voidaan liittyä. Nämä mahdollisuudet on hyvä ottaa huomioon tulevaisuuden laajennuksia varten. Tässä järjestelmässä avainasemassa ovat tietokannat. Koska tietokannoissa olevaa tietoa ei vielä tämän järjestelmän puitteissa juurikaan jalosteta, ovat tietokannat myös

kaikista suositeltavimpia integrointikohteita. Haluttaessa mahdollisimman tarkat anturien mittaustulokset kannattaa liittyä aikajatkuvaan tietokantaan. Jos integroitavalle sovellukselle riittää pelkkä koepistekohtainen tieto, kannattaa liittyä ensimmäiseen tapahtumapohjaiseen tietokantaan. Samalla kertaa ensimmäisestä tapahtumapohjaisesta tietokannasta saa myös koeajosuunnitelmat ja laboratoriotiedostot. Teknisessä mielessä liittymiseen voisi käyttää esimerkiksi ODBC- tai JDBC-yhteyttä. Uudessa järjestelmässä on myös mahdollisuus liittyä Excel-tiedostoihin. Tässä tapauksessa saavutetaan ainakin hyvät datasiirtomahdollisuudet. Excel-tiedostot voidaan siirtää ensin johonkin toiseen paikkaan ja ajaa muunnostoiminnot vasta siellä. Tulevaisuuteen varautumista uuden järjestelmän kohdalla kannattaa tarkastella myös siitä näkökulmasta voidaanko laboratoriotiedostojen muotoa muutella. Tämä tulee tarpeelliseksi siinä vaiheessa, kun laboratorioon tulee uusia mittauslaitteita. Myös mahdollisten ulkoapäin tulevien tietojen integroiminen järjestelmään tulisi ottaa huomioon.

9.3 Sovellusten välisen integrointiratkaisun suunnittelu ja toteutus

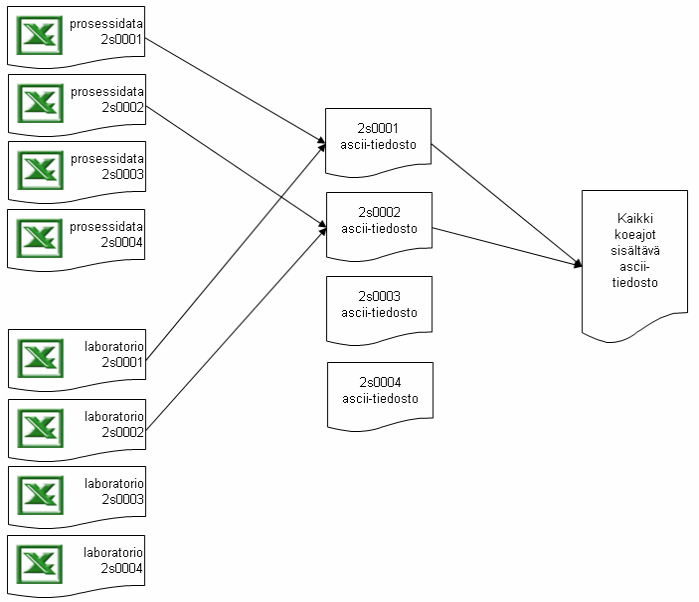
Integrointisovelluksen tarve tuli esille Metso Paper Oy:n halusta päästä hyödyntämään myös vanhassa järjestelmässä olevaa tietoa. Kuten aiemmin on esitetty, vanhassa järjestelmässä tietoa on tietokannoissa sekä Excel- ja Lotus Notes -tiedostoissa. Koska tietokantojen tiedot ovat epätäydellisiä, päädyttiin data lukemaan Excel-tiedostoista. Tavoitteena oli Excel-tiedostojen sisältämien tietojen looginen ryhmittely ja muuntaminen sellaiseen muotoon, että niitä voidaan helposti lukea esimerkiksi MatLab-ohjelmistolla. Ajosuunnitelmat, jotka sijatsevat Lotus Notes –tiedostoissa, jätettiin vielä tässä vaiheessa toteutuksen ulkopuolelle.

9.3.1 Väliohjelmiston toiminta



Kuva 9-3: Toteutetun väliohjelmiston suhde vanhaan järjestelmään.

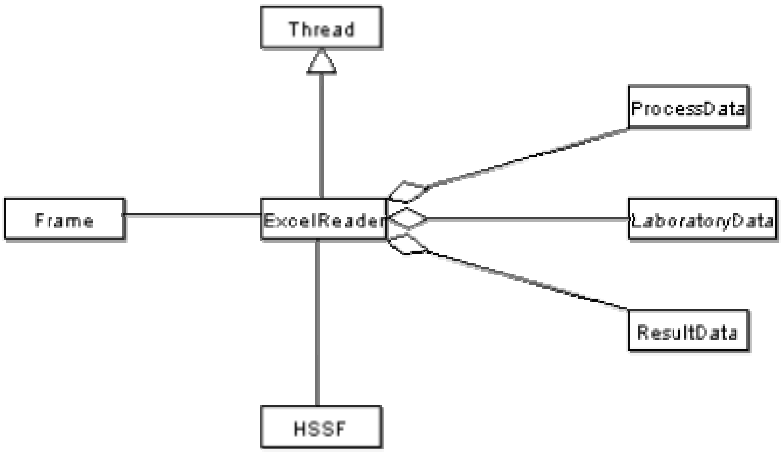
Tämän työn yhteydessä kehitetty väliohjelmisto lukee tuloksena saatuja Excel-tiedostoja suoraan vanhalta järjestelmältä ja kirjoittaa tiedot MatLab-yhteensopiviin ascii-tiedostoihin. Väliohjelmisto toimii siis puhtaasti datatasolla. Arkkitehtuurisessa mielessä väliohjelmisto on integrointisovitin, jossa Excel toimii palvelimena ja MatLab asiakkaana.



Kuva 9-4: Tiedon muunnosprosessi.

Käytännössä tiedon muunnosprosessi kulkee kuvan 9-4 mukaisesti. Ensin luetaan yksi prosessidataa sisältävä Excel-tiedosto ja tämän jälkeen etsitään siihen liittyvä laboratoriodataa sisältävä Excel-tiedosto ja luetaan myös se. Nämä tiedot kirjoitetaan yhteen ascii-tiedostoon, jonka jälkeen tämä tiedosto sisältää kaikki yhteen koeajoon liittyvät tiedot. Seuraavaksi luetaan uusi prosessidataa sisältävä Excel-tiedosto ja jatketaan samoin kuin edellä. Kun kaikki prosessi- ja laboratoriotiedostot on luettu, kootaan vielä kaikki tuloksena saadut koeajokohtaiset ascii-tiedostot yhdeksi isoksi ascii-tiedostoksi.

9.3.2 Väliohjelmiston arkkitehtuuri



Kuva 9-5: Väliohjelmiston arkkitehtuuri.

Väliohjelmiston arkkitehtuuri on pääpiirteissään kuvan 9-5 mukainen. Väliohjelmistolla on käyttöliittymäluokka `Frame`, jolla säädetään asetukset ja käynnistetään `ExcelReader`. `ExcelReader` on sovelluksen toiminnanohjauskomponentti. Se on peritty `Thread`-luokasta, jotta sille on saatu oma säie. Tämä mahdollistaa käyttöliittymän käytön sekä välitietojen näyttämisen käyttäjälle myös tiedostojen muunnosprosessin aikana. `ExcelReader` käyttää Excel-tiedostojen lukemiseen Apache Jakarta POI projektin `HSSF`-komponenttia. Tämän komponentin avulla saadaan Excel-tiedosto avattua ja suljettua. Datan lukeminen ja kirjoittaminen on siirretty omiin luokkiinsa: prosessidatan käsittely `ProcessData`-luokkaan, laboratoriodatan käsittely `LaboratoryData`-luokkaan ja tulosten käsittely `ResultData`-luokkaan.

9.3.3 Väliohjelmiston arviointi

Toteutettu väliohjelmisto toimii annettujen vaatimuksien mukaan, eli se osaa yhdistää yhteen koeajoon liittyvät prosessi- ja laboratorio-tiedostot sekä koota näistä yhden ison tiedoston. Tallennus tehdään MatLab-sovelluksen ymmärtämässä muodossa, jolla mahdollistetaan tiedon jatkokäsittely tiedonlouhinta- ja klusterointimenetelmillä. Väliohjelmiston toteutuksessa oli myös ongelmia. Ongelmat johtuivat muun muassa HSSF-komponentissa ilmenneistä virheistä. Esimerkiksi kaavojen laskennallisten arvojen lukeminen Excel-tiedostosta ei vielä onnistu. HSSF-komponentti on kuitenkin vielä kovan

kehityksen alla ja päivityksiä tulee kokoajan. Toinen ongelma väliohjelmiston toteutuksessa oli Excel-tiedostojen monimuotoisuus. Kaikki muunnettavat tiedostot eivät olleet käytössä olevien muotoilujen mukaisia. Tämä ratkaistiin ohittamalla tiedostot muunnosprosessin aikana ja merkitsemällä ne `errorlog`-tiedostoon. Näin tiedostot voidaan jälkeenpäin tarkastaa yksitellen ja korjata mahdolliset virheet.

Jatkokehitys ideoina väliohjelmistolle olisi useamman muotoisten Excel-tiedostojen lukemisen mahdollistaminen sekä koeajosuunnitelmien lukeminen ja mukaan liittäminen Lotus Notes -tiedostoista.

10 Yhteenveto

Näihin päiviin saakka tietotekniikan käyttö organisaatioiden toiminnassa on aina vain kasvanut ja näyttää kasvavan edelleen. Tämä johtaa yleensä tilanteeseen, jossa organisaatioon on kerääntynyt ajanmittaa useita erilaisia järjestelmiä. Jo pelkästään yhden järjestelmän päivittäminen kerralla uudempaan voi johtaa tilanteeseen, jossa tieto on eri paikassa kuin missä sitä haluttaisiin käyttää. On myös turhaa resurssien tuhlaamista pitää yllä samaa tietoa useissa eri järjestelmissä. Jos asiaan ei ole kiinnitetty huomiota, saattaa esimerkiksi yhden henkilön puhelinnumeron muuttaminen vaatia muutoksia kaikkiin niihin järjestelmiin joissa säilytetään henkilötietoja.

Edellä mainittujen tilanteiden ratkaisemiseen on vaihtoehtona kaikkien järjestelmien suunnitteleminen uudelleen ja yhden ison järjestelmän toteuttaminen. Tämä on ymmärrettävästi todella iso tehtävä ja kaiken lisäksi myöhemmin uusien tarpeiden ilmetessä ollaan yleensä takaisin lähtötilanteessa. Toinen vaihtoehto on integroida olemassa olevat järjestelmät toimimaan keskenään ja hyödyntämään yhteisiä tietolähteitä. Tällöin on kyse organisaation järjestelmäintegraatiosta. Oikein toteutettuna integrointiratkaisujen käyttö mahdollistaa myös tulevien järjestelmien liittämisen olemassa oleviin järjestelmiin hyvin pienellä vaivalla.

Organisaation järjestelmäintegraatio on toisaalta vaikea mutta myös palkitseva tehtävä. Integrointiratkaisujen suunnitteleminen ja toteuttaminen vaatii tarkkaa tietoa käytössä olevista järjestelmistä. Lisäksi täytyy tuntea monenlaisia teknologioita ohjelmistotekniikan alalta ja myös niiden soveltuvuus erilaisiin ratkaisuihin on tunnettava. Loppujen lopuksi voidaan kuitenkin saavuttaa hyvin yhteentoimivat järjestelmät, joita on myöhemmin helppo laajentaa.

Järjestelmäintegraatioiden suunnittelu aloitetaan tarpeiden kartoittamisella ja olemassa olevien järjestelmien hahmottelemisella. Tämän jälkeen päätetään käytettävän integrointiarkkitehtuurin ottaen huomioon nykyiset järjestelmät sekä mahdolliset tulevaisuuden laajennukset. Seuraavaksi tutkitaan millä integrointitasolla kohteena oleviin järjestelmiin on mahdollista liittyä ja miettitään väliohjelmiston tyyppiä ja toteutustekniikoita.

Tutkielman kokeellisessa osuudessa tutkittiin Metso Paper Oy:n tiedonkeruujärjestelmien integrointimahdollisuuksia. Vanhasta järjestelmästä löydettiin monia integrointipisteitä, joihin on hyvä liittyä. Näitä pisteitä on käytetty hyväksi myös uuden tiedonkeruujärjestelmän toteuttamisessa. Kokeellisessa osuudessa toteutettiin myös oma integrointiratkaisu, joka perustuu datatasolla toimivaan väliohjelmistoon. Ratkaisussa pyrittiin samaan tieto mahdollisimman yleiskäyttöiseen muotoon, jota erityisesti MatLab-ohjelmalla voisi lukea.

Kokonaisuutena tämä tutkielma toimii ohjeena järjestelmäintegraatioiden suunnittelussa. Tutkielmassa selvitettyjen ratkaisujen avulla aihetta on helpompi lähestyä tulevissa projekteissa. Vaikka teknologiat kehittyvät nopeasti, ovat tutkielmassa esitetyt ylemmän tason arkkitehtuurit ja väliohjelmistojen rakenteelliset perusratkaisut pitkäikäisiä.

Lähteet

[Agh02] Gul A. Agha, “Aaptive Middleware”, Communications of the ACM, June 2002 / Vol. 45, No. 6.

[Apa03] The Apache Software Foundation, “Apache Jakarta POI”, http://jakarta.apache.org/poi/, 2003.

[App98] Vidur Apparao, Steve Byrne, Mike Champion, Scott Isaacs, Ian Jacobs, Arnaud Le Hors, Gavin Nicol, Jonathan Robie, Robert Sutor, Chris Wilson, Lauren Wood, “Document Object Model (DOM) Level 1 Specification Version 1.0”, W3C, 1998.

[Ben95] Ron Ben-Natan, “CORBA: a guide to common object request broker architecture”, McGraw-Hill, 1995.

[Ben00] Brian Bennett, Bill Hahm, Avrahm Leff, Thomas Mikalsen, Kevin Rasmus, James Rayfield, Isabelle Rouvellou, “A Distributed Object Oriented Framework to Offer Transactional Support for Long Running Business Processes”, Springer-Verlag New York, Inc., 2000.

[Bra00] Tim Bray, Jean Paoli, C. M. Sperberg-McQueen, Eve Maler, “Extensible Markup Language (XML) 1.0 (Second Edition)”, W3C, 2000.

[Cla99a] James Clark, Steve DeRose, “XML Path Language (XPath) Version 1.0”, W3C, 1999.

[Cla99b] James Clark, Steve DeRose, “XSL Transformations (XSLT) Version 1.0”, W3C, 1999.

[ebX03] ebXML, http://www.ebxml.org, 2003.

[Ell01] John Ellis, Linda Ho, Maydene Fisher, “JDBC Specification 3.0 Final Release”, Sun Microsystems Inc., 2001.

[Gar98] Stephen R. Gardner, “Building the Data Warehouse”, Communications of the ACM, September 1998 / Vol. 41, No. 9.

[Gia01] Gianpaolo Rinetti, Mauricio Rubio, “Role of Middleware in Application Integration”, Article, Helsinki University of Technology, 2001.

[Hap02] Mark Hapner, Rich Burridge, Rahul Sharma, Joseph Fialli, Kate Stout, “JavaTM Message Service Specification Version 1.1”, Sun Microsystems, INC, 2002.

[ISO95] ISO/IEC 9075-3, “Information Technology - Database languages - SQL - Part 3: Call-Level Interface”, 1995.

[Jon98] Dr. Katherine Jones, “An Introduction to Data Warehousing: What are the Implications for the Network?”, International Journal of Network management, 8/1998.

[Lea00] Anne C. Lear, “XML Seen as Integral to Application Integration”, Article, IT Pro, September / October 1999.

[Lee03] Jinyoul Lee, Keng Siau, Soongoo Hong, “Enterprise Integration with ERP and EAI”, Communications of the ACM, February 2003 / Vol. 46, No. 2.

[Lin99] David S. Linthicum, “Mind Versus Muscle”, DevX, 12/1999.

[Lin00a] David S. Linthicum, “EAI Application Integration Exposed”, Software Magazine, 2/2000.

[Lin00b] David S. Linthicum, “Enterprise Application Integration”, Addison Wesley Longman Inc., 2000.

[Lin01] David S. Linthicum, “B2B Application Integration”, Addison-Wesley, 2001.

[Lut00] Jeffrey C. Lutz, “EAI Architecture Patterns”, EAI Journal, March 2000.

[Mar00] Didier Martin, Mark Birbeck, Michael Kay, Brian Loesgen, Jon Pinnock, Steve Livingstone, Peter Stark, Kevin Williams, Richard Anderson, Stephen Mohr, David Baliles, Bruce Peat, Nikola Ozu, "Professional XML", Wrox Press Inc., 2000.

[Mat01] Matjaz B. Juric, S. Jeelani Basha, Rick Leander, Ramesh Nagappan, "Professional J2EE EAI", Wrox Press Ltd., 2001.

[Mic03] Microsoft Corporation, "Microsoft MSDN Library", http://msdn.microsoft.com/library, 2003.

[Mow97] Mowbray, Thomas J, "Inside CORBA: distributed object standards and applications", Addison Wesley Longman Inc., 1997.

[Nis00] Pekka Niskanen, Mikko Kontio, Kimmo Vierimaa, "Inside Enterprise Java", IT Press, 2000.

[OMG02a] OMG Object Management Group Inc., "Common Object Request Broker Architecture: Core Specification Version 3.0.2", 2002.

[OMG02b] OMG Object Management Group Inc., "CORBA Components Version 3.0", 2002.

[Ope95] Open Group CAE Specification, "Data Management: SQL Call Level Interface (CLI)", 1995.

[Päi01] Päivärinta, T., "A Genre-Based Approach to Developing Electronic Document Management in the Organization", Väitöskirja, Jyväskylän yliopisto Tietojenkäsittelytieteen laitos, 2001.

[Rit98] David Ritter, "The middleware muddle", ACM Press/SIGMOD Record, December 1998 / Vol. 27 No. 4.

[Ros03] RosettaNet, http://www.rosettanet.org, 2003.

[Run00] Elke A. Rundensteiner, Andreas Koeller, Xing Zhang, “Maintaining Data Warehouses Over Changing Information Sources”, Communications of the ACM, June 2000 / Vol. 43, no. 6.

[Sto99] Michael Stonebraker, “Integrating Islands of Information”, EAI Journal, September / October 1999.

[SUN99] Sun Microsystems Inc., “Getting Started with the JDBC API”, http://java.sun.com/j2se/1.4.1/docs/guide/jdbc/getstart/GettingStartedTOC.fm.html, 1999.

[Tho97] Dean Thompson, Chris Exton, Leah Garret, A.S.M Sajeev, Damien Watkins, “Distributed Component Object Model (DCOM)”, Article, Monash University Melbourne, Australia, February 1997.