Hajautetut järjestelmät ja CORBA-standardi

Riku Tuominen

1.8.2004

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu –tutkielma

Tiivistelmä

Hajautettujen järjestelmien rakentaminen on pääsääntöisesti kalliimpaa ja monimutkaisempaa kuin keskitettyjen järjestelmien. Merkittävänä syynä tähän ovat hajautetuille järjestelmille ominaiset ja niissä usein toistuvat ongelmat. CORBA-standardi määrittelee hajautettujen järjestelmien toteutusalustan, joka sisältää tällaisten ongelmien ratkaisemiseen tarkoitettua toiminnallisuutta. Tässä tutkielmassa tutustutaan hajautettuihin järjestelmiin ja CORBA-standardiin, sekä tutkitaan CORBA:n käyttökelpoisuutta hajautettujen järjestelmien toteutusalustana. Tutkiminen tehdään jakamalla hajautetuille järjestelmille ominaiset ongelmat kuuteen eri ongelma-alueeseen ja tarkastelemalla CORBA:n näihin ongelma-alueisiin tarjoamia teknisiä ratkaisuja. Tutkimus keskittyy itse standardiin, ei sen toteutuksiin. Tämän vuoksi esimerkiksi tehokkuutta tutkitaan vain teoreettisella tasolla. Tutkimus osoitti CORBA:n olevan erittäin käyttökelpoinen toteutusalustaksi, sillä se tarjoaa paljon valmiita ratkaisuja hajautettujen järjestelmien keskeisiin ongelma-alueisiin. Erityisen kiittävästi standardissa on ratkaistu yhteensopivuusongelmat. Tietoturvaan liittyvissä ratkaisuissa sen sijaan on puutteita.

ACM-luokat (ACM Computing Classification System, 1998): C.1.4, C.2.4, D.4.7

Avainsanat: Hajautetut järjestelmät, CORBA

Sisältö

1 Johdanto

Hajautetut järjestelmät eli kokoonpanot joissa järjestelmän toiminnallisuus on hajautettu useaan eri palvelinkoneeseen, ovat monesti mielekäs vaihtoehto yhden ainoan palvelimen ympärille rakennetuille keskitetyille järjestelmille. Syynä toiminnan hajauttamiseen voi olla yksittäisen koneen riittämätön suorituskyky, tarve sijoittaa järjestelmän osia maantieteellisesti eri paikkoihin tai vaikkapa järjestelmän luotettavuuteen kohdistuvat tavallista suuremmat vaatimukset. Tämän tutkielman tarkoituksena on tutustua hajautettuihin järjestelmiin ja CORBA-standardiin. Lisäksi tutkielman tarkoituksena on tutkia CORBA-standardin käyttökelpoisuutta hajautettujen järjestelmien toteutusalustana.

Tutustumme hajautettuihin järjestelmiin luvussa 2. Aloitamme tämän tutustumisen esimerkeillä kohdassa 2.1. Nämä esimerkit eivät kuitenkaan esittele tutkielmassa keskeisen mielenkiinnon kohteena olevia hajautettuja järjestelmiä, vaan pikemminkin hajautettujen järjestelmien diversiteettiä. Kohdassa 2.2 tutustumme hajautetun järjestelmän määritelmään sekä aihepiirin peruskäsitteisiin. Tällaisia ovat löyhästi kytketty hajautettu järjestelmä, heterogeenisyys sekä hajautetun järjestelmän toteutusalusta. Kohdassa 2.3 tutustumme väliohjelmistoon, joka on yleensä käyttöjärjestelmän ja sovellusten väliin sijoittuva ylimääräinen kerros. Väliohjelmistot ovat tutkielmassa erityisen mielenkiinnon kohteena, sillä CORBA-standardi on toteutettu väliohjelmistona. Kohdassa 2.4 tutustumme objektien käyttöön hajautetuissa järjestelmissä.

Luvussa 3 pääsemme tutustumaan itse CORBA-standardiin, joka on suosittu hajautettujen järjestelmien toteutusympäristöjä kuvaava standardi. Tässä luvussa tutustumme itse standardin lisäksi sen keskeisiin rakenneosiin ja käsitteisiin, joita ovat muun muassa OMA, ORB, IOR, GIOP, IIOP, IDL, BOA, SII ja DII. Samalla saamme merkityksen näille sekä monille muille lyhenteille, joita standardi sisältää runsain mitoin.

Luvussa 4 käymme läpi hajautettujen järjestelmien teknisiä avainasioita ja tutkimme niiden avulla CORBA:n käyttökelpoisuutta hajautetun järjestelmän toteutusalustana.

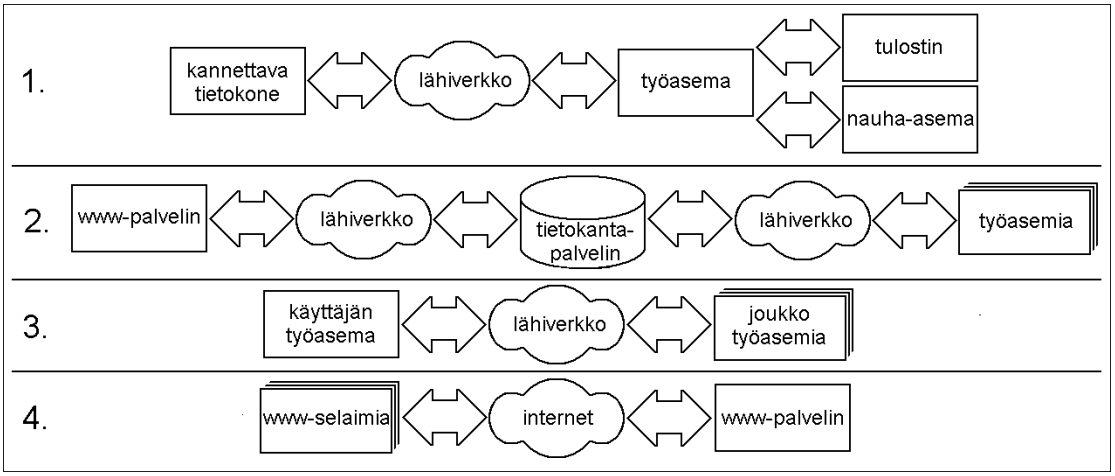
Tekniset avainasiat ovat erityisen keskeisessä asemassa olevia hajautettujen järjestelmien teknologia luonteisia ongelma-alueita. Ne sopivat edellä mainittuun käyttökelpoisuuden tutkimiseen hyvin, sillä tutkimalla CORBA-standardin näihin ongelma-alueisiin tarjoamia ratkaisuja, saadaan esiin standardin merkittävimmät ominaisuudet tai niiden puutteet.

2 Hajautetut järjestelmät

Tässä luvussa tutustumme hajautettuihin järjestelmiin. Aloitamme käymällä läpi kohdassa 2.1 muutamia esimerkkejä tavanomaisista järjestelmistä, jotka mahdollistavat resurssien jakamisen useamman laitteen tai käyttäjän kesken. Kohdassa 2.2 käymme läpi aihepiirin peruskäsitteitä ja hajautettujen järjestelmien rakenneosia. Näihin kuuluu hajautetun järjestelmän määritelmä, löyhästi kytketty hajautettu järjestelmä, heterogeenisyys ja toteutusalusta. Kohdassa 2.3 tutustumme väliohjelmistoihin ja kohdassa 2.4 käsittelemme objektien merkitystä hajautetuille järjestelmille.

2.1 Esimerkkejä hajautetuista järjestelmistä

Tässä kohdassa tutustumme esimerkeiksi valittuihin kokoonpanoihin, jotka edustavat sekä eri tyyppisiä että eri mittakaavassa toimivia hajautettuja järjestelmiä. Mukana on henkilökohtainen tietokone oheislaitteineen, työasemien jakama tietokantapalvelin, raskaaseen laskentaan tehty työasemaryväs sekä maailmanlaajuisesti tavoitettavissa oleva www-palvelin.



Kuva 1. Erilaisia hajautettuja järjestelmiä.

Kuva 1 esittelee erilaisia hajautettuja järjestelmiä. Ensimmäisessä kuvan esimerkissä on työasema, jossa on oheislaitteina tulostin ja nauha-asema. Tähän työasemaan on liitetty lähiverkon avulla kannettava tietokone, joka voi hyödyntää työasemaan

liitettyjä oheislaitteita. Näin vastaavien oheislaitteiden hankkiminen erikseen myös kannettavaan tietokoneeseen on tarpeetonta. Tällainen pienimuotoinen hajautettu järjestelmä, jossa yksittäisen tietokoneen resursseja jaetaan toisen tietokoneen kanssa on tyypillinen yksittäiselle käyttäjälle ja järjestelyllä tähdätään usein kustannussäästöihin laitteistohankinoissa. Vaikka järjestelmä on pienimuotoinen, vastaavanlaisia järjestelyitä käytetään usein myös paljon suuremmassa mittakaavassa, kuten lukuisia henkilökohtaisia työasemia sisältävissä lähiverkoissa, joissa verkkoon on sijoitettu tehokkaita verkkotulostimia sekä levypalvelimia jaettavaksi kaikkien työasemien kesken.

Toisessa esimerkissä on tietokantapalvelin, joka on jaettu sekä lähiverkon kautta siihen kytkettyjen työasemien kesken että WWW-palvelimen välityksellä tietokantapalvelimeen yhteydessä olevien Internetissä sijaitsevien tietokoneiden kesken. Tällainen järjestely mahdollistaa esimerkiksi samanaikaisesti yrityksen sisä- ja ulkopuolella työskentelevien työntekijöiden keskinäiden tiedon jakamisen. Tietokantapalvelimen tilalla (tai rinnalla) voisi olla myös levypalvelin. Huomionarvoista esimerkissä on myös hajautettujen järjestelmien hieman epäjohdonmukainen terminologia. Vaikka kyseinen järjestelmä on selvästi hajautettu ja järjestelmästä tekee hajautetun kaikille yhteisen tietokanta- tai levypalvelimen jakaminen, tämä palvelin on juuri se paikka johon kaikille järjestelmän osille yhteinen tieto tai toiminnallisuus keskitetään.

Kolmannessa esimerkissä on järjestelmä, jossa käyttäjällä on näennäisesti käytössään vain yksi työasema, mutta tosiasiassa käyttäjän työasema jakaa sille annetun tehtävän osiin, jotka se delegoi suoritettavaksi joukolle (yleensä tarpeettomista ohjauslaitteista kuten näytöistä ja näppäimistöistä) riisuttuja työasemia. Tällaista rinnakkaislaskentaa hyödyntävää työasemaryvästä voidaan käyttää, mikäli tietojenkäsittelytehtävä on liian raskas yksittäiselle tietokoneelle ja tehtävä on mahdollista jakaa erillisiin samanaikaisesti suoritettaviin osiin. Kuvan riisutut työasemat voisivat olla myös erilaisia alijärjestelmiä, joista kukin olisi erikoistunut vain tietynlaisen tehtävän ratkaisemiseen. Tällaisiä alijärjestelmiä voisivat olla esimerkiksi erillisiin tietokoneisiin sijoitetut yksittäisen tietokannan tietokantataulut, jolloin tietokantaan kohdistuva kysely voitaisiin suorittaa eri taulujen osalta samanaikaisesti ja näin lyhentää kyselyn suoritukseen tarvittavaa aikaa merkittävästi.

Neljännessä esimerkissä on nykyisin paljon käytetty hajautettu järjestelmä, jossa Internetiin sijoitettu WWW-palvelin tarjoaa WWW-sivuja maantieteellisesti mielivaltaisissa paikoissa sijaitseville WWW-selaimille. Nämä selaimet voivat sijaita yhdessä tai useammassa työasemassa, mutta esimerkin WWW-palvelin sijaitsee aina yhdessä tietyssä palvelinkoneessa, johon selainohjelmat ottavat yhteyden. Mikäli WWW-palvelinohjelmisto ja kaikki käytettävät selaimet olisi sijoitettu yksittäiseen tietokoneeseen, ei järjestelmää voisi sanoa hajautetuksi vaikka kummassakin tapauksessa käytetään täysin samoja ohjelmia.

Yhteistä kaikille kuvan järjestelmille (sekä hajautetuille järjestelmille yleisesti) onkin niiden toimintaperiaate: Järjestelmä koostuu itsenäisistä, fyysisesti erillisistä alijärjestelmistä, jotka tarjoavat niille itselleen ominaisia palveluita asiakkailleen. Järjestelmä toimii suorittamalla yksittäiset tietojenkäsittelytehtävät joko yhden tai useamman tällaisen palvelun avulla. Kun asiakasohjelma tarvitsee tehtävän suoritusta, se tekee palvelupyynnön palvelua tarjoavalle ohjelmalle. Nämä palvelupyynnöt ovat eri tietokoneisiin hajautettujen ohjelmien tapa kommunikoida keskenään, ohjelmille yhteisen tehtävän suorittamiseksi. Hajautetun järjestelmän palvelupyyntöjä voisi verrata karkeasti keskitetyssä järjestelmässä toimivan ohjelman aliohjelmakutsuihin, jotka toteuttavat ohjelman toiminnallisuutta mahdollisten syötetietojen perusteella, samalla kuitenkin piilottaen toteutuksen yksityiskohdat kutsujalta. Käytännössä hajautetun järjestelmän palvelupyynnön ja keskitetyn järjestelmän aliohjelmakutsun välillä on kuitenkin oleellisia eroja, joiden huomiotta jättäminen heikentää sovellusten tehokkuutta ja luotettavuutta. Tämä myös monimutkaistaa hajautetun järjestelmän rakentamista keskitettyyn järjestelmään verrattuna. Toisaalta siinä missä keskitetyn järjestelmän toiminnallisuus on sijoitettava vain yhteen fyysiseen paikkaan, voidaan hajautetun järjestelmän toiminnallisuutta sijoittaa tarpeen mukaan maantieteellisesti toisistaan hyvinkin etäällä sijaitseviin paikkoihin.

2.2 Hajautettujen järjestelmien peruskäsitteitä

Tässä kohdassa tutustumme hajautettujen järjestelmien keskeisiin peruskäsitteisiin kuten hajautetun järjestelmän määritelmään, löyhästi kytkettyihin hajautettuihin järjestelmiin, heterogeenisyyteen sekä hajautetun järjestelmän toteutusalustaan. Nämä käsitteet ovat avainasemassa tutkittaessa mistä hajautetuissa järjestelmissä on oikeastaan kysymys ja millaisia ongelmia niitä rakentaessa tulee vastaan.

Hajautetun järjestelmän määritelmä

Hajautetuiksi järjestelmiksi kutsutaan yhteenkytketyistä tietokoneista tai tietojärjestelmistä koostuvaa kokonaisuutta, jossa järjestelmän toiminnallisuutta on jaettu useiden laitteiden kesken. Tämä määritelmä on löyhä, mutta sisältää hajautettujen järjestelmien kaksi erittäin oleellista piirrettä. Nämä piirteet ovat laitteiden kytkeminen yhteen sekä järjestelmän toiminnallisuuden toteuttaminen käyttämällä yhteen kytkettyjä laitteita kokonaisuutena. Terminologian kannalta on ongelmallista, että lähes mikä tahansa järjestelmä jossa on useita yhteen kytkettyjä laitteita, sisältää edellä mainitut piirteet ja järjestelmä voidaan näin ollen lukea kuuluvaksi hajautettuihin järjestelmiin. Cornafion-tutkimusryhmä onkin todennut: "Vaikka termi hajautettu järjestelmä on yleisesti käytetty, sille ei ole olemassa tarkkaa ja yleispätevää määritelmää" [1]. Valitettavasti sama pätee myös moniin muihin hajautettuihin järjestelmiin liittyviin käsitteisiin. Selkeän ja yleispätevän käsitteistön puute onkin varmasti osaltaan vaikuttanut monitulkintaisen ja osittain päällekkäisen termistön syntyyn. Termistön selkeyttä ei myöskään helpota aihepiiriä käsittelevän kirjallisuuden suosiossa oleva runsas lyhenteiden käyttö.

Löyhästi kytketty hajautettu järjestelmä

Löyhästi kytketyiksi hajautetuiksi järjestelmiksi kutsutaan sellaisia järjestelmiä, jotka muodostuvat heikohkotehoisella kommunikointiverkolla toisiinsa kytketyistä tietokoneista [2]. Tällaisissa järjestelmissä on kaksi merkittävää erityispiirrettä, jotka tulee ottaa huomioon järjestelmien suunnittelussa ja kehittämisessä. Ensimmäinen näistä on järjestelmän osien välisen kommunikaatioverkon heikohko suorituskyky, jolla tarkoitetaan tässä verkon rajoittunutta tiedonsiirtokapasiteettia, tiedonsiirrossa

esiintyvää merkittävää viivettä sekä verkon virhealttiutta. Toinen erityispiirre on yhteen kytkettyjen rakenneosien keskinäinen erilaisuus eli heterogeenisyys, jota käsittelemme hieman myöhemmin.

Verkon heikohko suorituskyky ei tarkoita välttämättä sitä, että verkon tulisi olla ominaisuuksiltaan huono. Tällainen verkko voi olla toisinaan hyvinkin suorituskykyinen. Löyhästi kytketyn järjestelmän kommunikaatioverkon suorituskyky onkin heikko vain verrattaessa sitä niiden tietokoneiden sisäisen kommunikointiverkon suorituskykyyn, joita se yhdistää. Mikäli järjestelmän tietokoneita yhdistävä verkko on suorituskyvyltään samantehoinen tai parempi kuin siihen kytkettyjen tietokoneiden sisäinen kommunikaatioverkko, kyse ei ole löyhästi kytketystä hajautetusta järjestelmästä.

Tietokoneita toisiinsa yhdistävä kommunikaatioverkko toimii hajautetussa järjestelmässä eräänlaisena osia yhdistävänä *tukirankana (system backbone)*. Tämän vuoksi sen vaikutus järjestelmän eri ominaisuuksiin kuten tehokkuuteen ja luotettavuuteen on merkittävä. Verkon mahdollisesti heikko suorituskyky syytä ottaa huomioon järjestelmää suunniteltaessa. Tämä voidaan tehdä esimerkiksi erilaisilla ohjelmistoihin tehtävillä ratkaisuilla, joiden avulla verkon puutteita voidaan kompensoida. Tällaisia ratkaisuja käsitellään tutkielmassa luotettavuuden osalta kohdassa 4.4 ja tehokkuuden osalta kohdassa 4.5.

Heterogeenisyys

Hajautetun järjestelmän *heterogeenisyydellä* tarkoitetaan järjestelmän koostumista sekalaisista rakenneosista. Heterogeenisyyteen liittyy yleensä *monimuotoisuutta (variety)* ja *eroavaisuutta (difference)* [3]. Näistä monimuotoisuus tarkoittaa samantyyppisten, mutta toisistaan käytännössä eroavien rakenneosien samanaikaista käyttämistä. Monimuotoisuutta sisältävässä järjestelmässä voi olla samanaikaisesti esimerkiksi kaksi relaatiomalliin perustuvaa tietokantaohjelmistoa, jotka ovat toistensa kanssa yhteensopia mutta tulevat eri valmistajilta. Eroavaisuus puolestaan tarkoittaa käytettävien rakenneosien sellaista keskinäistä poikkeavuutta, jossa rakenneosien toimintamallit eroavat perusteellisesti toisistaan. Tällainen poikkeavuus ilmenee yleensä osien yhteensopimattomuutena. Eroavaisuutta sisältävässä

järjestelmässä aiemmin mainituista tietokantaohjelmistoista ensimmäinen voisi perustua relaatio- ja toinen oliomalliin. Tällöin tietokantaohjelmistot omaavat perusteellisesti toisistaan poikkeavat toimintamallit ja ovat erittäin suuressa määrin epäyhteensopivia toistensa kanssa.

Vaikka edellä käytettiin esimerkkeinä tietokantaohjelmistoja, eivät tietokantaohjelmistot eivätkä ohjelmistot yleensäkään ole ainoa järjestelmän heterogeenisyyteen vaikuttava tekijä. Omasta mielestäni heterogeenisyyttä voivat aiheuttaa periaatteessa kaikki järjestelmän toimintaan vaikuttavat rakenneosat tai niiden perustana olevat määrittelyt (joskin tämän määritelmän tulkitsemisessa on syytä olla menemättä liiallisuuksiin). Näin ollen on mahdollista että jokainen hajautetussa järjestelmässä oleva tietokone voi omata rakenneosan, jollaista järjestelmän muissa koneessa ei ole. Koneissa voi olla esimerkiksi erilaisia arkkitehtuureja käyttävät prosessorit, erilaiset käyttöjärjestelmät, erilaiset merkistöt, erilaiset tiedontallennusformaatit, eri tiedostojärjestelmiä käyttävät kiintolevyt, eri sovellusohjelmat sekä täysin erilaiset oheislaitekokoonpanot. Lisäksi tietokoneita voidaan käyttää eri maiden kielillä ja ohjelmoida eri ohjelmointikielillä. Myös koneita yhdistävät kommunikaatioverkot voivat olla erilaisia sekä toistensa kanssa yhteensopimattomia (esim. Ethernet ja Appletalk verkot). On huomionarvoista, että heterogeenisyydestä huolimatta järjestelmän eri osien on pystyttävä toimimaan luotettavasti yhtenä kokonaisuutena. Tämän vaatimuksen toteuttaminen tekniikan jatkuvasti kehittyessä onkin eräs hajautettujen järjestelmien suurimpia haasteita.

Hajautetun järjestelmän toteutusalusta

Hajautetun järjestelmän rakentamiseen liittyvät ongelmat ovat liian monimutkaisia ja työläitä ratkaistavaksi yhä uudelleen jokaisen uuden järjestelmän rakentamisen yhteydessä. Tämän vuoksi toteutusalustalla on merkittävä rooli järjestelmää rakennettaessa. *Toteutusalustalla* tarkoitetaan tässä valmista toiminnallisuutta tarjoavaa ympäristöä, jonka palveluita hyväksi käyttämällä ohjelmoijat voivat rakentaa sovellusohjelmansa tekemättä kaikkea itse. Ohjelmiston toteutusalustan merkitys on nykyisin suuri jo keskitetyissä järjestelmissä, joiden rakentamisessa aliohjelmakirjastot, ohjelmistokomponentit sekä kokonaiset sovelluskehykset toimivat aikaa ja kustannuksia säästävinä tekijöinä. Hajautetuissa järjestelmissä toteutusalustan

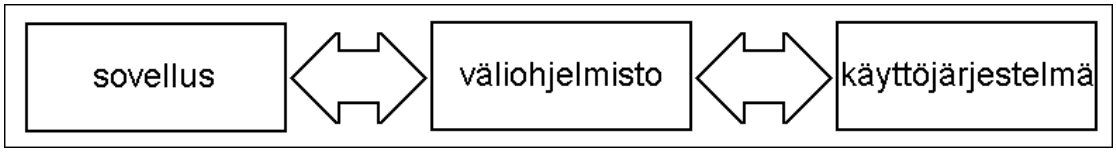
rooli on kuitenkin vielä moninaisempi kuin keskitetyissä järjestelmissä. Sovellusohjelmien rakentamiseen liittyvien valmiiden ratkaisuiden lisäksi toteutusalusta tarjoaa järjestelmän osien hajautukseen liittyvää toiminnallisuutta. Tällainen toiminnallisuus voi olla esimerkiksi joukko tapahtuman- tai samanaikaisuudenhallinnan toteuttavia palveluita, joiden avulla järjestelmän eri osat saadaan käyttäytymään yhtenä kontrolloituna kokonaisuutena. Mikäli toteutusalusta ei tarjoa riittävästi hajautukseen liittyviä palveluita, järjestelmän rakentajien on pahimmassa tapauksessa ohjelmoitava nämä palvelut itse. Edellä mainittujen asioiden lisäksi toteutusalusta toimii monesti myös rajapintana muihin järjestelmän osiin, muodostaen yhteensopivuuden osalta erään hajautetun järjestelmän kriittisimmistä kohdista.

2.3 Väliohjelmistot

Tietokoneen ohjelmointiympäristön muodostavat rajapinnat voidaan jaotella karkeasti kahteen eri ryhmään. Ensimmäisen ryhmän muodostavat toteutusalustan kuten käyttöjärjestelmän tarjoamat palvelut, jotka ovat monesti varsin yleiskäyttöisiä. Näiden palveluiden abstraktiotaso on kuitenkin varsin matala, jolloin yksittäinen palvelu toteuttaa vain vähän toiminnallisuutta. Toisen ryhmän muodostavat korkean abstraktiotason omaavat valmiit sovellusohjelmat, joiden yksittäiset toiminnot voivat toteuttaa kerralla paljon toiminnallisuutta. Tällaisten sovellusohjelmien käyttö rajoittuu usein kuitenkin vain hyvin kapealle alueelle. On tavallista että korkean tason erikoistuneet sovellusohjelmat rakennetaan käyttämällä toteutusalustan kuten käyttöjärjestelmän tarjoamia matalan abstraktiotason palveluita. Tästä kuitenkin seuraa ongelmia muun muassa sovellusohjelmien siirrettävyyden suhteen, sillä kun sovellusohjelma rakennetaan toteutusalustan palveluiden varaan, se myös sidotaan kiinni kyseiseen toteutusalustaan. Väliohjelmisto on eräs mielenkiintoinen ratkaisu, jonka avulla sovellusten ja toteutusalustan yhteenkytkennästä aiheutuvia ongelmia voidaan vähentää, menettämättä kuitenkaan yleiskäyttöisistä matalan abstraktiotason palveluista saatavaa hyötyä. Tässä kohdassa tarkastelemme väliohjelmistoa ja sen merkitystä hajautetuille järjestelmille. Tutustumme samalla merkittävimpiin ongelma-alueisiin, joihin se tarjoaa ratkaisuja.

Väliohjelmiston käsite ja sen merkitys

Väliohjelmisto (middleware) on yleiskäyttöisistä palveluista koostuvaa kerros, joka sijoittuu yleensä kahden tärkeän ohjelmistokerroksen väliin [4] (englanniksi "*in the middle*"). Hajautetuissa järjestelmissä käytettävissä väliohjelmistoissa, nämä kerrokset ovat yleensä käyttöjärjestelmä- ja sovellusohjelmakerros. Sijoittuessaan kerrosten väliin, väliohjelmisto korvaa käyttöjärjestelmäkerroksen tarjoamat palvelut omillaan, jolloin sovellusohjelmakerros sidotaan käyttöjärjestelmäkerroksen sijasta väliohjelmistokerrokseen. Käyttöjärjestelmäkerros ei kuitenkaan muutu tarpeettomaksi, sillä väliohjelmisto toteuttaa tarjoamiaan palveluita käyttöjärjestelmän tarjoamien palveluiden avulla. Väliohjelmisto yhdistääkin kerrokset toisiinsa, muodostaen uudenlaisia ominaisuuksia sisältävän kokonaisuuden. Kuva 2 havainnollistaa edellä mainittujen kolmen kerroksen keskinäistä suhdetta.



Kuva 2. Väliohjelmisto-, sovellus- ja käyttöjärjestelmäkerroksen suhde toisiinsa.

Tapaa jolla väliohjelmisto toimii eri kerrosten yhdistäjänä voidaan pitää eräänlaisena ohjelmallisena liimana. Väliohjelmistokerros ottaa vastaan pyyntöjä sovelluskerrokselta, toteuttaa pyynnöt käyttöjärjestelmää hyödyntäen ja palauttaa tulokset takaisin sovelluskerrokselle kytkien näin kerrokset yhteen. Tämä ei kuitenkaan tarkoita kerrosten kytkemistä toisiinsa pysyvästi, vaan asia on pikemminkin päinvastoin! Väliohjelmiston eräs keskeinen idea onkin eristää palveluita käyttävä sovelluskerros ja palveluita tarjoava käyttöjärjestelmäkerros toisistaan, mahdollistaen samalla kuitenkin kerrosten välisen yhteistoiminnan. Tästä nimittäin seuraa merkittäviä etuja sovellusten siirrettävyyden kannalta, sillä kun väliohjelmisto vähentää sovelluksen tarvetta käyttää käyttöjärjestelmän rajapintoja (ts. palveluita) tarjoamalla omat vastaavansa, sovellus ei ole enää täysin sidottu käyttöjärjestelmän rajapintoihin, eikä sovelluskehittäjä ole vastaavasti sidoksissa käyttöjärjestelmän valmistajaan. Tällainen alustariippumattomuus vähentää omalta

osaltaan hajautetun järjestelmän heterogeenisyyteen liittyviä ongelmia, erityisesti pitkällä aikavälillä.

Alustariippumattomuuden saavuttaminen ei kuitenkaan ole aivan näin yksinkertaista. Korvatessaan toteutusalustana toimivan käyttöjärjestelmän palvelut omillaan, väliohjelmistosta itsestään tulee se järjestelmän toteutusalustan osa johon sovellukset sidotaan. Tämän vuoksi väliohjelmisto on voitava tarvittaessa siirtää käyttöjärjestelmästä toiseen ja se on voitava tarvittaessa korvata toisella vastaavalla tuotteella. Nämä vaatimukset väliohjelmisto voi täyttää käyttämällä yksinomaan standardoituja rajapintoja. Käsitteellä *standardoitu rajapinta* tarkoitetaan tässä rajapintoja, jotka ovat hyvin määriteltyjä ja joiden määritelmät ovat kokonaisuudessaan julkisesti saatavilla (ts. rajapinnat ovat avoimia). Tällaisiin rajapintoihin perustuva väliohjelmisto on vaihdettavissa ja tätä kautta myös siirrettävissä, sillä se ei ole yleensä sidottu vain yhteen ohjelmistotuotteeseen tai valmistajaan. Hajautetuissa järjestelmissä voidaan myös joutua käyttämään useita erilaisia väliohjelmistoja samanaikaisesti, jolloin väliohjelmistojen välisen kommunikaation täytyy perustua kaikille osapuolille tuttuihin rajapintoihin. Näin ei asianlaita kuitenkaan monesti ole ja eri valmistajien väliohjelmistot eivät usein toimikaan toistensa kanssa [5].

Alustariippumattomuuden toteuttaminen ei kuitenkaan jää ainoaksi väliohjelmiston tarkoitukseksi. Sen tarkoitus on myös tarjota sovelluskehittäjille valmiita korkean abstraktiotason palveluita, jotka yksinkertaistavat hajautettujen järjestelmien rakentamista [6]. Näin kehittäjät voivat keskittyä enemmän itse sovelluksiin hajautukseen liittyvän toiminnallisuuden sijasta. Ellei nimittäin väliohjelmisto tai muu toteutusalustan osa tarjoa hajautuksen hallintaan liittyvää perustoiminnallisuutta valmiina, sen toteuttaminen jää yleensä sovelluksen ohjelmoijan tehtäväksi. Hajautettuihin järjestelmiin erikoistuneiden asiantuntijoiden toteuttama ja testaama perustoiminnallisuus tarjoaa myös luotettavan ja yhtenäisen pohjan järjestelmän rakentamiselle. Tästä on etua erityisesti silloin kun järjestelmä on niin suuri, että vastuu sen kehittämisestä ja ylläpitämisestä on jaettu usean toisistaan riippumattoman organisaation kesken. Tällöin on nimittäin olemassa vaara, että paikallisesti tehdyt ratkaisut aiheuttavat ongelmia järjestelmälle sen toimiessa kokonaisuutena. Pahimmillaan tällainen tilanne voi ilmetä vielä niin, ettei ongelmien aiheuttajaa kyetä

havaitsemaan lainkaan tarkasteltaessa järjestelmän toimintaa yhdestä yksittäisestä paikasta.

Väliohjelmistojen keskinäiset eroavaisuudet

Kaikki samojen määritelmien mukaan tehdyt väliohjelmistot eivät ole samanlaisia, eivätkä toistensa kanssa yhteensopivia. Tähän on syynä etupäässä väliohjelmiston abstrakti luonne. Väliohjelmisto on itse asiassa vain ohjelmistoarkkitehtuurinen malli, joka voidaan toteuttaa lukuisilla eri lähestymistavoilla. Tämän vuoksi konkreettiset väliohjelmistot (eli väliohjelmistotuotteet) noudattavat yleensä jotain valtavirtaan kuuluvaa ohjelmointiparadigmaa. Väliohjelmistot voidaankin luokitella niissä käytettävän ohjelmointiparadigman perusteella neljään eri luokkaan [6]. Nämä luokat ovat tapahtumapohjaiset, viestinvälitysorientoituneet, proseduraaliset sekä objekti- tai komponenttipohjaiset väliohjelmistot. Näistä viimeksi mainittuun lukeutuu OMG:n CORBA, joka on tässä tutkielmassa erityisen kiinnostuksen kohteena. Erilaisten ohjelmointiparadigmojen käyttö ei kuitenkaan saa varsinaisesti aikaan aiemmin mainittua väliohjelmistojen keskinäistä yhteensopivuutta. Sen saa mielestäni aikaiseksi poikkeminen yksiselitteisistä standardeista.

Syynä standardoitujakin rajapintoja käyttävien väliohjelmistojen keskinäiseen yhteensopimattomuuteen on yleensä useiden päällekkäisten standardien olemassaolo sekä rajapintoihin kohdistuvat valmistajakohtaiset lisäykset. Palvelukohtaisia ohjelmointirajapintoja ovat standardoineet muun muassa suuret standardointijärjestöt kuten ISO ja ANSI, muutamat tietokonealan yrityksistä koostuvat ryhmittymät kuten X/OPEN, OSF ja OMG sekä yritykset kuten IBM ja Microsoft. Useiden päällekkäisten standardien olemassaoloon on olemassa useita syitä. Tällaisia ovat muun muassa rajapintoihin kohdistuva historian painolasti, arvovaltakysymykset sekä eri ryhmittymien välinen kilpailu. Päällekkäisyys ei ole välttämättä ongelma, mikäli väliohjelmisto tukee samanaikaisesti kaikkia tarvittavia standardeja. Tällöin väliohjelmistoa voidaan käyttää samanaikaisesti useamman standardin mukaisella tavalla, eikä ongelmia pitäisi syntyä. Valmistajakohtaiset lisäykset ovat sen sijaan järjestelmän pitkän elinkaaren kannalta ongelmallisempi tekijä. Tällaiset lisäykset nimittäin perustuvat valmistajien omiin standardeihin, jotka ovat yleensä epäyhteensopivia muiden valmistajien vastaavien lisäysten kanssa (mikäli vastaavaa

lisäystä kilpailevasta tuotteesta edes löytyy). Vaikka lisäykset toisinaan sisältävätkin standardin ulkopuolelle jätettyä tarpeellista toiminnallisuutta, syyt niiden olemassaoloon ovat yleensä selkeästi kaupallisia [7]. Tämä on valitettavaa, sillä eräs väliohjelmiston tärkeimmistä tavoitteista on käyttää vain standardoituja rajapintoja.

2.4 Objektit hajautetuissa järjestelmissä

Objektikeskeisyydestä on tullut erittäin suosittu ohjelmointiparadigma. Tällaiseksi sen on tehnyt tarve systematisoida ohjelmistojen rakentamista, lisätä uudelleenkäytettävyyttä sekä helpottaa ylläpitoa [8]. Se nimittäin tarjoaa kaikkiin edellä mainittuihin seikkoihin muita ohjelmointiparadigmoja paremmat lähtökohdat. Tämän lisäksi objektiparadigman ympärille on kasvanut monipuolinen ohjelmointikulttuuri, joka tarjoaa välineitä käytännön ohjelmistokehitykseen. Näihin välineisiin lukeutuu esimerkiksi ohjelmointikieliä, luokkakirjastoja, analyysi- ja suunnittelumenetelmiä, suunnittelumalleja, sovelluskehyksiä sekä CASE-välineitä. Objektikeskeisyys on kiinnostava lähestymistapa myös hajautettujen järjestelmien rakentamiseen. Objekteihin perustuvien menetelmien on nimittäin todettu olevan hyödyllisiä hajautetuissa järjestelmissä, sillä abstrakteina datatyyppeinä määriteltävät objektit muodostavat luonnollisen perustan hajautukselle, resurssien hallinnalle sekä toiminnallisuuden abstraktoinnille [9]. Tässä kohdassa tarkastelemme lyhyesti hajautetun järjestelmän toteuttamista objektien avulla. Keskeinen osa tästä tarkastelusta kohdistuu objektien käyttöön sovellusten osittamisessa. Tutustumme myös lyhyesti aihepiirin terminologiaan.

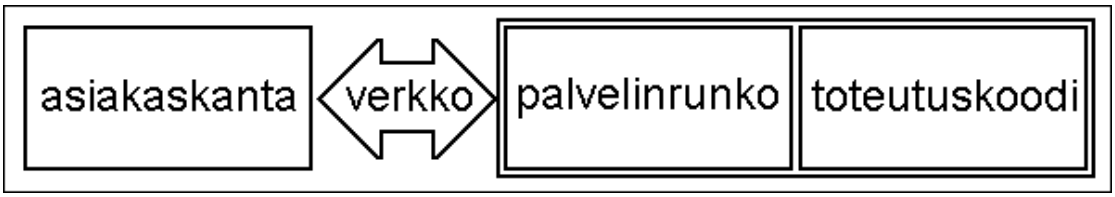
Hajautetut objektit

Hajautetussa järjestelmässä toimivan yksittäisen sovellusohjelman tulee usein toimia useammassa kuin yhdessä tietokoneessa samanaikaisesti. Tästä huolimatta sovelluksen on kuitenkin muodostettava yksi toiminnallinen kokonaisuus. Sovellus on siis tarpeen vaatiessa pystyttävä osittamaan erillisiksi yksiköiksi, jotka ovat riittävän itsenäisiä toimimaan erillisissä tietokoneissa. Sovellusten osittaminen on hajautetuissa järjestelmissä erittäin tärkeää, sillä järjestelmä voidaan osittaa ainoastaan jos sovellukset voidaan osittaa ja näin sovellusten osittamisesta tulee avainasia [10]. Osittaminen tarkoittaa käytännössä sovelluksen rakenneosien konkreettista jakamista

omiksi ajokelpoisiksi ohjelmikseen hajautetun järjestelmän eri tietokoneita varten, ellei käytettävä toteutusalusta tarjoa tarkoitukseen hienovaraisempaa ratkaisua.

Eräs tällainen ratkaisu on käyttää objekteja hajautuksen perusyksikköinä. Ratkaisu on toimiva, sillä objektit soveltuvat tähän tarkoituksen hyvin [9]. Paitsi että objektit ovat objektikeskeisessä ohjelmoinnissa ohjelman luonnollisia rakenneyksiköitä, ne myös jakautuvat helposti kahteen erilliseen osaan. Objektit nimittäin sisältävät sekä itsenäisen käyttöliittymän että tätä vastaavan käyttöliittymästä erillään olevan toiminnallisuuden sisältävän osan. Sovelluksia ositettaessa käyttöliittymä voidaan liittää objektin palveluita (operaatioita) käyttävään sovelluksen osaan ja varsinainen toiminnallisuus puolestaan johonkin muuhun mielekkääseen järjestelmän osaan, jossa toiminnallisuus (kutsuttavat objektin operaatiot) halutaan varsinaisesti suorittaa. Ositettu objekti vaatii toimiakseen käyttöliittymä- ja toteutusosan lisäksi näitä yhdistävää ylimääräistä toiminnallisuutta, mutta tästä huolehtii tarkoitukseen suunniteltu totetusalusta, eikä tämä objektille ylimääräinen toiminnallisuus yleensä näy objektia käyttävälle sovellukselle lainkaan. Osia yhdistävään toiminnallisuuteen kuuluu yleensä vähintään palvelupyyntöjen (ja niihin liittyvien mahdollisten parametrien) sekä operaation tuloksen (tai mahdollisen virheen) välittäminen objektin käyttöliittymän ja toteutusosan välillä.

Vaikka osiin jaettuja objekteja sekä niihin liittyviä rakenneosia on edellä esitelty niiden toiminnan ymmärtämistä helpottavilla termeillä, aihepiiriin liittyvä terminologia on kirjallisuudessa melko vakiintunutta ja sitä tulisi käyttää aina kuin mahdollista. Objekteja joita on aiemmin kutsuttu ositetuiksi tai jaetuiksi, kutsutaan yleisesti *hajautetuiksi objekteiksi (distributed object).* Hajautettu objekti on objektiparadigmaa noudattava ohjelmoinnin perusyksikkö, jonka käyttörajapinta ja toteutusosa on sijoitettavissa tietojärjestelmän eri osiin. Hajautetun objektin käyttöliittymän sisältävää osaa kutsutaan *asiakaskannaksi (stub)* ja käyttöliittymää vastaavaa varsinaista toiminnallisuutta sisältävää osaa puolestaan *palvelinrungoksi (skeleton).* Käytäntö yleensä vaatii että kummastakin osasta muodostetaan erikseen käännöskelpoinen luokka, jotta osat voidaan sijoittaa eri paikkoihin. Osittamatonta objektia kutsutaan hajautettujen järjestelmien yhteydessä toisinaan *paikalliseksi objektiksi* (local object) sekaannusten välttämiseksi. Kuva 3 esittää hajautettua objektia.



Kuva 3. Hajautettu objekti

Objektien käyttö ei kuitenkaan ole ainoa tapa osittaa sovellukset erottamalla ohjelmien perusyksiköiden (objekti, kirjasto, metodi, aliohjelma jne.) käyttöliittymä ja toteutusosa toisistaan. *Remote procedure call (RPC)* on objektien kulta-aikaa edeltävän aikakauden lähestymistapa hajautuksen toteutukseen ja se sopii hyvin proseduraalisia ohjelmointikieliä käyttäviin tietojärjestelmiin. RPC:nä pidetään yleisesti joukkoa menetelmiä, joille on yhteistä aliohjelmakutsujen välittäminen paikasta toiseen kommunikaatioverkon välityksellä. Aivan kuten objekteihin pohjautuvat menetelmät, myös RPC menetelmät perustavat toimintansa käännöskelpoiseen asiakaskantaan ja palvelinrunkoon. Lukuun ottamatta objektien ominaisuutta tallentaa tietoa attributteihinsa, suoraviivaisesti toteutettu objekteihin pohjautuva lähestymistapa ei juurikaan eroa toiminnallisesti RPC menetelmistä.

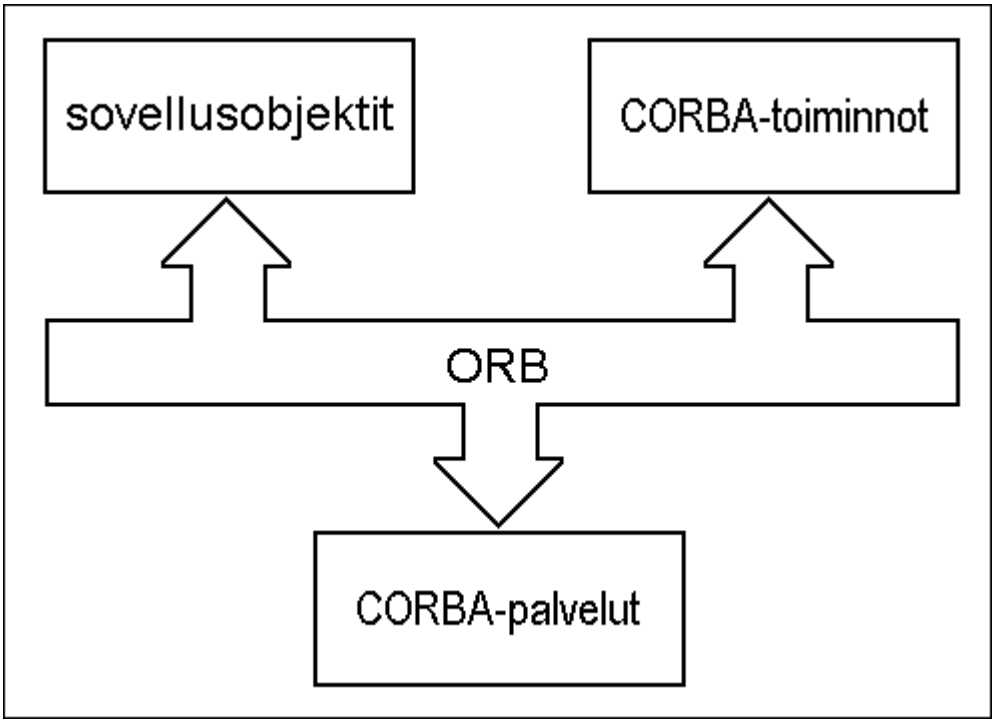
Objektikeskeisyyden vaikutus hajautettuihin järjestelmiin ei kuitenkaan rajoitu vain yllä mainitun kaltaiseen, objektien eroitetun käyttöliittymän ja toteutusosan (abstraktin tietotyypin) suoraviivaiseen hyödyntämiseen. Edellä kuvatun kaltaista lähestymistapaa kutsutaankin hajautetuista järjestelmistä puhuttaessa objektiperustaiseksi. Lähestymistapoja joissa hyödynnetään abstraktin tietotyypin lisäksi myös muita objektiparadigman oleellisimpia piirteitä, kutsutaan objektiorientoituneiksi lähestymistavoiksi hajautukseen. Oleellisimmat piirteet jotka erottavat objektiperustaisen järjestelmän objektiorientoituneesta ovat periytyminen, monimuotoisuus sekä ajonaikainen sidonta [11]. Näillä piirteillä saavutettavat edut eivät kuitenkaan liity suoranaisesti hajautuksen toteutukseen tai sen hallintaan, jonka vuoksi ne ovat tämän tutkielman aihepiirin ulkopuolella.

3 CORBA-standardi

Common object request broker architechture (CORBA) [12] tunnetaan yleisesti *Object Management Group:in (OMG)* alun perin hajautettujen järjestelmien sekalaisia menetelmiä yhdistäneenä teollisuusstandardina. Nykyisin CORBA on kohtuullisen suosittu hajautettujen järjestelmien toteutusympäristöjä kuvaava standardi, jolla on lukuisia (yli 100 kappaletta) toteutuksia. Toisin kuin useat edeltäjänsä (esimerkiksi RPC), se tarjoaa yhtenäisen pohjan useita erilaisia käyttöjärjestelmiä sekä ohjelmointikieliä samanaikaisesti sisältäviin ympäristöihin. CORBA toimii tällaisessa heterogeenisessä hajautetussa järjestelmässä kommunikaatio- ja tulkkauskanavana, joka mahdollistaa toistensa kanssa yhteensopimattomien osien yhteistoiminnan. Standardin ylevänä tarkoituksena on mahdollistaa minkä tahansa käyttöjärjestelmän tai ohjelmointikielen liittäminen osaksi CORBA-järjestelmää. Käytännössä edellä mainittuja voidaan liittää osaksi järjestelmää vain silloin, kun ohjelmointikieli kuuluu tuettuihin kieliin ja käyttöjärjestelmälle on olemassa ORB (jota käsitellään hieman myöhemmin). Tässä luvussa tutustumme CORBA-standardiin sekä sen keskeisiin rakenneosiin ja käsitteisiin.

3.1 CORBA ja OMA

CORBA-standardi pohjautuu OMG:n *Object Management Architechture (OMA)* malliin, joka määrittelee standardin perustan muodostavat rakenneosat. Nämä rakenneosat ovat *OMG-objektimalli (OMG Object Model)* sekä *OMA-viitemalli (OMA Reference Model).* OMG-objektimalli kuvaa semantiikan, jonka avulla objektien ulospäin näkyviä piirteitä voidaan määritellä toteutusriippumattomasti. OMA-viitemalli puolestaan luettelee ja kuvailee OMA:n muodostavat rakenneosat. Näihin osiin kuuluvat OMA:n komponentit, rajapinnat sekä yhteyskäytännöt. Kuva 4 esittelee OMA-viitemalliin perustuvat CORBA komponentit.



Kuva 4. OMA-viitemalliin perustuvat CORBA komponentit.

OMA:n kannalta keskeisin kuvan komponenteista on *Object Request Broker (ORB)*, joka toteuttaa objektien välisen kommunikoinnin objektipohjaisessa hajautetussa järjestemässä. ORB on myös CORBA:n kannalta OMA:n keskeinen komponentti, sillä nimenomaan CORBA-standardi määrittää ohjelmointirajapinnat OMA-viitemallin ORB-komponentille. Täten ORB-komponenttia voidaan pitää CORBA-standardin totetuksena, joskin se toteuttaa vain osan standardin sisällöstä. Toinen erittäin keskeinen komponentti on *CORBA-palvelut (CORBA services)*, joka on eräänlainen hajautettujen järjestelmien standardikirjasto. CORBA-palvelut sisältävät hajautettuihin objekteihin perustuvan järjestelmän yleiskäyttöisiä palveluita, joiden avulla pyritään vähentämään samojen perustavanlaatuisten ohjelmistoratkaisuiden jatkuvaa uudelleen keksimistä, sekä lisäämään sovellusten siirrettävyyttä ratkaisuiden yhdenmukaistuksen kautta. CORBA-palveluihin viitataankin toisinaan *rakennuspalikkoina (building blocks)*, joiden avulla voidaan rakentaa hajautettuja sovelluksia ja tätä kautta hajautettuja järjestelmiä. Ne myös auttavat pitämään *CORBA-ytimen (CORBA core)* mahdollisimman pienikokoisena ja yksinkertaisena CORBA-standardin jatkuvasti laajetessa. Nämä palvelut on kuvattu lyhyesti liitteessä 1.

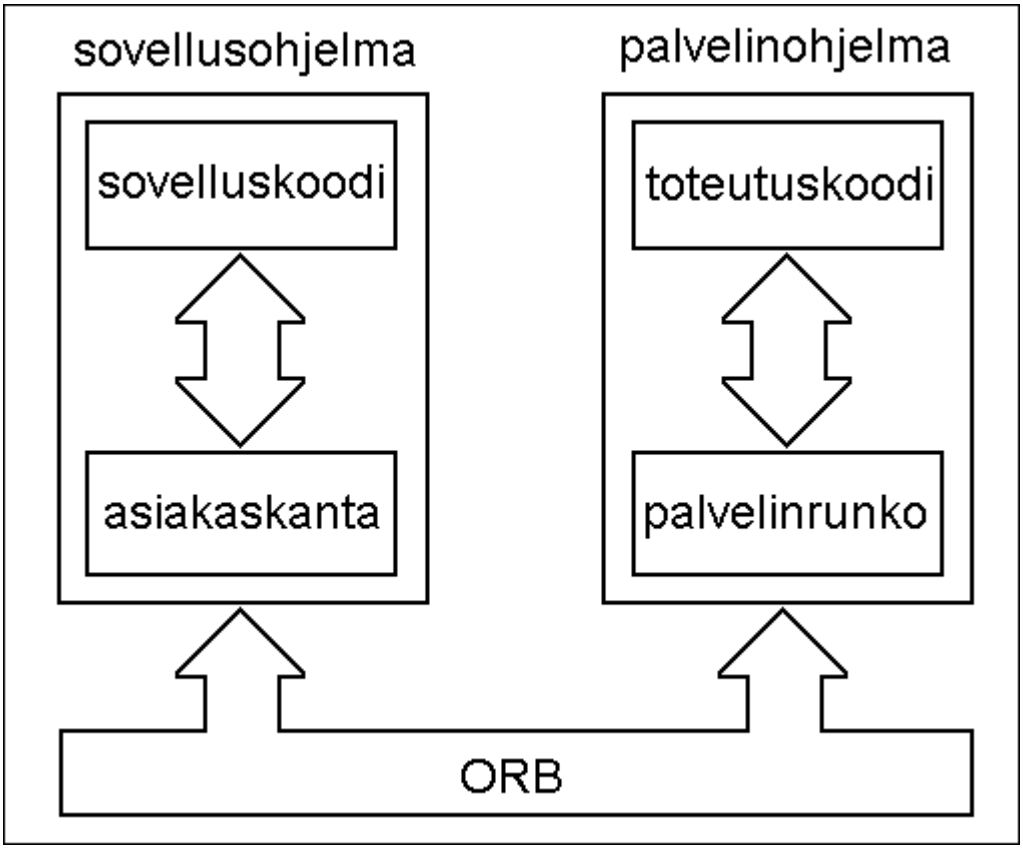
Siinä missä CORBA-palvelut tarjoavat valmiiksi toteutettuja perustoimintoja yksittäisten objektien käyttöön, *CORBA-toiminnot (CORBA facilities)* tarjoavat korkeamman asbtraktiotason toiminnallisuutta. CORBA-palvelut jakautuvat kahteen osaan, joita kutsutaan horisontaalisiksi sekä vertikaalisiksi palveluiksi. Horisontaaliset palvelut sisältävät loppukäyttäjiin orientoitunutta toiminnallisuutta kuten käyttöliittymiin, tiedon mallintamiseen, järjestelmänhallintaan sekä tehtävienhallintaan liittyviä palveluita. Vertikaaliset palvelut sen sijaan on tarkoitettu kokoamaan yhteen korkean tason toiminnallisia määrittelyitä (ns. business-logiikkaa), jotka liittyvät kapeisiin sekä tarkasti määriteltyihin sovellusalueisiin. Tällaisia ovat esimerkiksi terveydenhuollon, telekommunikaation tai vaikkapa sähköisen kaupankäynnin sovellusten käsitteistö, sekä suoritettavien toimintojen rakenteet. CORBA-palveluita toteutetaan s*ovellusobjekteina (application objects)*, jotka ovat sovelluksiin upotettavia hajautettuja objekteja. Ohjelmoijat voivat käyttää näitä objekteja sellaisinaan tai muokata niistä periyttämisen kautta sopivia objekteja (tai oikeastaan luokkia). Sovellusobjektien ei tarvitse olla yksittäisiä objekteja, vaan ne voivat esimerkiksi edustaa kokonaista *sovelluskehystä (application framework)*.

3.2 ORB

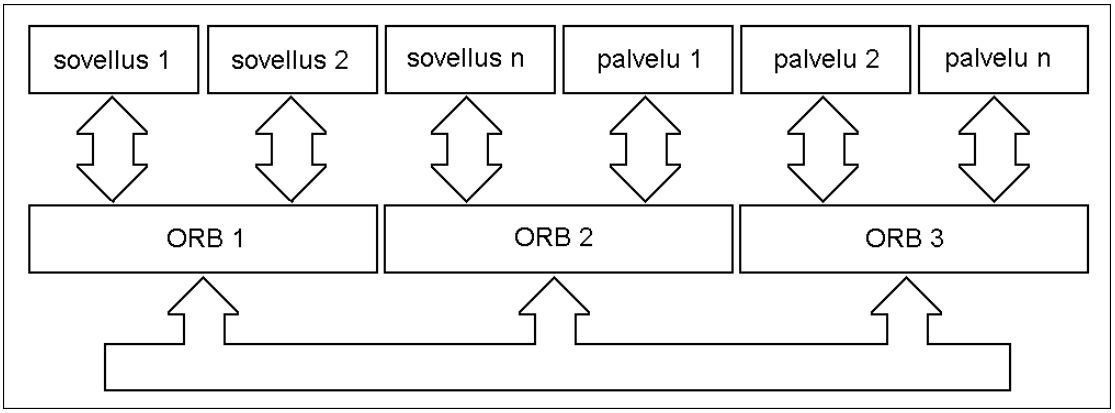
ORB on CORBA:an perustuvan hajautetun järjestelmän kulmakivenä toimiva mekanismi, jonka tehtävänä on välittää operaatioita tai attribuutteja koskevat palvelupyynnöt asiakaskannalta palvelinrungoille, sekä palauttaa mahdolliset tulokset takaisin kutsut tehneelle asiakaskannalle. ORB toteuttaa tämän asiakaskannan ja palvelinrungon välisen kommunikoinnin siten että hajautettu objekti käyttäytyy sen käyttäjän kannalta aivan kuten sen paikallinen vastine käyttäytyisi [13]. Tällaista *läpikuultavaa (transparent)* kommunikaatiota käytettäessä objekteja käyttävä sovellus erottaa hajautetun objektin paikallisesta periaatteessa vain ylimääräisestä viiveestä, jota yleensä esiintyy hajautettuja objekteja käytettäessä. ORB:in toimintaa voidaan tarkastella tutkimalla kuinka hajautettua objektia vastaavat konkreettiset luokat luodaan järjestelmään, sekä kuinka viestit kulkevat näiden luokkien välillä.

Ohjelmoija muodostaa hajautettua objektia vastaavan luokan kirjoittamalla ensin IDL-kielellä (jota käsittelemme myöhemmin) kuvauksen objektin rajapinnoista ja ajamalla

tämän kuvauksen IDL-kääntäjän läpi. Tuloksena tästä IDL-käännöksestä on asiakaskannan ja palvelinrungon lähdekoodit. Tässä vaiheessa palvelinrungon lähdekoodissa metodit ovat sisällöltään tosin vielä tyhjiä. Nämä metodit ohjelmoijan tulee täydentää ohjelmakoodilla, ennen kuin palvelinrungosta saadaan luokka, joka voidaan kääntää ja sijoittaa jonkin sopivan ORB:in alaisuuteen toteuttamaan haluttua palvelua. Nyt järjestelmään rakennettava sovellus voi käyttää asiakaskantaa kuten paikallista objektia, sillä asiakaskanta käännetään sovellukseen kuten muutkin sovelluksen luokat. Kutsuessaan asiakaskannan metodeja tai lukiessaan sen attribuuttien arvoja, sovellus tekee palvelupyynnön jonka asiakaskanta välittää ORB:lle asianmukaisen rajapinnan kautta. ORB puolestaan välittää tämän palvelupyynnön palvelun toteutuksesta vastaavalle *toteutusobjektille*, mikäli tämä objekti sijaitsee sen itsensä alaisuudessa. Muussa tapauksessa se välittää palvelupyynnön (yleensä verkon yli) toiselle, kyseisen objektin omistavalle ORB:ille, joka puolestaan välittää palvelupyynnön itse objektille. Saadessaan palvelupyynnön, toteutusobjekti ajaa palvelua vastaavan ohjelmakoodin ja palauttaa vastauksen takaisin pyynnön tekijälle samaa reittiä pitkin jota itse pyyntökin kulki. Lopulta vastaus näkyy kutsun tehneelle sovellukselle kutsutun metodin palautusarvona. Kuva 5 havainnollistaa kommunikointiin liittyviä rakenneosia.



Eräs mielenkiintoinen tapa nähdä tämä kokonaisuus on tarkastella yhteenkytkettyjä ORB:eja *ohjelmistoväylänä (software bus)*, joka yhdistää hajautetun järjestelmän objekteja toisiinsa. Sovellusohjelmat voivat käyttää lähes mielivaltaisesti väylään kytkettyjä palveluita (objekteja) väylän piilottaessa kommunikointiin, palveluiden toteutukseen ja sijaintiin liittyvät yksityiskohdat. Väylän käyttökelpoisuutta lisää merkittävästi CORBA:n nimipalvelu, joka mahdollistaa palveluiden sijaintitietojen tallentamisen keskitettyyn tietovarastoon, jonka sisältöä voidaan muuttaa milloin tahansa järjestelmän toiminnan siitä häiriintymättä. Näin väylään kytkettyjen palveluiden sijaintia voidaan muuttaa ‘lennossa’ ja vieläpä palveluita toteuttavia järjestelmän osia uudelleen kääntämättä. Kuva 6 esittää ohjelmistoväylä-tyyppistä näkökulmaa useita ORB:eja sisältävään hajautettuun järjestelmään.



Kuva 6. Ohjelmistoväylä

3.3 IOR

Voidakseen osoittaa toteutusobjektille palvelupyyntöjä, sovellus tarvitsee yksiselitteisen osoitteen jonka perusteella se voi kohdentaa pyynnöt oikealle objektille. Tämän osoitteen tulee sisältää riittävästi informaatiota sekä toteutusobjektin paikallistamiseksi, että sen identifioimiseksi. Objekti on nimittäin kyettävä ensin paikallistamaan järjestelmään kytketyiltä palvelimilta, jonka jälkeen se on pystyttävä identifioimaan muiden samaa luokkaa edustavien objektien joukosta. CORBA:ssa objektien osoitteistus toteutetaan tällä hetkellä IIOP-yhteyskäytäntöön pohjautuvien objektien välisten viitteiden, *IOR:eiden (Interoperable Object*

Reference) avulla. IOR on CORBA:n sisäinen käytäntö objektiviitteiden toteuttamiseksi, eikä CORBA tarjoa toiminnallisuutta jonka avulla ohjelmoija voisi tutkia IOR:in sisältämää tietoa. Tämän vuoksi toteutettaessa järjestelmään viittauksia tarvitsevia ohjelmia, IOR:it muunnetaan käytettävän ohjelmointikielen mukaisiksi osoitinrakenteiksi. Ohjelmoijan ei siis tarvitse käyttää IOR:eita suoraan, toteuttaessaan järjestelmään sovelluksia tai palveluita.

3.4 GIOP ja IIOP

IOR on CORBA:n sisäinen käytäntö objektiviitteiden toteuttamiseksi. Se olisi täysin riittävä viittausmekanismina, mikäli järjestelmässä olisi vain yksi ORB-ohjelmisto (tai tarkalleen ottaen vain yksi, yksiselitteinen osoiteavaruus). Käytännössä CORBA-perustaisessa hajautetussa järjestelmässä on kuitenkin oltava käytössä useita ORB:eja samanaikaisesti. Tämän vuoksi ORB:ien väliseen kommunikointiin on määritelty oma *GIOP (General Inter-ORB Protocol)* yhteyskäytäntönsä. Toisin kuin useimmat yhteyskäytännöt, GIOP on abstrakti määritelmä, jonka perusteella ei voida suoraan toteuttaa toimivaa kommunikointimekanismia. Käytännössä GIOP:in sijasta käytetäänkin jotain sen täydennettyä versiota kuten *IIOP (Internet Inter-ORB)* yhteyskäytäntöä, jossa fyysiseksi yhteyskäytännöksi on määritelty TCP/IP. Avoimeksi GIOP:in määritelmässä ei nimittäin ole jätetty kuin fyysinen yhteyskäytäntö. Vaikka IIOP on vain yksi monista yhteyskäytännöistä, se on erittäin suosittu käyttämänsä TCP/IP:n suosion vuoksi ja näin myös erittäin keskeisessä asemassa CORBA ympäristössä.

3.5 IDL

Joustavien sovellusten kehittäminen heterogeenisessä toteutusympäristössä vaatii objektien käyttöliittymien huolellista eroittamista niiden toteutusosasta [14]. Tätä tarkoitusta varten CORBA sisältää *IDL-kielen (Interface Definition Language)*, joka on objekti-orientoitunut määrittelykieli. Sen avulla hajautettujen objektien käyttöliittymät voidaan määritellä, sitomatta määritelmää mihinkään tiettyyn toteutuskieleen tai –tapaan. IDL-kieli sisältää vain ominaisuuksia jotka ovat tarpeellisia objektien käyttöliittymien (luokkien kuvausten) määrittämiseksi. Näihin ominaisuuksiin lukeutuu yksittäisen luokan määrityksen tuki, moduulit joilla

määrityksiä voidaan koota kokonaisuuksiksi, poikkeuksien tuki, yksittäis- ja moniperiminen sekä tuki kattavalle joukolle erilaisia tietotyyppejä. Tuettuja tietotyyppejä ovat perustietotyypit sekä näistä johdetut tarkentavat tietotyypit, kuten vakiot, taulukot, rakenteiset ja geneeriset tietotyypit sekä tyyppimääritelmät. Kielestä puuttuu useita ohjelmointikielissä esiintyviä ominaisuuksia, joita ei määrittelyssä tarvita. Tällaisia ovat muun muassa kontrollirakenteet sekä osoittimet. Kielestä puuttuu myös monia objekti-orientoituneista kielistä tuttuja ominaisuuksia, vaikka sen esikuvana on ollut C++ kieli. Tällaisia ominaisuuksia ovat *jäsenmuuttujat (data members), operaatioiden kuormittaminen (operator overloading)* sekä *rakentajat (contructors)* ja *tuhoajat (destructors)*. IDL-kieli onkin suunniteltu varsin yksinkertaiseksi ja helposti opittavaksi. Kielen yksinkertaisella rakenteella on ollut tarkoitus tukea mahdollisimman selkeiden käyttöliittymien kirjoittamista.

Pelkkänä määrittelykielenä IDL-kielellä ei voida toteuttaa ohjelmia ja siksi OMG onkin laatinut IDL-kielen ja olemassa olevien ohjelmointikielten välisiä *kartoituksia (mappings)*. Nämä kartoitukset mahdollistavat IDL-kielellä kirjoitetun käyttöliittymäkuvauksen muuntamisen totutuksen mahdollistaville ohjelmointikielille, kuten esimerkiksi C++ tai Java-kielelle. Muunnos IDL-kuvauksesta halutulle ohjelmointikielelle tehdään IDL-kääntäjän avulla ja tuloksena tästä IDL-käännöksestä on käännöskelpoista ohjelmointikielen ohjelmakoodia. Näin IDL-kielellä määritellyt käyttöliittymäkuvaukset voidaan aina tarvittaessa siirtää uuteen ympäristöön ohjelmointikieli- ja laitteistoriippumattomasti, kunhan käytettävä ohjelmointikieli on kartoitettu IDL-kieleksi ja kyseisen laitteiston toimintaympäristö tukee tätä kieltä. Tämän vuoksi IDL-kieli onkin erityisen keskeisessä asemassa CORBA-standardissa. CORBA:n ohjelmointikieli- ja laitteistoriippumattomuus perustuu nimittäin vahvasti standardoituihin rajapintakuvauksiin, joissa rajapintojen tarjoamien palveluiden käyttöliittymät on tehokkaasti eristetty toteutuksistaan määrittelemällä kyseiset rajapinnat IDL-kuvauksina. Näin standardin toteutukset ovat siirrettävissä ympäristöstä toiseen ja uudet toteutukset ovat rajapintojen suhteen toistensa kanssa yhteensopivia.

3.6 Objektisovitin

Objektisovitin (object adapter) on ohjelmoijille yleensä täysin näkymätön rakenneosa, jonka vastuulla on toteutusobjekteihin liittyvistä järjestelmätason toiminnoista huolehtiminen. Näihin toimintoihin kuuluu toteutusobjektin käytön mahdollistavat toimenpiteet kuten objektien luominen (tai oikeastaan objektiviitteiden luominen), toteutusobjektien aktivointi ja käytöstä poistaminen sekä järjestelmän sisäisessä muodossa olevien palvelupyyntöjen tulkitseminen toteutusobjektien ymmärtämään muotoon. Lisäksi objektisovitin toimii eräänlaisena siltana, jonka kautta toteutusobjekti pääsee käsiksi ORB:in tarjoamiin palveluihin. Tällaisia palveluita voivat olla esimerkiksi CORBA-palvelut.

OMG määritteli objektisovittimen alun perin rakenneosaksi, joka eristää palvelun toteuttavan objektin ja ORB:in voimakkaasti toisistaan. Ajatuksena oli mahdollistaa objektin toteutukselle mielivaltainen toimintaympäristö, määrittelemällä objektisovitin vaihdettavaksi aina tarpeen mukaan. Näin objektin toteutus voi olla tiedostosta ajattavan ohjelman sijaan vaikkapa tietokantaan tallennettu proseduuri, jolle ORB välittää palvelupyynnön tähän tarkoitukseen kirjoitetun objektisovittimen kautta. Objektisovitin vastaa nimensä mukaisesti yleisesti tunnettua sovitin-rakennemallia [15],[16]. Objektisovittimien kirjoittaminen ei käytännössä ole kuitenkaan aivan yksinkertaista ja CORBA:an perustuvaa järjestelmää rakentaessaan, suunnittelijat tyytyvät lähes poikkeuksetta käyttämään tavanomaisia objektien toteutus- ja ajoympäristöjä, sekä CORBA:n näihin ympäristöihin tarjoamia valmita objektisovittimia. Yleisin käytetty objektisovitin on epäilemättä *BOA (Basic Object Adapter)* tai *POA (Portable Object Adapter)* riippuen CORBA-standardin versiosta. Nämä objektisovittimet ovat toiminnaltaan hyvin samanlaisia. POA on oikeastaan vain BOA:n uudempi versio, jossa OMG on korjannut eräitä BOA:n siirrettävyyteen liittyviä ongelmia. Alkuperäiskielisten nimiensä sekä ominaisuuksiensa perusteella BOA:a voisikin luonnehtia perusobjektisovittimeksi ja POA:a siirrettäväksi perusobjektisovittimeksi.

3.7 Luokkarajapintojen staattinen sidonta (SII)

Ohjelmia kirjoittaessaan ohjelmoija määrittelee monesti kaikki käytettävät luokat ennen ohjelman kääntämistä. Tällaista tapaa sitoa luokkien rajapinnat toisiinsa käännösaikaisesti kutsutaan *staattiseksi sidonnaksi (static binding)* ja IDL-käännöksestä saadut asiakaskanta- ja palvelinrunkoluokat sidotaan muihin käännöksessä käytettäviin luokkiin juuri näin. Asiakaskannan ja palvelunrungon rajapinnat tunnetaankin standardissa nimellä *SII (statical invocation interface)* eli staattisina palvelupyyntörajapintoina. CORBA tarjoaa SII:n lisäksi myös mahdollisuuden sitoa luokkarajapinnat vasta ajonaikaisesti SII:tä vastaavan dynaamisen sidonnan avulla.

3.8 Luokkarajapintojen dynaaminen sidonta (DII)

Luokkien kutsurajapintojen käännösaikainen sitominen SII:n avulla on melko suoraviivaista, selkeää ja helppoakin. Kutsurajapintojen käännösaikainen sitominen ajaakin asiansa useimmissa tapauksissa, mutta on olemassa myös tilanteita joissa tällainen menettely ei ole mielekästä. Tällainen tilanne voi tulla eteen esimerkiksi rakennettaessa sovellusta, joka ei varsinaisesti käsittele saamaansa tietoa, vaan siirtää sitä eteenpäin jollekin tietylle tiedon laadun perusteella valitulle vastaanottajalle. Vastaanottaja voi nimittäin olla objekti, jonka IDL-kuvausta eikä näin ollen IDL-käännöksestä saatavaa asiakaskantaluokkaakaan, ole ollut olemassakaan välittäjää käännettäessä. Tällöin välittäjä ei luonnollisesti pysty lähettämään tietoa vastaanottajalle, sillä se voi kommunikoida (staattista sidontaa käytettäessä) tämän kanssa vain asiakaskannan kautta. CORBA ratkaisee tämän kaltaiset ongelmat mahdollistamalla luokkarajapintojen ajonaikaisen eli dynaamisen sidonnan.

DII (Dynamic Invocation Interface) rajapinta mahdollistaa luokkarajapintojen ajonaikaisen sitomisen. Sen kautta sovellusohjelma voi päästä käsiksi haluamiinsa objekteihin, ilmoittamalla tämän objektin luokan nimen ASCII-merkkeinä. Tämän tiedon perusteella DII palauttaa kutsujalle viitteen kyseistä luokkaa edustavaan objektiin. Ongelmina DII:ssä ovat SII:tä huomattavasti tehottomampi toiminta, dynaamisen ohjelmointirajapinnan käytön monimutkaisuus sekä tyyppitarkistusten siirtyminen käännösaikaisesta ajonaikaiseksi. Sovellusohjelmien käyttöön tarkoitetun

DII-ohjelmointirajapinnan lisäksi, CORBA sisältää myös vastaavan palvelinohjelmien käyttöön tarkoitetun ohjelmointirajapinnan. *DSI (Dynamic Skeleton Interface)* mahdollistaa palveluiden toteuttamisesta vastaavien toteutusobjektien ajonaikaisen sitomisen, aivan kuten DII mahdollistaa sovellusohjelmien ajonaikaisen sitomisen. Valitettavasti DSI kärsii myös samoista ongelmista kuin DII. Yhdessä DII ja DSI muodostavat kuitenkin täysimittaisen ajonaikaisen tyypitysjärjestelmän, jolla voidaan ratkaista käännösaikaiseen sitomiseen liittyviä ongelmia. DII- ja DSI-ohjelmointirajapintojen toimintaa täydentää *IRF (Interface Repository)* komponentti, joka toimii luokkarajapintakuvausten keskitettynä tallennus- ja hakupaikkana. Sitä voidaan pitää eräänlaisena hajautetun järjestelmän luokkamääritysten keskusvarastona. Tärkemmät IRF-komponentin tarjoamat palvelut ovatkin luokkarajapintavaraston sisällön selaaminen sekä yksittäiseen luokkarajapintaan liittyvien täydellisten tietojen tarjoaminen.

3.9 Pysäyttävät ja ei-pysäyttävät kutsut

Kutsutapaa jossa kutsuja jää odottamaan kutsuttavan operaation suorituksen päättymista, sanotaan *pysäyttäväksi kutsuksi (blocking call)*. Tämä kutsutapa on epäilemättä yleisimmin käytetty lukemattomista eri syistä. CORBA:ssa kutsuista tulee automaattisesti pysäyttäviä, mikäli kutsut tehdään kutsujaan staattisesti sidotun asiakaskannan kautta. Käytettäessä dynaamista sidontaa, vastaavanlainen pysäyttävä kutsu saadaan aikaiseksi käyttämällä invoke() operaatiota. Kutsuttavien operaatioiden suorituksen päättymisen odottaminen ei kuitenkaan ole aina paras mahdollinen tapa edetä kutsuvan ohjelmakoodin suorituksessa. Tämän vuoksi CORBA tarjoaa myös *ei-pysäyttäviä (non-blocking)* kutsutapoja, jotka mahdollistavat kutsuvassa koodissa etenemisen odottamatta kutsuttavien operaatioiden suoritusten päättymistä. Tällaisten kutsutapojen käyttö johtaa yleensä *rinnakkaisuuteen (parallerism)* ohjelmakoodin suorittamisessa.

Yksisuuntainen kutsu (oneway) on yksinkertaisin CORBA:n ei-pysäyttävä kutsutapa. Tätä kutsutapaa käytettäessä ohjelmakoodin suoritusta jatketaan välittömästi operaation kutsun jälkeen, eikä kutsuja jää odottamaan operaation suorituksen lopputulosta. Tällaisen operaation onnistunut suoritus (tai suoritus yleensäkään) ei ole täysin varmaa, sillä kutsuja ei voi lukea lainkaan operaation palautusarvoa.

Yksisuuntaiset kutsut toteutetaankin *yritetään parasta (best-effort)* semantiikalla, jossa järjestelmä yrittää parhaansa operaation suorittamiseksi, muttei takaa onnistunutta suoritusta [17]. Tällainen kutsu tehdään käytännössä määrittelemällä ensin kutsuttava operaatio yksisuuntaiseksi luokan IDL-kuvaksessa, jonka jälkeen itse kutsuminen tehdään DII:n kautta send_oneway() operaatiolla, invoke() operaation sijasta. Yksisuuntaiseksi määritellyn operaation IDL-kuvaukseen ei voi sisällyttää paluuarvoa, eikä myöskään OUT- tai INOUT-tyyppisiä parametrejä. Yksisuuntaisia kutsuja ei myöskään voi tehdä lainkaan käytettäessä staattista sidontaa.

Myöhästetty (a)synkroninen kutsu (deferred (a)syncronous) on yksisuuntaista kutsua monipuolisempi, mutta samalla myös monimutkaisempi tapa toteuttaa ei-pysäyttävä kutsu. Toisin kuin yksisuuntainen kutsu, myöhästetty kutsu mahdollistaa paluuarvon sekä OUT- ja INOUT-tyyppisten parametrien käytön. Sitä voidaan käyttää myös pysäyttävän kutsun tavoin, mikäli ohjelmoija näin haluaa. Kutsun valinnaisen toimintatavan mahdollistaa se, että vaikka kutsuttava operaatio suoritetaan heti kutsun jälkeen, sen tulosta ei palauteta kutsujalle vaan järjestelmä tallentaa tuloksen myöhempää käyttöä varten. Näin kutsujan ei tarvitse jäädä odottamaan tulosta, vaan se voi tiedustella sitä jonain sopivampana ajankohtana. Tämän ominaisuuden avulla myöhästettyjä kutsuja voidaan käyttää estävinä tiedustelemalla tulosta erikseen jokaisen operaatiokutsun jälkeen, jolloin kutsuvan ohjelmakoodin suoritus pysähtyy aina operaation suorituksen ajaksi. Ei-estävinä kutsuja voidaan käyttää tiedustelemalla tuloksia vasta useamman operaation kutsumisen jälkeen, jolloin kutsuvaa ohjelmaakoodia suoritetaan pysähtymättä aina tiedusteluun saakka. Myöhästettyjen kutsujen monimutkaisuus on seurausta paitsi tulosten tiedustelun tarpeesta, niin myös pakosta käyttää DII:tä. Myöhästettyjä kutsuja voidaan nimittäin suorittaa toistaiseksi vain sen kautta.

4 Hajautettujen järjestelmien tekniset avainasiat

Rakennettaessa konkreettisia tietojärjestelmiä, törmätään poikkeuksetta joukkoon erilaisia ongelmia. Nämä ongelmat ovat niin yleisiä ja niiden lukumäärä on niin suuri, että ongelmien luokittelu vakavuuden perusteella on käytännössä lähes pakollista. Näin on ainakin silloin, kun tietojärjestelmät aiotaan saada joskus valmiiksikin. Tarve luokitella ongelmia vakavuusasteen mukaan, on olemassa epäilemättä myös rakennettaessa hajautettuja järjestelmiä. Niissä esiintyvät ongelmat eivät nimittäin ole ainakaan helpommin hallittavissa, kuin vastaavien keskitettyjen järjestelmien ongelmat. Erityisen keskeisessä asemassa olevia, teknisiin asioihin liittyviä hajautettujen järjestelmien ongelma-alueita kutsutaan tässä tutkielmassa teknisiksi avainasioiksi. Tekniset avainasiat ovat siis erityisen keskeisessä asemassa olevia hajautetun järjestelmän rakentamiseen liittyviä teknologia luoteisia ongelma-alueita. Ne ovat erityisen keskeisessä asemassa, sillä niissä esiintyvät ongelmat ovat erittäin yleisiä hajatettujen järjestelmien keskuudessa. Champine määrittelee tällaisiksi avainasioiksi tietoturvan, samanaikaisuudenhallinnan, yhteensopivuuden (tai oikeastaan muutoksen kestävyyden), luotettavuuden, tehokkuuden ja monimutkaisuuden hallinnan [10]. Hän ei kuitenkaan ole ainoa, joka korostaa tiettyjen osa-alueiden merkitystä hajautettujen järjestelmien rakentamisessa. Myös Coulouris, Dollimore ja Kindberg esittävät tavoitteita, joihin hajautettua järjestelmää rakennettaessa tulisi pyrkiä [3]. Nämä tavoitteet ovat melko lailla yhteneviä Champinen määrittelemien teknisten avainasioiden kanssa. Myös lukuisat muut tutkijat vaikuttavat olevan kohtuullisen yksimielisiä siitä, että edellä mainitut asiat ovat erittäin oleellisia hajautettuja järjestelmiä rakennettaessa. Koska hajautetun järjestelmän toteutusvälineen tulisi mielestäni tarjota tukea järjestelmän rakentajille vähintään kaikkein oleellisimmilla ongelma-alueilla, käytän tässä tutkielmassa Champinen määrittelemiä teknisiä avainasioita CORBA-standardin tutkimiseen. Tekniset avainasiat soveltuvat tähän tarkoitukseen hyvin, sillä tarkastelemalla CORBA-standardin hajautettujen järjestelmien yleisimpiin ongelma-alueisiin tarjoamia ratkaisuja, saadaan esiin standardin merkittävimmät ominaisuudet tai niiden puutteet. Seuraavissa kohdissa tarkastelemme näitä ratkaisuja ongelma-alueittain.

4.1 Tietoturva

Tietoturvaominaisuuksilla tarkoitetaan tässä tutkielmassa toiminnallisuutta, jonka avulla estetään järjestelmän ja siihen kuuluvien osien luvaton käyttö sekä näiden sisältämän tieton luvaton tarkastelu, muuttaminen ja poistaminen. Yleisesti ottaen tietoturvaominaisuuksien tarkoitus on estää kaikenlainen järjestelmän epätoivottu käyttö. Käytännössä tietoturvasta ei kuitenkaan saada, eikä yritetäkään saada absoluuttista suojaa kaikkea mahdollista epätoivottua toimintaa vastaan. Sen sijaan tietoturvan taso yritetään monesti saada järjestelmän arvoa vastaavalle tasolle. Tietoturvan toteuttaminen nimittäin maksaa rahaa lisääntyneinä kehitys- ja ylläpitokustannuksina ja usein heikentää järjestelmän käytettävyyttä. Tietoturvaongelmien luonteeseen kuuluu ongelma-alueen lievä ennalta-arvaamattomuus. Tämä ilmenee muun muassa tietomurtojen kohdentumisena järjestelmän kaikkiin vähänkin julkisesti käytettävissä oleviin osiin, vaikka nämä osat tuntuisivat tietoturvan kannalta epäoleellisilta järjestelmän suunnittelu- tai toteutusvaiheessa. Asiansa tuntevat murtomiehet hyödyntävät nimittäin kaikkia käytettävissä olevia keinoja. Edellä mainitun sekä lukuisten muiden seikkojen vuoksi tietoturva on tärkeä osa nykyaikaista tietojärjestelmää. Mielestäni käytännöllisesti katsoen kaikki hyötykäyttöön tarkoitetut tietojärjestelmät tarvitsevat jonkinlaisia tietoturvaominaisuuksia. Tässä kohdassa tarkastelemme hajautettujen järjestelmien tietoturvaan liittyviä kysymyksiä sekä tutustumme CORBA:n tietoturvaominaisuuksiin.

Tietoturva hajautetuissa järjestelmissä

Hajautettujen järjestelmien tietoturvan toteuttaminen on yleensä monimutkaisempaa kuin vastaavien keskitettyjen järjestelmien tietoturvan toteutus. Tämä johtuu etupäässä siitä, että hajautetuissa järjestelmissä on potentiaalisia tietoturvaongelmia jonkin verran enemmän kuin vastaavissa keskitetyissä järjestelmissä. Hajautettujen järjestelmien tietoturvaongelmiin lukeutuu nimittäin vastaavanlaisten keskitettyjen järjestelmien tietoturvaongelmat, sekä lisäksi koko joukko tiedon ja toiminnallisuuden hajautuksesta aiheutuvia, tiedonsiirtoon liittyviä tietoturvaongelmia. Keskeisimmät näistä ongelmista liittyvät tiedon siirtämiseen julkisia verkkoja pitkin. Tiedonsiirron

yhteydessä esiintyviä uhkia ovat muun muassa tietoliikenteen salakuuntelu, muokkaus sekä luvaton uudelleenohjaus kolmannelle osapuolelle.

Tietoturva CORBA standardissa

Tietoturvaominaisuuksien tarve on huomioitu myös CORBA-standardissa. CORBA:n *tietoturvapalvelu (CORBASEC)* sisältää toiminnallisuutta, jota hyödyntämällä rakennettavan järjestelmän tietoturvan taso on mahdollista nostaa tarvittaessa hyvinkin korkeaksi. Tietoturvapalvelun pohjana toimiva Security Service specification [18] määrittelee järjestelmän tietoturvalle neljä perusvaatimusta, jotka täytettyään se on OMG:n mielestä kaupallisille järjestelmille yleisesti hyväksytyllä tasolla. Nämä vaatimukset ovat *luottamuksellisuus (confidentiality), koskemattomuus (integrity), vastuullisuus (accountability) ja saavutettavuus (availability).* Näistä luottamuksellisuus pitää sisällään järjestelmän sisältämän tiedon tarkastelun rajoittamisen ainoastaan niille käyttäjille joilla tähän on oikeus. Koskemattomuus taas pitää sisällään sen, että järjestelmän sisältämää tietoa voivat muokata ainoastaan tähän oikeutetut henkilöt ja vain sallituilla tavoilla. Tietoa tulee myös voida siirtää vain ja ainoastaan niiden henkilöiden välillä, joiden välillä kyseistä tietoa on ollut alun perin tarkoitus siirtää ja vain aiotuilla tavoilla. Vastuullisuuteen sisältyy velvollisuus saada käyttäjät tarvittaessa vastuuseen tietoturvaa loukanneista toimistaan. Erikoistapaus tästä vastuusta on on *kiistämättömyys (non-repudiation)*, jossa käyttäjää estetään kiistämästä vastuutaan esimerkiksi hävittämällä tai väärentämällä todistusaineistoa. Saavutettavuus puolestaan on vaatimus, jonka mukaan järjestelmän luvallista käyttöä ei tule voida estää vihamielisillä toimenpiteillä.

Perusvaatimusten lisäksi tietoturvapalvelun pohjana toimiva spesifikaatio määrittelee joukon käsitteitä ja ominaisuuksia, jotka toimivat varsinaisen toiminnallisuuden pohjana. Merkittävimmät näistä ovat *toimeksiantaja (principal), oikeudet (privileges), valtuudet (credentials), attribuutit (attributes)* ja *turva-alueet (security domains).* Näistä toimeksiantaja on käyttäjä, jota koskevat tiedot ovat olemassa järjestelmässä ja joka voi todistaa identiteettinsä näiden tietojen perusteella (esimerkiksi antamalla nimensä ja salasanansa tarkastettavaksi kirjautuessaan sisään järjestelmään). Toimeksiantajaan liittyy aina yksilöllinen käyttäjätunnus, jonka avulla hänet voidaan tunnistaa yksiselitteisesti. Käyttäjän lisäksi toimeksiantajana voi myös toimia sovellus

tai jokin sovelluksen osa kuten yksittäinen objekti. Oikeudet puolestaan määrittelevät mitä toimeksiantaja saa tehdä järjestelmässä, mihin ryhmään hän kuuluu, mitä roolia hän edustaa sekä mitä kykyjä ja selvityksiä hänellä on. Toimeksiantajan oikeudet määritellään käytännössä sitomalla häneen oikeusattribuutteja. Attribuutit ovat tyyppi-arvo pareja, jotka assosioidaan toimeksiantajaan ja joita säilytetään hänen valtuuksissaan. Oleellisimmat attribuutesista ovat identiteetti-attribuutit (eli identiteetti) sekä oikeusattribuutit (eli oikeudet). Näistä identiteetti-attribuuttien sisältämät arvot yksilöivät yhdessä toimeksiantajan ja oikeusattribuuttien arvot puolestaan yksilöivät sen mitä toimeksiantaja saa järjestelmässä tehdä. Erilaisia attribuutteja on Security Service spesifikaatiossa määritelty identiteetin ja oikeuksien lisäksi koko joukko (esimerkiksi käyttäjäryhmä, rooli sekä kyky) ja attribuutteja on mahdollista määritellä tarvittaessa lisää. Erilaisten attribuuttisen avulla toimeksiantajille voidaan määritellä joustavasti oikeuksia järjestelmän eri osiin. Valtuudet pitävät sisällään toimeksiantajaan liittyvät attribuutit, sekä lisäksi muutamia muita tärkeitä tietoja kuten toimeksiantajan valtuuksien voimassaoloajan. Valtuuksia käytetään käyttöoikeuksien varastoinnin lisäksi käyttäjän toimien seurantaan ja palvelupyyntöjen lähettäjän identiteetin varmennukseen. Toimeksiantajaan liittyvien ominaisuuksien lisäksi edellä mainittu spesifikaatio määrittelee turva-alueeksi kutsutun rakenteen. Turva-alueeseen on mahdollista liittää yksittäisiä järjestelmän osia niin, että kaikille näille osille pätevät tietoturvan osalta samat säännöt. Turva-alueiden ei tarvitse välttämättä jakaa järjestelmää osiin vaan, niitä voidaan määritellä tarvittaessa myös *päällekkäin (overlapping)* ja *sisäkkäin (nesting)*. Näiden alueiden tarkoitus on pienentää järjestelmän tietoturva-asetusten lukumäärää sekä vähentää oikeuksien uudelleenmäärittelyä järjestelmän muuttuessa.

CORBA:n tietoturvapalvelun keskeiset toiminnot

Käyttäjän tunnistus (user identification) sekä *käyttäjän identiteetin varmentaminen (user authentication)* ovat epäilemättä käytetyimpien tietoturvatoimintojen joukossa. Käyttäjän tunnistuksessa on kysymys käyttäjän identiteetin saattamisesta järjestelmän tietoon, esimerkiksi vaatimalla käyttäjää antamaan käyttäjätunnuksen. Käyttäjän identiteetin varmentamisessa on puolestaan kysymys identiteetin oikeellisuuden sekä käyttöoikeuden tarkistamisesta. Tämä voidaan tehdä esimerkiksi vertaamalla annettua käyttäjätunnusta järjestelmän käyttäjäluetteloon sekä pyytämällä käyttäjätunnusta

vastaava salasana, jota puolestaan verrataan käyttäjäluettelon kyseisen käyttäjän kohdalle tallennettuun salasanaan. Käyttäjän tunnistus ja identiteetin varmentaminen yhdessä pyrkivät varmistamaan, että käyttäjä todella on se joka väittää olevansa ja että kyseisellä käyttäjällä on lupa käyttää järjestelmää. Tietoturvapalvelu käyttää identiteetin varmennukseen *toimeksiantajaperustaista varmennusta (principal authentication),* jossa käyttäjä todistaa identiteettinsä tietoturvasta vastuussa oleville järjestelmän osille ja nämä osat myöntävät käyttäjälle identiteetille kuuluvat oikeudet. Käytännössä varmentaminen tapahtuu joko salasanan tai salausavaimen avulla sisäänkirjautumisen yhteydessä. Samalla sisäänkirjautuminen toteuttaa myös käyttäjän identifioinnin. Varmentaminen voidaan tarvittaessa suorittaa myös kaksisuuntaisesti, jolloin sekä järjestelmä voi varmistaa käyttäjän identiteetin oikeellisuuden, että käyttäjä voi varmistua käyttämänsä järjestelmän osan identiteetin oikeellisuudesta.

Käytön kontrolloinnilla (access control) rajoitetaan järjestelmän tai sen osien käyttöä niin, että järjestelmän käyttäjä voi suorittaa vain itselleen sallittuja toimenpiteitä. Tietoturvapalvelu mahdollistaa järjestelmään sijoitettujen objektien käytön kontrolloinnin jopa yksittäisten operaatioiden tarkkuudella. Kontrollointi on toteutettu käytännössä kohteisiin sidottavilla rajoittavilla attribuuteilla, kyvyillä sekä toimeksiantajiin sidottavilla ominaisuuksilla kuten valtuuksilla.

Käytön seurannan (security auditing) avulla ylläpito voi seurata järjestelmän käyttöä käyttäjäkohtaisesti. Seurannan tarkoitus on saattaa ylläpidon tietoon tilanteet, joissa (toisinaan näennäisesti) luvalliset käyttäjät suorittavat heiltä kiellettyjä toimenpideitä. Tällaisiin toimenpiteisiin lukeutuu esimerkiksi yleisesti tietossa olevien tietoturva-aukkojen systeemaattinen etsiminen ja hyödyntäminen. Tietoturvapalvelu toteuttaa käytön seurannan tallentamalla *tietoturvaan liittyvät tapahtumat (security related events).* Tallennuksen avulla on tarkoitus havaita sekä tapahtuneet että yrityksen asteelle jääneet tietoturvarikkeet. Vaikka tallennus toteutetaan monesti vain kirjaamalla tapahtuma lokitiedostoon, se voidaan toteuttaa tarvittaessa muullakin tavoin. Esimerkiksi vakavan rikkeen kohdalla tallennus voisi tarkoittaa tapahtuman kirjauksen lisäksi myös välittömän hälytyksen antamista järjestelmän ylläpitäjille. Palvelun keskeinen tarkoitus on kuitenkin mahdollistaa käyttäjien epäilyttävien toimien tallennus lokitiedostoihin, jotta järjestelmän ylläpito voi tarvittaessa tarkastella ja korjata käyttäjän tekemiä tuhoja jälkikäteen. Jotta lokitiedostot eivät

kuitenkaan täyttyisi turhasta seurantatiedosta, seurantaa on mahdollista rajoittaa edeltäkäsin määriteltävällä *seurantapolitiikallla (audit policy).* Sen avulla ylläpitäjät voivat määrittää mitä tapahtumia seurataan ja millaisissa tilanteissa seurantaa suoritetaan. Tapahtumia voidaan myös ryhmitellä *tapahtumaperheiksi (event families),* jolloin niitä voidaan käsitellä yksittäisiä tapahtumia suurempina kokonaisuuksina.

Kiistämättömyyden (non repudiation) avulla ylläpitäjät voivat kiistatta osoittaa toimeksiantajan suorittaneen järjestelmässä kiellettyjä toimenpiteitä. Näin ylläpito saa käyttäjät vastuuseen teoistaan tarpeen vaatiessa. Osoittaminen voidaan toteuttaa käytännössä esimerkiksi keräämällä todisteita suoritettuihin toimintoihin liittyvien palvelupyyntöjen lähettäjistä, sekä näihin palvelupyyntöihin liittyvien paluuviestien vastaanottajista. Näin voidaan todistaa teko, sen tekijä sekä lopputuloksen vastaanottaja. Toteutuksen on kuitenkin syytä varmistaa, ettei käyttäjälle jää mahdollisuutta poistaa tai muokata kerättyä todistusaineistoa. Kiistämättömyyden toteutus on määritelty tietoturvapalvelun kuvaavassa spesifikaatiossa harkinnanvaraisesti toteutettavaksi, eli sitä ei tarvitse välttämättä toteuttaa tietoturvapalveluun lainkaan.

Kommunikoinnin tietoturvalla (security of communication) voidaan luoda tietoturvan kannalta luotettavia kommunikaatiokanavia julkisten verkkojen yli. Luotettavat kommunikointikanavat ovat hajautetun järjestelmän tietoturvan kulmakivi, sillä järjestelmän sisäisen tiedonsiirron vuotaminen asiattomille tahoille pudottaa tietoturvan tason yleensä olemattomaksi. Esimerkiksi tiedonsiirtoa salakuuntelemalla selville saadut käyttäjätunnukset salasanoineen, avaavat yleensä tien järjestelmään ohi kaikkien muiden turvajärjestelyiden. Salakuuntelun lisäksi muita kommunikointiin kohdistuvia uhkia ovat muun muassa siirrettävän tiedon muuttaminen siirron aikana, kommunikaatiokanavien tukkiminen sekä valeidentiteetin omaksuminen lähettävässä tai vastaanottavassa päässä. Tietoturvapalvelussa vastapuolen identiteetin oikeellisuus varmistetaan avaamalla asiakkaan ja toteutusobjektin välille *turvayhteys (secury association).* Sen avaaminen muodostaa samalla *turvakontekstin (security context),* jonka sisällä tieto siirtyy tarvittaessa salakirjoitetussa muodossa. Turvakontekstin käyttö pienentää tietoturvan toteutuksen järjestelmälle aiheuttamia haittoja, sillä kontekstin muodostumisen jälkeen sen piirissä toimivat objektit voivat kommunikoida

aivan normaalisti. Näin vastapuolen identiteetin varmentaminen ja salakirjoitukseen liittyvä salausavainten vaihto tarvitsee suorittaa vain kerran avattua yhteyttä kohti. Ilman tällaista järjestelyä, varmentaminen ja salausavainten vaihto pitäisi suorittaa jokaisen palvelupyynnön yhteydessä erikseen. Turvayhteyden toteutukseen liittyvät yksityiskohdat määritellään *turvayhteyskäytännössä (Secure Interoperability Protocol)*, joka määrittää kommunikoinnissä käytettävät rajapinnat. Tällaisia yhteyskäytäntöjä on useita erilaisia ja niiden tukemat toiminnot vaihtelevat käytännöittäin. OMG ei suosittele mitään tiettyä yhteyskäytäntöä, vaan siirtää vastuun valinnasta järjestelmän suunnittelijoille. Yhteyskäytännöistä tosin *Secure Inter-ORB Protocol (SECIOP)* näyttää olevan tämän tutkielman kirjoitushetkellä erityisen kiinnostuksen kohteena. OMG asettaa kuitenkin kaikille tuetuille yhteyskäytännöille ja niissä käytettäville salausmenetelmille korkeat vaatimukset, joiden mukaan muun muassa turvayhteyskäytäntöjen tulee perustua julkisesti saatavilla oleviin spesifikaatioihin ja niissä tulee käyttää vain luotettavina pidettyjä salausmenetelmiä.

Tietoturvapalvelun ongelmia ja puutteellisuuksia

Mikään tietoturvan toteuttava järjestely ei ole koskaan täydellinen. Valitettavasti tämä koskee myös tietoturvapalvelua, jonka sisältöön on kohdistunut perusteltua arvostelua [19]. Ensimmäinen näistä kritiikin aiheista on muuttuvien sovellusten ja tarkasti rajaamattomien ympäristöjen heikko tuki. CORBA:n tietoturvapalvelu on arvostelijoiden mukaan alun perin suunniteltu muuttumattomia sovelluksia sekä tarkasti rajattuja ympäristöjä varten. Tämän vuoksi se soveltuu melko huonosti jatkuvasti muuttuviin ympäristöihin ja sovelluksiin. Esimerkiksi Internetin yli toimivien sovellusten tietoturvan toteuttaminen tietoturvapalvelun avulla on ongelmallista, sillä siirtotien varrelle sijoitetut palomuurit usein estävät ORB:ien välisen kommunikoinnin, eikä järjestelmän ylläpitäjillä ole monesti mahdollisuutta vaikuttaa kaikkien näiden palomuurien toimintaan.

Toinen kritiikin aihe on yksittäisten tietoturvatoimintojen sijoituksen vaikeus. Arvostelijoiden mukaan toisinaan on vaikeaa päättää, käytetäänkö ORB:in alaisuuteen sijoitettuja valmiita tietoturvapalvelun toimintoja vai rakennetaanko kyseiset toiminnot itse suoraan sovellukseen. Periaatteessa tietoturvapalvelun tarjoamaa valmista toiminnallisuutta tulisi käyttää aina kun mahdollista, sillä sen käyttö säästää

työtä ja sen sisältöä voidaan pitää luotettavana. Käytännössä palvelun tietoturvaominaisuudet ovat toisinaan kuitenkin liian raskaita yksinkertaisten tietoturvatoimintojen toteuttamiseen, jolloin tietoturvaominaisuudet on järkevintä rakentaa itse suoraan sovelluksiin. Tästä johtuen on usein epäselvää, miten yksittäinen tietoturvatoiminto tulisi toteuttaa ja minne se tulisi sijoittaa.

Kolmas kritiikin aihe on *sisäisten kommunikaatiokanavien (covert channels)* salaamattomuus, joka on tietoturvan kannalta erittäin paha puute. Näillä kanavilla ORB:it nimittäin siirtävät salamattomana järjestelmän sisäisestä toiminnasta kertovaa informaatiota, kuten palvelimien ja objektien nimiä. Sisäisten kanavien salakuuntelu voikin paljastaa väärinkäyttöä suunnitteleville tahoille luottamukselliseksi määriteltyä tietoa järjestelmän sisäisestä rakenteesta.

Neljäs kritiikin kohde on toimeksiantajaa ja hänen oikeuksiaan kuvaavien valmiiksi määriteltyjen attribuuttien niukka määrä. Nämä attribuutit eivät aina riitä määrittämään, mitä käyttäjä tarkalleen ottaen edustaa ja mitä hän saa järjestelmässä tehdä. Tämä aiheuttaa jo lyhyelläkin tähtäimellä ongelmia, sillä ilman riittävän ilmaisuvoimaisia attribuutteja toimeksiantajien valtuuksien määrittelyssä joudutaan tekemään kompromisseja. Näiden kompromissien seurauksena myönnetyt valtuudet voivat olla joko sallivampia tai rajoittavampia kuin olisi tarpeen. Pitkällä tähtäimellä attribuuttien niukka määrä on vieläkin vakavampi ongelma, sillä se pakottaa valmistajat lisäämään OMG:n tietoturva-attribuuttien rinnalle omia yhteensopivuutta heikentäviä standardoimattomia attribuuttejaan. Riittämättömät esimääritellyt attribuutit ovatkin tietoturvapalvelun eräs merkittävä heikkous.

Viides kritiikin aihe on käyttäjien toimien seurannan pohjana olevien määritysten puutteellisuus. Tietoturvapalvelun pohjana toimiva spesifikaatio jättää määrittelemättä seuranta-informaation salauksen sekä kerätyn tiedon analysointiin liittyvän toiminnallisuuden. Spesifikaatio jättää myös määrittelemättä kerättävän tiedon tallentamiseen käytettävän tallennusmuodon, joten epäyhteensopivuus eri valmistajien toteutusten kesken on enemmän kuin todennäköistä. Tämä estää muun muassa seurantatiedon keskitetyn käsittelyn, mikäli järjestelmässä käytetään useamman kuin yhden valmistajan toteutusta tietoturvapalvelusta.

Kuudes kritiikin kohde liittyy kiistämättömiin todisteisiin. Kiistämättömyyden käyttökelpoisuutta heikentää sen pohjana olevien määritelmien puutteellisuus ja sen käyttöön liittyvät ongelmat. Tarpeellisia ja täysin määrittelemättä jääneitä ominaisuuksia ovat esimerkiksi *jakelun päätäntävalta (delivery authority)*, *tuomion langettaja (adjucator)* sekä *turvatallennus (secure storage)*. Näiden ominaisuuksien puute heikentää kerättyjen todisteiden uskottavuutta. Esimerkiksi ilman jakelusta vastaavaa yksiselitteistä tahoa, todisteita voi joutua asiattomillekin henkilöille ja ilman turvatallennusta todisteiden muuttamattomuutta ei voida taata varmuudella. Kerätyt todisteet eivät ole myöskään itse teon todistuksen kannalta täysin kiistattomia, sillä niiden avulla voidaan osoittaa tarkasteltavaan operaatioon liittyvän palvelupyynnön lähettäminen ja vastaanottaminen, muttei itse operaation onnistunutta suoritusta. Näin todisteiden avulla voidaan osoittaa periaatteessa vain yritys, eikä itse tekoa. Suurimman ongelman muodostanee kuitenkin lainsäädäntö, joka rajoittaa käyttäjän toimia tarkasti kuvaavan seurantatiedon keräämistä useissa eri maissa, kuten esimerkiksi Suomessa. Kiistattomien todisteiden keräämisen toteuttava toiminnallisuus on kaiken lisäksi määritelty tietoturvapalveluun valinnaisesti mukaan otettavaksi ominaisuudeksi, joten sitä tuskin on toteutettu useankaan valmistajan tietoturvapalvelu komponenttiin. Kaikkien edellä mainittujen seikkojen perustella kiistämättömyys vaikuttaakin ominaisuudelta, johon on toistaiseksi syytä suhtautua varauksella.

Seitsemäs kritiikin aihe liittyy *toimintatapojen hallinnointiin (policy management).* Tietoturvapalvelu tarjoaa SecurityAdmin-rajapinnan, joka tarjoaa karkeasti määriteltyä toiminnallisuutta toimintatapojen hallinnointiin. Palvelusta kuitenkin puuttuu useita tärkeitä tietoturvaominaisuuksien hallinnointiin liittyviä ominaisuuksia, kuten toteutusalustan tarjoamien tietoturvapalveluiden ja sovellustason turvaominaisuuksien hallinnointi sekä ristiriitaisten toimintatapojen käsittelysäännöt ja toimintatapojen yhdistely.

Kahdeksas kritiikin aihe liittyy *luotettavuuteen (assurance).* Luotettavuus on laajalle alueelle levitetyissä hajautetuissa järjestelmissä usein vaikeasti saavutettavissa, sillä tällaiset järjestelmät sisältävät lukuisia rakenneosia, joiden hallinnointi voi vieläpä kuulua eri organisaatioille. Lisäksi tällaiset järjestelmät elävät jatkuvassa muutostilassa. Tietoturvapalvelun pohjana toimiva spesifikaatio tosin esittelee

hajautetun turvakeskuksen (distributed trusted computing base), jonka tarkoituksena on parantaa nimenomaan luotettavuutta. Tämän keskuksen tulisi sisältää kaikki tietoturvan kannalta kriittiset järjestelmän rakenneosat. Tällaisia osia ovat tietoturvasta vastuussa olevat sovellusobjektit, ORB ytimen koodi, objektisovittimet, salausalgoritmit sekä järjestelmän laitteisto. Keskuksen toteuttaminen käytännössä saattaa kuitenkin olla vaikeaa, sillä kaikki tietoturvan kannalta kriittiset rakenneosat eivät useinkaan ole järjestelmästä vastaavien tahojen hallinnassa. Esimerkiksi sovellusobjektit kuuluvat sovelluksiin, jotka puolestaan voivat kuulua järjestelmää käyttäville asiakkaille.

4.2 Samanaikaisuudenhallinta

Nykyaikaiset tietokoneet suorittavat lähes poikkeuksetta useampia operaatioita samanaikaisesti, oli sitten kyseessä yksittäinen tietokone tai moniosainen tietojärjestelmä. Selviytyminen tästä samanaikaisuudesta luotettavasti ja tehokkaasti, onkin tärkeä osa tietokoneen tai tietojärjestelmän toiminnallisuutta. Operaatioiden samanaikainen suoritus voidaan toteuttaa joko aidosti samanaikaisella rinnakkaissuorituksella tai vain näennäisesti samanaikaisella vuoroajolla. Käytettävästä toteutustavasta riippumatta, useiden operaatioiden suorittaminen samanaikaisesti aiheuttaa joukon suorituskykyyn ja luotettavuuteen liittyviä ongelmia. Nämä ongelmat on otettava huomioon järjestelmää rakennettaessa. Toimintaa jonka avulla tällaisilta ongelmilta pyritään välttymään, kutsutaan tässä tutkielmassa *samanaikaisuudenhallinnaksi (concurrency control)*. Tässä kohdassa tutustumme samanaikaisuuteen hajautettuissa järjestelmissä sekä CORBA-standardin samanaikaisuudenhallintaan.

Samanaikaisuudenhallinta hajautetuissa järjestelmissä

Yksittäisen tietokoneen samanaikaisuudenhallinta toteutetaan yleensä käyttöjärjestelmätasolla. Nykyiset käyttöjärjestelmät tarjoavat tähän tarkoitukseen vähintään moniajoa tukevan prosessien hallinnan, sekä muun siihen kiinteästi liittyvän toiminnallisuuden kuten muistinhallinnan. Tällainen prosessitason hallinnointi ei kuitenkaan riitä hajautetun järjestelmän samanaikaisuudenhallinnan toteuttamiseen, sillä se toteuttaa vain yksittäisen järjestelmän osan sisäisen

samanaikaisuudenhallinnan, sivuuttaen järjestelmän käyttäytymisen kokonaisuutena. Siksi toteutusvälineen tulisikin tarjota yleiskäyttöistä toiminnallisuutta, jonka avulla samanaikaisuudenhallinta voidaan toteuttaa koko järjestelmään sen yksittäisten osien toimiessa rinnakkaisesti. Vaikka edellä puhutaan rinnakkaisuudesta, samanaikaista tietojenkäsittelyä ei tule kuitenkaan sekoittaa *tietojen rinnakkaiskäsittelyyn (paraller computing)*, jossa pyritään mahdollisimman suureen tehokkuuteen yleensä järjestelmän yleiskäyttöisyyden kustannuksella. Hallitussa samanaikaisuudessa nimittäin pyritään ensisijaisesti vain selviytymään rinnakkain suoritettavien operaatioiden toisilleen aiheuttamista ristiriitatilanteista minimaallisin vahingoin. Tämä ei kuitenkaan tarkoita tehokkuusnäkökohtien täydellista unohtamista, vaan yksinkertaisesti luotettavuuden priorisoimista tehokkuusnäkökohtien edelle.

Lukot samanaikaisuudenhallinnan työkaluna

CORBA-standardi määrittelee samanaikaisuudenhallinnan toteutettavaksi *samanaikaisuudenhallintapalvelussaan (Concurrency Control Service)*. Tämän palvelun kuvaavassa spesifikaatiossa standardi määrittelee muutamia keskeisiä käsitteitä, jotka on hyvä tuntea. Näistä käsitteistä *asiakas (client)* on taho joka käyttää järjestelmän resursseja ja jota edustaa tyypillisesti objekti. *Resurssi (resource)* on järjestelmän mielivaltainen osa, joka voi tarjota käyttäjälleen sisältämäänsä tietoa tai toiminnallisuutta ja joka tyypillisesti käyttää samanaikaisuudenhallintapalvelua hallitakseen useita samanaikaisesti itseensä kohdistuvia palvelupyyntöjä. *Lukko (lock)* on puolestaan resurssiin kiinnitettävä elementti, jonka avulla resurssiin kohdistuvia vaateita voidaan kontrolloida hallitusti. Resurssin kontrollointi lukon avulla toteutetaan asettamalla lukon kiinnittänyt asiakas etulyöntiasemaan usean asiakkaan käyttäessä resurssia. Kontrollointiin sisältyy monesti myös useiden samanaikaisten lukkojen kiinnittämisen estäminen, mikäli lukkoja kiinnittävien asiakkaiden toimet aiheuttavat jaettavalle resurssille ristiriitatilanteen. Tällainen ristiriitatilanne voisi edellyttää esimerkiksi resurssina toimivan objektin tilan olevan samanaikaisesti erilainen eri asiakkaille, mikä ei (vielä toistaiseksi) ole objektiparadigman perusteella mahdollista. Objektin tilan tulee nimittäin olla aina yksiselitteinen. Lukot ovat erittäin keskeisessä asemassa CORBA:n samanaikaisuudenhallinnassa, sillä ne ovat käytännöllisesti katsoen ainoa CORBA:n ohjelmoijalle tarjoama työkalu järjestelmätason samanaikaisuudenhallintaan.

Lukkojen toimintaperiaate on hyvin yksinkertainen, mutta ne voidaan toteuttaa monin eri tavoin. Naiivi lukko edustaa näistä toteutustavoista yksinkertaisinta. Se varaa resurssin täysin yhdelle asiakkaalle, estäen muita asiakkaita pääsemään käsiksi resurssiin ennen lukon poistamista. Naiivin lukon toteuttaminen ja käyttäminen on varsin suoraviivaista ja se onkin eräs yksinkertaisimmista työkaluista resurssien hallitun jakaminen toteuttamiseen. Näin toteutetut lukot eivät kuitenkaan ole täysin ongelmattomia. Naiivien lukkojen käyttö nimittäin vähentää järjestelmän tehokkutta. Yhden asiakkaan varatessa resurssin täysin omaan käyttöönsä, muut sitä tarvitsevat asiakkaat voivat vain odotella sen vapautumista tekemättä mitään. Tämä on merkittävä ongelma, sillä naiivien lukkojen käytöstä aiheutuva tehokkuuden menetys voi olla järjestelmässä erittäin suurta.

CORBA:n samanaikaisuudenhallinta palvelu toteuttaa lukot naiveja lukkoja edistyneemmällä tavalla. Samanaikaisuudenhallinta palvelu antaa järjestelmän rakentajille mahdollisuuden vähentää lukkojen käytöstä aiheutuvaa *joutoaikaa (idle time)* määrittelemällä lukoille viisi erilaista tilaa. Näiden tilojen avulla resursseja ei tarvitse varata yhden ainoan asiakkaan käyttöön, vaan ne voidaan jakaa eri asiakkaiden kesken asetetuista lukoista huolimatta. Jaettavaan resurssiin kohdistuvat vaateet eivät kuitenkaan saa aiheuttaa ristiriitatilanteita resurssin käyttäytymiselle. Tämän vuoksi samanaikaisuudenhallinta palvelussa määritelläänkin lukitustilojen *yhteensopivuuskartoitus (lock compability mapping)*, jonka perusteella lukkojen toteutuksesta vastaava taho voi havaita mahdolliset ristiriitatilanteet etukäteen. Lukitustilat eivät kuitenkaan ratkaise resurssien lukituksesta aiheutunutta tehohävikkiä automaattisesti, sillä asiakkaan on valittava käytettävä tila lukkoa asetettaessa itse. Näin tilat antavat ohjelmoijille mahdollisuuden resurssien käytön yksityiskohtaisempaan hallintaan ja tätä kautta tehohävikin pienentämiseen. Tämä tarkoittaa valitettavasti myös sitä, että lukitustiloista saatava hyöty riippuu täysin ohjelmoijien kyvystä ja viitseliäisyydestä hienosäätää resurssien hallintaa erilaisilla lukitustiloilla. Mikäli ohjelmoijat käyttävät jatkuvasti vain vahvinta mahdollista lukkoa, samanaikaisuudenhallinta palvelun lukot käyttäytyvät aivan kuten naiivit lukot. Taulukko 1 esittelee samanaikaisuudenhallinta palvelun lukitustilojen yhteensopivuuskartoituksen.

Myönnetty tila (pystytasossa) / **Anottu tila** (vaakatasossa)	Aikomus lukea	Luku	Päivitys	Aikomus kirjoittaa	Kirjoitus
Aikomus lukea	*OK*	*OK*	*OK*	*OK*	*RISTIRIITA*
Luku	*OK*	*OK*	*OK*	*RISTIRIITA*	*RISTIRIITA*
Päivitys	*OK*	*OK*	*RISTIRIITA*	*RISTIRIITA*	*RISTIRIITA*
Aikomus kirjoittaa	*OK*	*RISTIRIITA*	*RISTIRIITA*	*OK*	*RISTIRIITA*
Kirjoitus	*RISTIRIITA*	*RISTIRIITA*	*RISTIRIITA*	*RISTIRIITA*	*RISTIRIITA*

Taulukko 1 : Erilaiset lukitustilat sekä niiden väliset yhteensopivuudet.

Tehottomuus ei kuitenkaan ole ainoa lukkojen käyttöön liittyvä ongelma. Lukkoja käytettäessä resurssien välille syntyy usein riippuvuussuhteita, joista voi muodostua ratkeamattomia pattitilanteita eli *lukkiumia (deadlocks)*. Lukkiumassa operaatiot odottavat resurssien vapautumista periaatteessa ikuisesti, pysäyttäen pahimmillaan koko järjestelmän toiminnan. Tällainen tilanne voi syntyä esimerkiksi kahden toisistaan riippuvaisen resurssin odottaessa toistensa vapautumista. Lukkiumat ovatkin lukkojen käyttöön liittyvä vakava ongelma ja mielestäni hajautetun järjestelmän toteutusympäristön tulisi tarjota toiminnallisuutta, jonka avulla lukkiumista voidaan selviytyä hallitusti. Lukkiumia voidaan nimittäin hallita estämällä niiden syntyminen tai avaamalla ne niiden syntymisen jälkeen. Kummassakin tapauksessa lukkiumat tulee kuitenkin pystyä havaitsemaan. Menetelmät joiden avulla lukkiumat voidaan havaita jo ennen niiden syntymistä perustuvat yleensä algoritmeihin, jotka osoittavat mahdolliset lukkiumat asiakkaan asettaessa lukkoa resurssiin. Tällainen algoritmi voi perustua esimerkiksi syklien etsimiseen riippuvuussuhteita kuvaavasta verkosta. Jälkikäteen lukkiuman havaitsevat menetelmät perustuvat usein *määräaikoihin (timeout)*, jotka antavat lukon olemassaololle vain rajallisen eliniän. Määräajan umpeuduttua lukon pitävyyttä löysennetään tai se poistetaan kokonaan. Tällöin on olemassa mahdollisuus että lukkiuma häviää. Ongelmaksi määräaikoja käytettäessä muodostuu kuitenkin menetelmän epävarmuus, sillä lukkiumien syntymistä valvova taho ei tiedä johtuuko määräajan ylitys lukkiumasta vai jostain muusta syystä. Lukkoa vain löysennetään määräajan umpeudettua, olipa syy sitten mikä tahansa. Seurauksien huomioimisesta puhumattakaan.

CORBA-standardi ei suoranaisesti ota kantaa lukkiumien hallintaan. Periaatteessa koko standardista puuttuu lukkiumien hallinnan toteuttava toiminnallisuus, sillä CORBA ei tarjoa minkäänlaista algoritmistä lähestymistapaa lukkiumien havainnointiin, eikä se myöskään tue lukoille asetettavia määräaikoja. Lukkiumien hallinnan voi kuitenkin toteuttaa CORBA:ssa kohtuullisen pienellä vaivalla, esimerkiksi asettamalla palvelupyyntöjen suoritukselle määräajat pyynnöt suorittavassa ohjelmakoodissa. Tällöin ohjelmoijat joutuvat kuitenkin toteuttamaan itse sen, mitä toteutusvälineen tulisi tarjota valmiina. Lukkiumien hallinnan puuttuminen onkin omasta mielestäni vakava puute CORBA:n samanaikaisuudenhallinnassa.

4.3 Yhteensopivuus

Yhteensopivuuteen liittyvät ongelmat muodostavat erityisen keskeisen hajautettujen järjestelmien ongelma-alueen. Nämä ongelmat tulevat monesti esiin jo järjestelmän elinkaaren alkuvaiheessa, jolloin järjestelmää ollaan vasta rakentamassa. Jo tällöin joudutaan usein käyttämään rakenneosia, jotka eivät ole suunniteltu toimimaan toistensa kanssa. Järjestelmiä ei nimittäin yleensä rakenneta täysin puhtaalta pöydältä, vaan uuteen järjestelmään joudutaan monesti liittämään vanhempia olemassa olevia tietojärjestelmiä tai –varastoja, jotka eivät ole yhteensopivia uusien rakenneosien kanssa. Näin muodostuvat osien väliset liitokset ovat harvoin ongelmattomia. Ongelmaa pahentaa usein se ettei tietojärjestelmien yhdistely rajoitu pelkästään järjestelmän rakentamisen alkuvaiheisiin, vaan yhdistely voi jatkua vaikka koko tietojärjestelmän elinkaaren ajan, jolloin ongelmat kasaantuvat ajan myötä yhä suuremmiksi. Pelkän yhteensopivuuden sijasta asiaa voisikin käsitellä myös muutoksen kestävyytenä, mutta koska termi yhteensopivuus pitää sisällään sekä välittömästi että tulevaisuudessa ilmenevät yhteensopivuuteen liittyvät ongelmat, asiaa käsitellään tässä tutkielmassa yksinkertaisesti yhteensopivuutena. Tässä kohdassa käymme läpi yhteensopivuusongelmien keskeisiä osa-alueita ja tutustumme CORBA:n yhteensopivuusongelmiin tarjoamiin ratkaisuihin.

Ohjelmointikielten yhteensopivuus

Ohjelmointikielet ovat yhteensopivia pääsääntöisesti vain itsensä kanssa. Tämä aiheuttaa ongelmia, sillä hajautetuissa järjestelmissä toiminnallisuutta joudutaan monesti kirjoittamaan useammalla kuin yhdellä kielellä. CORBA toteuttaa eri ohjelmointikielillä kirjoitettujen osien yhteensovittamisen IDL-kieleen perustuvan järjestelyn avulla. Tämä järjestely mahdollistaa eri ohjelmointikielillä kirjoitettujen järjestelmän osien välisen kommunikoinnin, sillä järjestelyssä osat eivät kommunikoi suoraan toistensa kanssa, vaan IDL-kielellä kuvattujen ohjelmointikieliriippumattomien rajapintojen kautta. IDL-kieleen perustuvassa järjestelyssä käytettävien ohjelmointikielten väliset eroavaisuudet peitetään tulkkaamalla tietoa lähettävän ohjelman lähdeohjelmointikielen piirteet ensin IDL-kielen yleispäteviksi piirteiksi, jotka tämän jälkeen tulkataan takaisin tietoa vastaanottavan ohjelman käyttämälle ohjelmointikielelle. Koska tulkinta voidaan suorittaa kaikkien IDL-kartoitusten kautta tuettujen kielten välillä, voidaan kaikkia näitä kieliä käyttää järjestelmässä samanaikaisesti. Samalla ratkaisu mahdollistaa yhteensopivuuden rakantamisen olemassa olevien ja tulevaisuudessa kehitettävien kielten välille. IDL-kieli toimiikin hajautetuissa järjestelmissä universaalina ohjelmistorajapintanotaationa [20]. IDL-kieltä ja kartoituksia on käsitelty luvussa 3.

Tiedon erilaiset esitystavat

Tietojärjestelmissä käytetään vain harvoin yhtä ainoaa esitystapaa tiedon käsittelyssä ja sen tallennuksessa. Tämän vuoksi järjestelmiin onkin monesti toteutettava toiminnallisuutta, joka muuntaa tarvittaessa tietoa esitystavasta toiseen. IDL-kieleen perustuva järjestely toteuttaa CORBA järjestelmissä muunnoksen automaattisesti eri ohjelmointikielien tietotyyppien välillä, mutta se ei ratkaise kaikkia tiedon esitystapoihin liittyviä ongelmia. Järjestelmissä on nimittäin usein tietovarastoja kuten tiedostoja ja tietokantoja, joiden sisältöä on tarpeen muuntaa esitystavasta toiseen. CORBA ei tarjoa tähän muuntamiseen valmiita työkaluja, mutta IDL-kieltä tukevien ohjelmointikielten avulla järjestelmän ylläpitäjät voivat tehdä itse muunnokset toteuttavat konversio-ohjelmat.

Erilaiset tiedonsiirtoprotokollat

Erilaisten tiedonsiirtoprotokollien keskinäinen yhteensopimattomuus on hajautetuissa järjestelmissä ongelma, sillä toimivan järjestelmän rakenneosien tulee voida kommunikoida keskenään. CORBA vähentää tällaisia ongelmia käyttämällä ORB:ien välisessä kommunikaatiossa omaa GIOP yhteyskäytäntöään. GIOP vähentää yhteensopivuusongelmia, sillä se tarjoaa avoimen ja koko järjestelmälle yhteisen yhteyskäytännön, joka ei ole sidottu minkään yksittäisen valmistajan patenteilla suojattuun standardiin. ORB:ien välistä tiedonsiirtoa ja GIOP-yhteyskäytäntöä on käsitelty luvussa 3.

Käyttöjärjestelmien yhteensopivuus

Käyttöjärjestelmä muodostaa perustan, jonka päälle sovellukset sekä järjestelmän sisäiset palvelut rakennetaan. Se tarjoaa sovelluksille peruspalvelut kuten tiedostojärjestelmän sekä muistin- ja prosessien hallinnan. Tällaisia palveluita käyttämällä ohjelmoijat välttyvät kirjoittamasta perustavanlaatuista toiminnallisuutta yhä uudelleen ja voivat keskittyä järjestelmän varsinaisen toiminnallisuuden toteuttamiseen. Käyttöjärjestelmän palveluissa on kuitenkin myös varjopuolia. Näitä palveluita käyttäessään ohjelmoijat sitovat ohjelmat tiukasti käyttöjärjestelmien omiin ohjelmointirajapintoihin, jotka ovat lähes poikkeuksetta epäyhteensopivia muiden käyttöjärjestelmien vastaavien ohjelmointirajapintojen kanssa. Näin ohjelmien siirrettävyys eri käyttöjärjestelmillä varustettujen tietokoneiden välillä heikkenee merkittävästi. Tämä on ongelmallista hajautetuissa järjestelmissä, joissa käytetään monesti samanaikaisesti useaa eri käyttöjärjestelmää. CORBA:n väliohjelmisto-tyyppinen arkkitehtuurimalli vähentää usean käyttöjärjestelmän samanaikaisesta käytöstä seuraavia yhteensopivuusongelmia korvaamalla käyttöjärjestelmäkohtaisia ohjelmointirajapintoja omilla rajapinnoillaan. Näin sovelluksia ei tarvitse sitoa suoraan käyttöjärjestelmien rajapintoihin, vaan ne voidaan sitoa CORBA:n väliohjelmistorajapintoihin, jotka puolestaan sidotaan käyttöjärjestelmien rajapintoihin ORB:eissa. Tämä mahdollistaa sovelluksen siirtämisen käyttöjärjestelmästä toiseen, sillä CORBA:n väliohjelmistorajapinnat ovat aina samanlaisia, oli käytettävä käyttöjärjestelmä sitten mikä tahansa.

4.4 Luotettavuus

Luotettavuudella tarkoitetaan tässä tutkielmassa järjestelmien tai niiden osien kykyä suoriutua niille annetuista tehtävistä jollain ennaltakäsin tunnetulla onnistumisen todennäköisyydellä. Luotettavuuden merkitys tietojärjestelmille on suuri, sillä tietojärjestelmät ovat tulleet yhä tärkeämmäksi tekijäksi sekä yksittäisten ihmisten että erilaisten ogranisaatioiden toiminnassa. Rahoitusyhtiöt, media, tukku- ja jälleenmyynti, kuljetuspalvelut sekä muut ihmisten jokapäiväiseen elämään vaikuttavat tahot ovat yhä suuremmissa määrin riippuvaisia tietojärjestelmiensä toiminnasta. Mikäli järjestelmät eivät toimi odotetusti, organisaatioille sekä niiden asiakkaille aiheutuu vahinkoa, jonka vakavuus voi vaihdella pienimuotoisesta mielipahasta aina ihmishenkien menetyksiin asti. Vahinkojen laajasta vakavuusasteikosta huolimatta sellaisten organisaatioiden lukumäärä, joille tietojärjestelmien toimintahäiriöt aiheuttavat merkittäviä kustannuksia on jatkuvasti kasvussa [21]. Tämän vuoksi tietojärjestelmien luotettavuuteen tulee kiinnittää erityistä huomiota niin hajautetuissa kuin keskitetyissäkin järjestelmissä. Tässä kohdassa tarkastelemme poikkeuksia, tapahtumanhallintaa sekä toisinnusta järjestelmän luotettavuutta parantavina työkaluina. Tutustumme myös CORBA:n tapaan toteuttaa kyseiset ominaisuudet.

Poikkeukset ja luotettavuus

Poikkeukset (exceptions) ovat objektiperustaisessa ohjelmoinnissa keskeinen väline virhetilanteiden havainnointiin ja käsittelyyn. Niitä voidaan pitää eräänlaisina ilmaisimina, jotka osoittavat virhetilanteet ohjaamalla ohjelman suorituksen erillisiin poikkeuksenkäsittelijöihin. Poikkeusten käytön eräs merkittävä etu on mahdollisuus virheiden käsittelyyn varsinaisen toiminnallisuuden toteuttavan ohjelmakoodin ulkopuolella. Ohjelmakoodi selkeytyy huomattavasti, kun toiminnallisuus ja virheiden käsittely eroitetaan toisistaan. Lisäksi poikkeusten mukana on mahdollista kuljettaa tarkkaa tietoa virheiden syistä ja esiintymispaikoista. Jos poikkeukset ovat käyttökelpoisia rakennettaessa keskitettyjä järjestelmiä, niin ne ovat erittäin hyödyllisiä rakennettaessa hajautettuja järjestelmiä. Toiminnallisuuden hajauttaminen useaan eri tietokoneeseen kasvattaa nimittäin potentiaalisten virheiden määrää, sillä hajautus itsessään aiheuttaa paljon sellaisia virhetilanteita, joihin keskitetyissä

järjestelmissä ei tarvitse varautua lainkaan. Tällaisia virhetilanteita ovat esimerkiksi kommunikaatioverkon häiriöistä johtuvat virheet. Huomioitavien virhetilanteiden määrän ollessa suuri, toiminnallisuuden toteuttavan koodin ja virheiden käsittelyn eroittaminen toisistaan nousee oleelliseksi tekijäksi rakennettaessa luotettavasti toimivaa järjestelmää. Myös virheen syystä ja esiintymispaikasta kertova tieto on erittäin käyttökelpoista korjattaessa esimerkiksi toistuvia virheitä monimutkaisessa ympäristössä. Poikkeukset ovatkin toimiva keino selkeyttää ohjelmakoodin virheenkäsittelyä ja näin ne myös tukevat luotettavasti toimivien hajautettujen järjestelmien rakentamista ja ylläpitoa.

CORBA tukee poikkeuksia ja poikkeusten määrittely toteutetaan standardissa IDL-kielen kautta. Ohjelmoija voi määritellä itsenäisiä poikkeuksen käsittelijöitä, jotka voidaan sitoa operaatioihin IDL-kuvauksissa. Tällaisten *käyttäjän määrittelemien poikkeusten (user defined exceptions)* lisäksi CORBA tarjoaa joukon *järjestelmäpoikkeuksia (standard exceptions*), jotka kattavat yleisimmät toteutusalustassa ilmenevät virhetilanteet. Tällaisia virhetilanteita ovat etupäässä muistinvarauksessa, kommunikaatiossa sekä ORB:in rakenneosissa esiintyvät häiriöt. CORBA ei kuitenkaan kartoita kaikkia mahdollisia virhetilanteita omiksi poikkeuksikseen, vaan yksittäinen järjestelmäpoikkeus kattaa useita erilaisia samaan kontekstiin liittyviä virhetilanteita. Esimerkiksi dynaamisen muistinvarauksen virheestä ilmoittava järjestelmäpoikkeus kattaa sekä muistinvaraus- että vapausyritykseen liittyvät virheet. Järjestelmäpoikkeukset parantavatkin CORBA:an perustuvan hajautetun järjestelmän toimintavarmuutta, sillä ilman niitä järjestelmätason virheiden havaiseminen jäisi täysin ohjelmoijien vastuulle.

Tapahtumanhallinta ja luotettavuus

Tapahtumanhallinta (transaction handling) tunnetaan parhaiten tietokantojen luotettavuuden parantamismenetelmänä. Nykyisin on kuitenkin yleisesti myönnetty, että se on avain myös luotettavasti toimivien hajautettujen sovellusten rakentamiseen [22]. Tapahtumanhallinta ei kuitenkaan menetelmänä ole *hopealuoti (the silver bullet),* jolla sovellusten tai järjestelmien epäluotettavuus poistetaan päiväjärjestyksestä. Itse asiassa se ei ole yksinään edes riittävä menetelmä järjestelmän luotettavuuden toteuttamiseen. Se tarjoaa kuitenkin varsin yksinkertaisen

tavan tehdä sovelluksista entistä luotettavampia. Objektiperustaisissa sovelluksissa käytettävä tapahtumanhallinta toimii ohjelman kannalta hyvin samankaltaisesti kuin relaatiotietokantojen tapahtumanhallinta. Aluksi ohjelma aloittaa uuden tapahtuman kertomalla siitä tapahtumanhallinnalle (vrt. SQL-kielen begin käsky). Tämän jälkeen ohjelma luo, käyttää, muokkaa tai poistaa sovelluksen objekteja (vrt. tietokannan rivejä), jotka ovat jonkin luokan (vrt. tietokantataulun) instansseja. Lopuksi ohjelma kertoo tapahtumanhallinnalle hyväksyvänsä muutokset (vrt. SQL-kielen commit käsky). Vasta tässä vaiheessa ohjelman objekteihin kohdistamat toiminpiteet muuttuvat pysyviksi. Virheen sattuessa ohjelman suoritus ajautuu poikkeuksenkäsittelijään, jonne sijoitettu tapahtumanhallinnalle annettava muutosten perumiskäsky (vtr. SQL-kielen rollback käsky) kumoaa kaikki tapahtuman sisällä tehdyt toimenpiteet. Näin tapahtumanhallinta varmistaa objektien tilojen eheyden ja tätä kautta nopeuttaa toipumista erilaisista virhetilanteista.

Tapahtumanhallinta on tärkeä osa luotettavuuden parantamiseen tähtäävää toiminnallisuutta CORBA-standardissa. CORBA toteuttaa tapahtumanhallinnan omalla tapahtumanhallinta palvelullaan, jonka keskeisiä toimintaperiaatteita ovat *ACID-ominaisuudet*, *kaksivaiheinen hyväksymisprosessi (two-phase commit)* sekä *sisäkkäiset tapahtumat (nested transactions).* ACID-ominaisuuksiksi kutsutaan neljästä toimintaperiaatteesta koostuvaa joukkoa, jonka tarkoitus on tehdä tapahtumanhallinnasta hyödyllinen ja luotettava. Joukon muodostavat toimintaperiaatteet ovat *atomisuus (atomicy)*, *yhtenäisyys (consistency)*, *eristäminen (isolation)* sekä *kestävyys (durability).* Atomisuus on periaate, joka takaa että kaikki tapahtuman sisällä tehdyt muutokset joko saatetaan voimaan tai perutaan. Periaate takaa ettei voimaan saateta koskaan vain osaa tehdyistä muutoksista. Yhtenäisyys puolestaan tarkoittaa periaatetta, jonka mukaan tapahtumanhallinnan on saatettava objektit yhdestä yhtenäisestä tilasta toiseen yhtenäiseen tilaan. Käsiteltävän objektijoukon eheys ei siis saa millonkaan hajota tapahtumanhallinnan toiminnan seuraksena. Eristäminen takaa sen etteivät tapahtuman sisällä tehdyt muutokset näy ulkopuolelle ennen niiden voimaansaattamista. Tämä periaate mahdollistaa muutosten perumisen ilman järjestelmälle aiheutuvia vahingollisia sivuvaikutuksia. Kestävyys puolestaan takaa että voimaan saatetut muutokset jäävät voimaan pysyvästi, lukuun ottamatta ehkä täysin kestämättömiä virhetilanteita kuten laitteiston toiminnan pysäyttäviä sähkökatkoksia.

Kaksivaiheinen voimaansaattaminen on kaupallisissa tietokantaohjelmistoissa vakiintunut menetelmä, joka nimensä mukaisesti suorittaa voimaansaattamis prosessin kahdessa erillisessä vaiheessa. Voimaansaattamisen ensimmäisen vaiheen aikana tapahtumanhallinta voimaansaattaa alustavasti kaikki tapahtumaan sisältyvät muutokset, jonka jälkeen se pyytää kaikilta muutoksista vastaavilta tahoilta kuittauksen voimaansaattamisen onnistumisesta. Vasta varmistuttuaan kaikkien voimaasaattamisten onnistumisesta, tapahtumanhallinta muuttaa voimaansaattamisen pysyväksi. Tällainen käytäntö sietää vikatilanteita hyvin ja se soveltuukin vikasietoisena menetelmänä käytettäväksi myös hajautetussa tietojenkäsittelyssä [3]. Sisäkkäiset tapahtumat ovat tapahtumanhallinnan ominaisuus, joka sallii usean sisäkkäin sijoitetun hyväksymättömän tapahtuman samanaikaisen olemassaolon. Siinä toistensa alle sisentyvät tapahtumat muodostavat puurakenteen muodon omaavan hierarkian. Sisäkkäiset tapahtumat mahdollistavat kookkaan tapahtuman jakamisen useisiin pieniin alitapahtumiin. Tämä lisää vikasietoisuutta, sillä yksittäisen alitapahtuman epäonnistuminen ei välttämättä johda koko tapahtuman perumiseen. Alitapahtumia voidaan nimittäin yrittää saattaa voimaan tarpeen vaatiessa useamminkin kuin kerran, esimerkiksi yrittämällä suorittaa epäonnistuneen tapahtuman sisältämä toiminto uudestaan. Näin voidaan ehkäistä esimerkiksi satunnaisten tiedonsiirtovirheiden haitallista vaikutusta järjestelmän toimintaan.

Toisinnus ja luotettavuus

Toisinnus (replication) on menetelmä, jossa järjestelmän resursseista luodaan *kopioita (copy,replica)* käytettäväksi alkuperäisen resurssin sijasta. Se on eräs keskeisessä asemassa oleva menetelmä vikasietoisten hajautettujen järjestelmien rakentamisessa [3]. Kopioiden käytön voima luotettavuuden parantamisessa perustuu varsin suoraviivaisesti vian esiintymistodennäköisyyden pienenemiseen kopioiden lukumäärän kasvaessa. Usean tarvittaessa toisensa korvaavan rakenneosan samanaikaisen hajoamisen todennäköisyys on nimittäin periaatteessa käänteisesti verrannollinen kopioiden lukumäärän neliöön, mikäli rakenneosien hajoamistodennäköisyys on kaikille kopioille yhtä suuri. Toisinnus voidaan toteuttaa joko aktiivisena tai passiivisena. *Aktiivisessa toisinnuksessa (active replication)* asiakas kommunikoi kaikkien kopioiden kanssa täysin tietoisina kopioiden

olemassaolosta. *Passiivisessa toisinnuksessa (passive replication)* asiakas luulee kommunikoivansa vain yhden objektin kanssa, eikä näin ollen ole tietoinen kopioista. Tässä tutkielmassa käsiteltävä CORBA-standardin versio ei suoranaisesti tue toisinnusta, mutta se sisältää yleiskäyttöistä toiminnallisuutta, joka avustaa ohjelmoijia toisinnuksen toteuttamisessa. Esimerkiksi tapahtumapohjainen viestintäpalvelu on käyttökelpoinen apuväline tähän tarkoitukseen, sillä se mahdollistaa yhdeltä monelle tyyppisen viestinnän. Sen avulla viestit parametreineen saadaan välitettyä kohteina olevien resurssin lisäksi samanaikaisesti myös niiden kopioille.

4.5 Tehokkuus

Tehokkuus on luotettavuuden ohella eräs tärkeimmistä tietojärjestelmän ominaisuuksista. Tämän vuoksi se on myös erityisen huomionarvoinen hajautun järjestelmän suunnittelukriteeri. Hajautettujen järjestelmien tehokkuudelle asetetut vaatimukset eivät yleensä eroa merkittävästi keskitettyjen järjestelmien vastaavista vaatimuksista. Tehokkuuden saavuttamiseksi tarvittavat keinot ovat kuitenkin eri tyyppisiä, sillä hajautettujen ja keskitettyjen järjestelmien rakenteet eroavat toisistaan merkittävästi. Tässä kohdassa tutustumme hajautetun järjestelmän pullonkauloihin sekä tehokkuusnäkökohtien huomioimiseen CORBA-standardissa.

Hajautetun järjestelmän pullonkaulat

Verrattaessa hajautetun ja keskitetyn järjestelmän rakenteita toisiinsa tehokkuuden näkökulmasta, merkittävimmät eroavaisuudet löytyvät järjestelmien kommunikaation toteuttavista osista: Siinä missä keskitetyn järjestelmän kommunikaatioväylä on suorituskykyinen, löyhästi kytketyn hajautetun järjestelmän kommunikaatioväylän suorituskyky on yleensä heikohko. Hajautetun järjestelmän kommunikointiverkossa esiintyy kahta perustavanlaatuista ongelmaa, joiden vuoksi verkosta muodostuu valitettavan usein järjestelmän pullonkaula. Nämä ongelmat ovat verkon riittämätön *tiedonsiirtokapasiteetti (transfer capasity)* sekä tietonsiirrossa esiintyvä merkittävä *verkkoviive (network latency)*. Riittämätön tiedonsiirtokapasiteetti tarkoittaa verkon kyvyttömyyttä siirtää tarvittavaa määrää tietoa käytettävissä olevassa ajassa. Tämä ongelma paljastuu yleensä siirrettäessä suuria määriä tietoa järjestelman eri osien

välillä, jolloin riittämätön tiedonsiirtokapasiteetti kasvattaa siirtoon kuluvan ajan kohtuuttoman pitkäksi. Verkkoviive puolestaan viittaa siihen ajanjaksoon, joka kuluu odotettaessa tiedonsiirron alkamista. Sitä ilmenee aina siirrettäessä tietoa verkon kautta, eikä se ole ongelma pienessä mittakaavassa. Verkkoviiveen operaatioiden kutsuja lievästi hidastavalla vaikutuksella on kuitenkin ikävä ominaisuus kasautua merkittäväksi tehottomuutta aiheuttavaksi tekijäksi, tiedonsiirtokertojen kasvaessa suuriksi. Jos esimerkiksi jokaista kuvitteellisen ohjelman tuhatta verkon yli tehtävää aliohjelmakutsua hidastaa kymmenen millisekunnin verkkoviive kutsua kohden, koko ohjelman suoritus sisältää jo lähes kaksi minuuttia odotteluun käytettyä odotusaikaa, vaikka siirrettävän tiedon määrä olisi vain yksi bitti aliohjelmakutsua kohden. Verkon aiheuttamaa järjestelmän tehokkuutta alentavaa vaikutusta voidaan kuitenkin vähentää suunnittelemalla järjestelmä oikein. Tämä voidaan tehdä esimerkiksi sijoittamalla järjestelmän ruuhkaisimpiin osiin riittävän tehokas verkko ja pienentämällä verkkoliikenteen määrää sijoittamalla järjestelmän osat järkevästi toisiinsa nähden [23].

Tehokkuusnäkökohdat CORBA-standardissa

CORBA-standardi määrittää sisältämänsä toiminnallisuuden etupäässä liitännöistä muodostuvien rajapintakuvausten avulla, eikä se ota suoranaisesti kantaa näiden rajapintojen toteutuksesta vastaavien ORB-ohjelmistojen tehokkuuteen. OMG toki luettelee standardiin liittyvissä spesifikaatioissaan koko joukon kunnianhimoisia tavoitteita, joiden joukkoon kuuluu myös tavoite tehokkuudesta. Nämä tavoitteet ovat kuitenkin melko ylimalkaisia eikä spesifikaatio esitä kovinkaan konkreettisia ratkaisuja lueteltujen tavoitteiden saavuttamiseksi. Vastuu CORBA-standardiin perustuvan alustan tehokkuudesta jää näin ORB ohjelmistojen valmistajille. Standardi tarjoaa kuitenkin ominaisuuksia, joiden avulla järjestelmän suunnittelijat voivat vaikuttaa tehokkuuteen. Tällaisia ovat muun muassa operaatioiden ei-pysäyttävät (asynkroniset) kutsut. Näiden ominaisuuksien avulla suunnittelijat voivat sekä kompensoida verkkoviivettä että pienentää verkkoliikenteen määrää. Ei-pysäyttäviä kutsuja käytettäessä ohjelman suorituksen ei nimittäin tarvitse pysähtyä odottamaan vastausta verkon kautta lähetettyyn viivettä sisältävään palvelupyyntöön, vaan ohjelman suoritus voi edetä rinnakkain verkossa siirtyvän palvelupyynnön kanssa [30]. Paikkatuntumattomuuden avulla suunnittelijat voivat puolestaan siirtää helposti

palveluita toteuttavia objekteja järjestelmän vähemmän ruuhkaisiin osiin ja näin tasoittaa verkon tiedonsiirtokapasiteettia.

4.6 Monimutkaisuus

Monimutkaisuus on merkittävä hajautettujen järjestelmien rakentamiseen ja ylläpitoon liittyvä ongelma. Tähän on syynä hajautettujen järjestelmien rakenneosien potentiaalisten kombinaatioiden suuri määrä sekä näiden rakenneosien keskinäinen heterogeenisyys. Näistä kahdesta erityisesti heterogeenisyys johtaa monimutkaisuuden lisääntymiseen jo järjestelmän suunnitteluvaiheessa, sillä järjestelmän erilaiset rakenneosat on usein sovitettava toiminaan keskenään. Hajautetun järjestelmän monimutkaisuus voi riistäytyä nopeasti käsistä, ellei monimutkaisuuden hallintaa pidetä tärkeänä tavoitteena jo järjestelmän suunnittelu- tai peräti määrittelyvaiheessa. Monimutkaisuudella on nimittäin ikävä tapa kantautua kumulatiivisesti järjestelmän kehitysvaiheesta toiseen. Käytännössä tämä tarkoittaa sitä että jos monimutkaisuus on ongelma jo järjestelmän määrittelyvaiheessa, se on varmasti vielä suurempi ongelma suunnitteluvaiheessa ja vieläkin suurempi ongelma toteutusvaiheessa. Tällöin ylläpitovaiheen hallinta on monimutkaisuuden kannalta jo aivan omaa luokkaansa. Monimutkaisuuden hallinnan tulee pitää sisällään joukko toteutettavissa olevia toimenpiteitä, joiden avulla järjestelmä pidetään hallitussa tilassa koko sen elinkaaren ajan. Mikäli tässä ei onnistuta, tuloksena on hallitsematon kaaos. Monimutkaisuuden hallinnassa onnistuminen ei kuitenkaan ole aina helppoa, sillä jo hajautettujen järjestelmien suunnittelu on vaikeampaa kuin vastaavien keskitettyjen järjestelmien suunnittelu [10]. Tässä kohdassa tutustumme läpikuultavuuden, väliohjelmiston sekä objektiperustaisuuden käyttöön hajautetun järjestelmän monimutkaisuuden hallintamenetelminä sekä tarkastelemme CORBA-standardin suhdetta näihin menetelmiin.

Läpikuultavuus monimutkaisuuden hallinnassa

CORBA tukee monimutkaisuuden hallintaa määrittelemällä osan toiminnallisuudestaan toteutettavaksi läpikuultavasti. *Läpikuultavuus*[1] *(transparency)* on malli, jossa toimintojen toteutukset piilotetaan niiden käyttäjiltä niin, ettei käyttäjien tarvitse tietää miten käytettävät toiminnot on toteutettu. Hajautetun järjestelmän toteutusmallina läpikuultavuus on erittäin käyttökelpoinen, sillä se yksinkertaistaa järjestelmän rakennetta ja tekee toimintojen käyttämisestä helpompaa. Se onkin suositeltava keino vähentää hajautettujen järjestelmien monimutkaisuutta [2]. Läpikuultavuus toimii myös keskeisenä elementtinä avoimien hajautettujen järjestelmien suunnitteluperiaatteita kuvaavassa ISO standardoidussa RM-ODP (Reference Model for Open Distributed Processing) mallissa [24], joka jakaa läpikuultavuuden kahdeksaan eri ilmenemismuotoon. Nämä ovat *saantituntumattomuus (access transparency), paikkatuntumattomuus (location transparency), toisinnustuntumattomuus (replication transparency), vikaantumistuntumattomuus (failure transparency), keskeytystuntumattomuus (persistence transparency), siirtymistuntumattomuus (migration transparency), uudelleensijoitustuntumattomuus (relocation transparency) sekä* tapahtumanhallintatuntumattomuus *(transaction transparency).* CORBA tukee näistä vain saanti- ja paikkatuntumattomuutta [25]. Tässä kohdassa tarkastelemme näitä läpikuultavuuden ilmenemismuotoja objektiperustaisen hajautetun järjestelmän näkökulmasta.

Saantituntumattomuutta tukevassa hajautetussa järjestelmässä objekteja käyttävä asiakas voi käyttää paikallisia ja etäobjekteja identtisesti. Tällöin asiakkaan ei tarvitse välittää käyttämiensä objektien tai niiden ympäristöjen toteutukseen liittyvistä yksityiskohdista lainkaan. Asiakkaan ei myöskään tarvitse välittää kommunikointiin liittyvistä yksityiskohdista, vaan asiakas voi luoda, poistaa ja käyttää objekteja aina samalla tavalla riippumatta objektin tyypistä, sen ympäristöstä tai sijainnista järjestelmässä. CORBA toteuttaa paikallisen ja etäobjektin käytön

[1] Puhuttaessa läpikuultavuuden ilmenemismuodoista suomen kielellä, käytetään usein liitteitä -tuntumattomuus tai -riippumattomuus (esimerkiksi vikaantumistuntumattomuus tai paikkariippumattomuus). Tässä tutkielmassa kahdesta edellisestä on valittu liite -tuntumattomuus. Itse läpikuultavuudesta käsitteenä sen sijaan puhutaan sen omalla nimellä.

yhdenmukaistamisen asiakasohjelmiin käännettävien asiakaskantojen avulla, sillä asiakaskantojen kautta ohjelmat voivat käyttää etäobjekteja aivan kuten paikallisia objekteja [17]. CORBA peittää myös kommunikointiin liittyvät yksityiskohdat toteuttamalla asiakaskannan ja palvelinrungon välisen tiedonsiirron asiakkalle näkymättömästi, sekä muuntamalla siirrettävän tiedon lähettäjän käyttämästä esitysmuodosta vastaanottajan käyttämään esitysmuotoon IDL-kuvausten avulla.

Paikkatuntumattomuutta tukevassa hajautetussa järjestelmässä objekteja käyttävän asiakkaan ei tarvitse tuntea objektien tarkkaa sijaintia, voidakseen käyttää niiden palveluita. Asiakas vain yksilöi haluamansa objektit jollain sovitulla tavalla, jättäen objektien paikannuksen järjestelmän suoritettavaksi. Haluamansa objektit asiakas voi yksilöidä esimerkiksi ilmoittamalla kyseisten objektien luokkien nimet. CORBA tarjoaa useita erilaisia tapoja sitoa objektit niitä käyttäviin asiakkaisiin. Ensimmäinen näistä tavoista on objektien sitominen asiakkaisiin absoluuttisten osoitetietojen perusteella. Asiakas voi halutessaan päästä käsiksi tarvitsemiinsa objekteihin ilmoittamalla niiden absoluuttiset osoitteet, jolloin objekteja vastaavien luokkien nimet sekä niiden sijainnit määrittelevät verkko-osoitteet (eli objektit omistavien ORB:in osoitteet) yleensä ohjelmoidaan suoraan asiakasohjelmiston lähdekoodiin tai paikallisiin asetuksiin. Tällöin asiakas pääsee objektiin käsiksi erittäin nopeasti, mutta objekteja ei sidota asiakkaaseen paikkatuntumattomasti. Toinen tapa on sitoa objektit asiakkaisiin nimipalvelun kautta. Tällöin objektit sidotaan nimipalveluun, josta asiakas voi tiedustella objektien sijaintia niiden luokkien nimien perusteella. Näin objektien paikkatietoja ei tarvitse sitoa kiinteästi asiakkaaseen. Nimipalvelun avulla objekteja voidaankin käsitellä paikkatuntumattomasti [26]. Kolmas tapa on käyttää kaupankäyntipalvelua, jota voidaan käyttää paikkatuntumattomuuden toteuttamiseen samalla tavalla kuin nimipalvelua. Kaupankäynti- ja nimipalvelu eroavat toisistaan lähinnä vain objektien luokittelu- ja hakuperusteiden osalta, eikä näillä eroavaisuuksilla ole paikkatuntumattomuuden kannalta merkitystä.

Toisinnustuntumattomuutta tukevassa hajautetussa järjestelmässä joukko toisinnettuja järjestelmän resursseja näkyy asiakkaalle kopioiden sijasta yhtenä resurssina. Näin asiakas voi käyttää useita samanaikaisesti toimivia kopioita, aivan kuten yksittäistä resurssia. Asiakkaan ei näin tarvitse olla edes tietoinen useiden kopioiden

olemassaolosta. CORBA ei tue toisinnustuntumattomuutta, eikä itse asiassa edes toisinnusta.

Vikaantumistuntumattomuutta tukevassa hajautetussa järjestelmässä toteutusalusta voi piilottaa järjestelmässä esiintyneitä virheitä siinä suoritettavilta ohjelmilta, mikäli nämä virheet ovat korjattavissa. Toteutusalusta voi yrittää korjata virheitä esimerkiksi suorittamalla virheitä aiheuttaneita operaatioita uudelleen useita kertoja, kunnes se lopulta saa kaikkien operaatioiden suoritukset onnistumaan. CORBA ei suoranaisesti tue vikaantumistuntumattomuutta. Vikojen havaitseminen ja niistä toipuminen voidaan kuitenkin toteuttaa esimerkiksi poikkeuksien ja tapahtumanhallinnan avulla, mutta tällöin toteuttaminen jää täysin järjestelmän rakentajien vastuulle.

Keskeytystuntumattomuutta tukevassa hajautetussa järjestelmässä hajautetut objektit voidaan aktivoida tai deakvoida palvelimella niitä käyttävän asiakkaan tästä häiriintymättä. Objektien deaktivoinnilla tarkoitetaan tässä objektien siirtämistä palvelimen keskusmuistista pysyväismuistiin (esimerkiksi kiintolevylle) ja aktivoinnilla niiden siirtämistä pysyväismuistista takaisin keskusmuistiin ja tätä kautta takaisin sovellusten käyttöön. Syy deaktivointiin voi olla esimerkiksi pitkään käyttämättä olleiden objektien sitomien resurssien vapauttaminen hyödyllisempään käyttöön. CORBA ei tue keskeytystuntumattomuutta.

Siirtymistuntumattomuutta tukevassa hajautetussa järjestelmässä hajautettujen objektien siirtäminen paikasta toiseen kesken objekteissa tapahtuvan operaatioiden suorituksen, ei häiritse näiden objektien eikä niitä käyttävien asiakkaiden toimintaa. Uudelleensijoitustuntumattomuutta tukevassa hajautetussa järjestelmässä hajautettuihin objekteihin sidottujen sovellusten siirtäminen paikasta toiseen kesken sovelluksen suorituksen, ei vastaavasti myöskään häiritse asiakkaiden eikä hajautettujen objektien toimintaa. CORBA ei tue siirtymistuntumattomuutta eikä uudelleensijoitustuntumattomuutta.

Tapahtumanhallintatuntumattomuutta tukevassa objektiperustaisessa hajautetussa järjestelmässä tapahtumanhallinta on toteutettu niin, ettei sen piiriin liitettäviin hajautettuihin objekteihin tarvitse rakentaa erikseen tukea tapahtumanhallinnalle. Tällainen ohjelmoijalle täysin automaattisesti toimiva objektien tapahtumanhallinta on

kuitenkin lähes mahdotonta toteuttaa [24]. Esimerkiksi yksittäiseen objektiin kohdistuva muutosten perumiskomento ei automaattisesti mitätöi sen ympäristöön tekemiä muutoksia. Objektien tuleekin toteuttaa itse jokin muutokset aidosti peruva operaatio, jota tapahtumanhallinta voi perumiskomennon yhteydessä kutsua. Näin tapahtumanhallinta on toteutettu myös CORBA:n tapahtumanhallintapalvelussa, jonka alaisuuteen liitettävien objektien on periydyttävä tapahtumanhallintapalvelun määrittelemistä abstrakteista luokista ja näin toteutettava tapahtumien kontrollointiin tarvittavat operaatiot. Näin ollen CORBA ei tue tapahtumanhallintatuntumattomuutta.

Väliohjelmisto ja objektit monimutkaisuuden hallinnassa

CORBA-standardiin perustuva toteutusalusta tukee monimutkaisuuden hallintaa jo pelkän rakenteensa avulla. Sen väliohjelmistotyyppisen rakenteen eräs tehtävä on nimittäin auttaa pitämään hajautetun järjestelmän monimutkaisuus kurissa [27]. Väliohjelmisto vähentää monimutkaisuutta tarjoamalla järjestelmässä suoritettaville ohjelmille juuri niiden tarvitsemia korkean abstraktiotason palveluita. Näin kyseisiä palveluita ei tarvitse toteuttaa järjestelmän eri osissa paikallisesti, mahdollisesti toisistaan poikkeavilla tavoilla. CORBA:n objektiperustainen väliohjelmisto tarjoaa koko järjestelmälle esimerkiksi yhteisen kommunikaatiokanavan. Tämän kommunikaatiokanavan kautta ohjelmat voivat siirtää kokonaisia objekteja paikasta toiseen, sen sijaan että objektien sisältämät attribuutit pitäisi siirtää yksi kerrallaan primitiivisinä tietotyyppeinä paikallisten tiedonsiirtomenetelmien avulla. Väliohjelmistotyyppisen rakenteensa sekä läpikuultavan toimintojen toteutusmallinsa lisäksi CORBA tukee monimutkaisuuden hallintaa myös objektiperustaisuudellaan. Objektit auttavat nimittäin hallitsemaan monimutkaisuutta rakennettaessa suuria järjestelmiä [28]. Esimerkiksi objektiperustaiset ohjelmistonkehitystekniikat ovat tärkeitä hajautettujen järjestelmien monimutkaisuuden hallinnassa [29].

5 Yhteenveto

Hajautetut järjestelmät ovat vaihtoehto keskitetyille järjestelmille. Hajauttaminen on mielekästä, mikäli tiedon tai toiminnallisuuden keskittäminen yhteen tietokoneeseen ei tarjoa esimerkiksi riittävästi suorituskykyä, luotettavuutta tai alueellista kattavuutta. Hajautuksen avulla saavutetut edut eivät kuitenkaan ole ilmaisia, sillä hajautettujen järjestelmien rakentaminen on pääsääntöisesti kalliimpaa ja monimutkaisempaa kuin keskitettyjen järjestelmien. Merkittävänä syynä tähän ovat hajautetuille järjestelmille ominaiset ja niissä usein toistuvat ongelmat. CORBA-standardi on suosittu hajautettujen järjestelmien toteutusympäristöjä kuvaava standardi. Se tarjoaa yhtenevän toteutuspohjan useita erilaisia käyttöjärjestelmiä sekä ohjelmointikieliä sisältäviin ympäristöihin. CORBA toimii tällaisessä heterogeenisessä hajautetussa järjestelmässä kommunikaatio- ja tulkkauskanavana, joka mahdollistaa toistensa kanssa yhteensopimattomien osien yhteistoiminnan.

Tämän tutkielman tarkoituksena on ollut hajautettuihin järjestelmiin ja CORBA-standardiin tutustumisen lisäksi tutkia CORBA:n käyttökelpoisuutta hajautettujen järjestelmien toteutusalustana. Se on tehty jakamalla hajautetuille järjestelmille ominaiset ongelmat kuuteen eri ongelma-alueeseen ja tarkastelemalla CORBA:n näihin ongelma-alueisiin tarjoamia teknisiä ratkaisuja. Ongelma-alueet ovat tietoturva, samanaikaisuudenhallinta, yhteensopivuus, luotettavuus, tehokkuus sekä monimutkaisuuden hallinta. *Tietoturvan toteuttamiseen* CORBA tarjoaa oman tietoturvapalvelunsa, joka sisältää käyttäjän tunnistuksen ja identiteetin varmentamisen, käytön kontrolloinnin, käytön seurannan, kiistämättömät todisteet sekä tiedonsiirron salaamisen. Tietoturvapalveluun on kuitenkin kohdistunut perusteltua kritiikkiä, sen sisältämien ongelmien ja puutteellisuuksien suuren määrän vuoksi. *Samanaikaisuudenhallintaan* se tarjoaa samanaikaisuudenhallintapalvelunsa, joka tarjoaa resurssien kontrollointiin edistyneitä lukkoja. *Yhteensopivuusongelmien ehkäisemiseen* se tarjoaa universaalina ohjelmistorajapintanotaationa toimivan IDL-kielen, koko järjestelmälle yhteisen GIOP-protokollan sekä muun muassa sovellusten siirrettävyyttä parantavan väliohjelmistoarkkitehtuurin. *Luotettavuuden parantamiseen* se tarjoaa virheiden havainnointia ja käsittelyä helpottavat IDL-kielen poikkeukset sekä relaatiotietokantojen tapahtumanhallintaa vastaavan

objektiperustaisen tapahtumanhallinnan. *Tehokkuuden parantamiseen* se tarjoaa rinnakkaisuuden mahdollistavan operaatioiden ei-pysäyttävän kutsutavan sekä toteutusobjektien siirtämistä helpottavan paikkatuntumattomuuden. *Monimutkaisuuden hallintaan* se tarjoaa läpikuultavuuden, joka yksinkertaistaa järjestelmän rakennetta ja tekee toimintojen käyttämisestä helpompaa. Lisäksi CORBA:n väliohjelmisto tyyppinen arkkitehtuuri sekä objektiperustaisuus auttavat pitämään monimutkaisuuden kurissa.

Tutkimus osoitti CORBA:n olevan erittäin käyttökelpoinen hajautettujen järjestelmien toteutusalustana, sillä se tarjoaa paljon valmiita ratkaisuja kaikkiin hajautettujen järjestelmien keskeisiin ongelma-alueisiin. Erityisen kiittävästi standardissa on ratkaistu yhteensopivuusongelmat. Tietoturvaan liittyvissä ratkaisuissa sen sijaan on puutteita. Tehtyä tutkimusta voisi jatkaa esimerkiksi tarkastelemalla CORBA:n tehokkuutta tarkemmin. Tämän voisi tehdä esimerkiksi vertailemalla ORB-ohjelmistojen mitttua suorituskykyä muiden hajautettujen järjestelmien toteutusvälineiden suorituskykyyn. Vertailukohteita voisivat olla esimerkiksi Sun Microsystems:in *Java RMI* ja Microsoft:in *DCOM*.

Viitteet

[1] CORNAFION group (1985) *Distributed computing systems*. Elsevier Science Publishers, North Holland.

[2] Coulouris G., Dollimore J. (1998) *Distributed systems: Concepts and design* (1st.edition). Addison wesley, New York.

[3] Coulouris G. , Dollimore J., Kindberg T. (2000) *Distributed systems: Concepts and design* (3rd.edition). Addison-Wesley, US.

[4] Emmerich W. (2000) *Engineering distributed objects,* Wiley, UK.

[5] Charles J. (1999) *Middleware moves to the forefront*. Computer 1999(5),17-19.

[6] Emmerich W. (2000) *Software engineering and middleware : A Roadmap*. International Conference on Software Engineering, ACM Press, New York, 117-129.

[7] Berstein P. A (1996) *Middleware : A model for distributed system services*. Communications of the ACM 39(2),86-98.

[8] Koskimies K. (2000) *Oliokirja*. Satku, Helsinki.

[9] Liskov B. (1988) *Distributed programming in Argus*. Communications of the ACM 31(3), 300-312.

[10] Champine G. A., Coop R. D., Heinselman R. C (1980) *Distributed computer systems: Impact on management, desing and analysis.* North-holland publishing company, New York.

[11] Tripathi A. R., Noonan T., (1998) *Desing of a Remote Procedure Call System of Object-Oriented Distributed Programming*. Software-practice and experience 28(1), 24-47.

[12] OMG (1995) *The common object request broker: Architecture and specification* (version 2.0) . www.omg.org

[13] OMG (1997) *A discussion of the object management architecture*. http://www.omg.org

[14] Schmidt D. C. (1998) *Developing distributed object computing applications with CORBA (CORBA tutorial)*. http://www.cs.wustl.edu/~schmidt

[15] Gamma E., Helm R., Johnson R., Vlissides J. (1995) *Desing patterns: Elements of reusable object-oriented software*. Addison-wesley. US.

[16] Schmidt D. C., Vinoski S. (1997) *Object adapters: Concepts and terminology*. C++ Report 9(11).

[17] Baker S. (1997) *Corba distributed objects: Using Orbix*. Addison wesley, Essex.

[18] OMG (1998) *Security Service specification* (version 1.2). http://www.omg.org

[19] Alireza A., Lang U., Padelis M., Schreiner R., Schumacher M. (2000) *The challenges of CORBA security*. Springer-Verlag, Informatik aktuell series.

[20] Mowbray T. J. (1997) *CORBA desing patterns*. Wiley, UK.

[21] Cristian F. (1991) *Understanding Fault-Tolerant Distributed Systems*. Communications of the ACM 34(2),57-78.

[22] OMG (2000) *Transaction Service Specification* (version 1.1, New edition - May). http://www.omg.org

[23] Smith C. U., Williams L. G. (1998) *Performance Engineering Models of CORBA-based Distributed-Object Systems*. 24th International Computer Measurement Group Conference.

[24] *Open Distributed Processing - Reference model (RM-ODP)*. ISO/IEC standard 10746-1. www.iso.org

[25] Mayers C. (1996) *ANSAwise - Engineering Distributed Systems*. http://www.ansa.co.uk

[26] OMG (2000) *CORBA service: Naming Service Specification* (version 1.0, New edition - April).

[27] Campbell A. T., Coulson G., Kounavis M. E. (1999) *Managing complexity: Middleware explained*. IEEE IT Pro 1999(9-10).

[28] Tiwary A., Raj R. K., Lea D. S., Bosch C. J. (1995) *Building large distributed software systems using objects*. OOPSLA 95.

[29] Vinoski S. (1996) *CORBA: Integrating diverse applications within disributed heterogeneous environments*. IEEE Communications 35(2).

[30] Brunsch D., O'Ryan C., Schmidt D. C. (2000) *Designing an Efficient and Scalable Server-side Asyncrony Model for CORBA*. Proceedings of the ACM SIGPLAN workshop on Languages, compilers and tools for embedded systems. ACM Press, New York.

Liite 1: CORBA-palvelut

Tässä liitteessä kuvataan lyhyesti CORBA-palvelut (*CORBA services*).

Tapahtumapohjainen viestintä (Events)

Palvelu toteuttaa tapahtumapohjaisen kommunikointimekanismin. Siinä viestin lähettäjää ja vastaanottajaa ei sidota toisiinsa kiinteästi, vaan mekanismi perustuu kommunikaatiokanaviin joihin lähettäjät voivat viestinsä. Nämä viestit välittyvät kaikille vastaanottajille, jotka ovat rekisteröityneet kuuntelemaan lähettäjän käyttämää kanavaa. Nämä kanavat nähdään yleensä viestin lähettäjiin kohdistuvina tapahtumina, joihin tapahtumista kiinnostuneet viestien vastaanottajat reagoivat.

Tietoturva (Security)

Sisältää tietoturvan toteuttamiseen käytettävää toiminnallisuutta. Tähän toiminnallisuuteen kuuluu käyttäjien identiteetin varmentaminen, objektien käytön rajoittamnen käyttöoikeuksien perusteella, käyttöoikeuksien delegointiin käyttäjältä niitä tarvitseville objekteille sekä käyttäjien toimien seuranta.

Tapahtumanhallinta (Transactions)

Palvelu toteuttaa objektiperustaisen tapahtumanhallinnan, joka vastaa tietokannoissa käytettävää tapahtumanhallintaa, mutta toimii objekteilla tietokantataulujen rivien sijaan.

Kaupankäynti (Trading)

Tämä palvelu on vaihtoehto nimipalvelulle. Siinä missä nimipalvelu välittää objekteja niihin sidottujen nimien perusteella, kaupankäyntipalvelu välittää objekteja niihin sidottujen ominaisuuksien perusteella.

Objektien elinkaari (Life cycle)

Palvelu sisältää objektien ajonaikaiseen hallintaan tarkoitettua toiminnallisuutta. Palvelun pääpaino on *tehdas (factory)* [15] mallia noudattavan objektityypin käytöllä.

Objektien tallentaminen (Externalization)

Palvelun avulla objekteja voidaan siirtää ajonaikaisesti järjestelmän keskusmuistista massamuistiin, josta ne voidaan tarvittaessa jälleen elvyttää toimiviksi objekteiksi.

Lisenssointi (Licencing)

Palvelu sisältää lisenssien valvontaan liittyvää toiminnallisuutta. Lisenssien käyttöä voidaan valvoa käyttäjätunnusten, käyttäjien lukumäärän, käyttökertojen, ajanjakson ja käyttöpaikkojen perusteella. Seurantasäännöt ovat varsin joustavasti määriteltävissä palvelun käyttäjän tarpeiden mukaisesti.

Aika (Time)

Palvelu sisältää aikaan, ajanhallintaan sekä ajastukseen liittyviä yleiskäyttöisiä palveluita.

Objektien ominaisuudet (Property)

Palvelun avulla voidaan lisätä luokkiin ominaisuuksia. Nämä ominaisuudet asettuvat CORBA:n tyyppijärjestelmän päälle, eivätkä muuta IDL-kuvauksen tai DII-rajapinnan kautta objektille määriteltyjä ominaisuuksia.

Luokkien suhteet (Relationships)

Palvelun avulla voidaan luoda luokkien välisiä suhteita. Nämä suhteet eivät vaikuvat luokkien IDL-kuvauksiin tai DII-rajapinnan kautta tehdyihin luokkamäärittelyihin, vaan ne asettuvat olemassa olevien suhteiden päälle.

Samanaikaisuudenhallinta (Concurrency control)

Palvelu sisältää samanaikaisuudenhallinnassa käytettävää toiminnallisuutta. Sen sisältö rajoittuu lähinnä objekteihin asetettaviin lukkoihin, joiden avulla voidaan ehkäistä resurssien samanaikaisesta käytöstä aiheutuvia ristiriitatilanteita.

Objektien säilöntä (Persistent objects)

Palvelu määrittelee *sovelluskehyksen (framework),* jolla käytössä olevat objektit voidaan tallentaa esimerkiksi tietokantaan tai tiedostoihin myöhempää käyttöä varten. Ennen käyttöä palvelun sovelluskehys tulee täydentää toiminnallisuudella, joka toteuttaa varsinaisen tallennusvaiheen käytettävään massamuistiin.

Kyselyt (Query)

Palvelu määrittelee yksinkertaisen sovelluskehyksen ja kokoelmatyypin. Määriteltyä sovelluskehystä täydentämällä palvelussa määriteltyyn kokoelmaan voidaan kohdistaa hakuja. Palvelu ei määrittele uutta hakukieltä, vaan rohkaisee olemassa olevien kyselykielien kuten SQL:n soveltamista haun toteuttamiseen.

Nimipalvelu (Naming)

Palvelun avulla hajautettuihin objekteihin voidaan sitoa nimiä, joiden perusteella niiden paikkaa voidaan tiedustella nimipalvelulta. Nimipalvelu on siis eräänlainen keskitetty objektien paikkatiedon varasto. Sen avulla objektien paikkatietoa ei tarvitse sitoa kiinteästi niitä käyttäviin asiakkaisiin. Tämä mahdollistaa objektien siirtämisen paikasta toiseen, ilman tarvetta tehdä muutoksia objekteja käyttäviin asiakkaisiin. Palvelu tarjoaa myös *nimeämisyhteydet (naming context),* joiden avulla nimiä voidaan ryhmitellä loogisiksi kokonaisuuksiksi. Nimipalvelun rooli on CORBA järjestelmässä erittäin oleellinen ja se onkin eräs yksikertaisimmista ja käyttökelpoisimmista CORBA palveluista [17].