VTT TIEDOTTEITA – MEDDELANDEN – RESEARCH NOTES 1911

CORBAn soveltaminen joustavan valmistusjärjestelmän perusohjelmistoon

Mikko S. Holappa

VTT Elektroniikka

VALTION TEKNILLINEN TUTKIMUSKESKUS
ESPOO 1998

ISBN 951–38–5310–1 (nid.)
ISSN 1235–0605 (nid.)

ISBN 951–38–5311–X (URL: http://www.inf.vtt.fi/pdf)
ISSN 1455–0865 (URL: http://www.inf.vtt.fi/pdf)

Copyright © Valtion teknillinen tutkimuskeskus (VTT) 1998

JULKAISIJA – UTGIVARE – PUBLISHER

Valtion teknillinen tutkimuskeskus (VTT), Vuorimiehentie 5, PL 2000, 02044 VTT
puh. vaihde (09) 4561, faksi (09) 456 4374

Statens tekniska forskningscentral (VTT), Bergsmansvägen 5, PB 2000, 02044 VTT
tel. växel (09) 4561, fax (09) 456 4374

Technical Research Centre of Finland (VTT), Vuorimiehentie 5, P.O.Box 2000,
FIN–02044 VTT, Finland
phone internat. + 358 9 4561, fax + 358 9 456 4374

VTT Elektroniikka, Sulautetut ohjelmistot, Kaitoväylä 1, PL 1100, 90571 OULU
puh. vaihde (08) 551 2111, faksi (08) 551 2320

VTT Elektronik, Inbyggd programvara, Kaitoväylä 1, PB 1100, 90571 ULEÅBORG
tel. växel (08) 551 2111, fax (08) 551 2320

VTT Electronics, Embedded Software, Kaitoväylä 1, P.O.Box 1100, FIN–90571 OULU, Finland
phone internat. + 358 8 551 2111, fax + 358 8 551 2320

Toimitus Leena Ukskoski

Libella Painopalvelu Oy, Espoo 1998

Holappa, Mikko S. CORBAn soveltaminen joustavan valmistusjärjestelmän perusohjelmistoon [Applying CORBA in the basic software for a flexible manufacturing system]. Espoo 1998, Valtion teknillinen tutkimuskeskus, VTT Tiedotteita – Meddelanden – Research Notes 1911. 95 s.

Avainsanat manufacturing systems, FMS, software design, software components, CORBA

Tiivistelmä

Työssä tutkittiin CORBA-spesifikaation määrittelemää ohjelmistoarkkitehtuuria, sen toteutuksia sekä soveltuvuutta pehmeiden reaaliaikajärjestelmien ohjaukseen. Tarkastelun alla oli erityisesti joustavan valmistusjärjestelmän perusohjelmiston toteuttaminen CORBA-yhteensopivalla ohjelmistolla.

Ohjelmistojen koon ja niiden välisten integrointien lukumäärän kasvaminen hankalasti hallittaviksi on aiheuttanut tarvetta kehittää yhtenäisiä rajapintoja käyttäviä ohjelmia sekä laajennettavia ja uudelleenkäytettäviä ohjelmistoarkkitehtuureja ja menettelytapoja. Ongelmien välttämiseksi ohjelmiston eri osat tulisi saada mahdollisimman riippumattomiksi toisistaan ja riippuvuussuhteet tulisi kuvata standardilla tavalla, jotta muutokset yhteen osaan ohjelmistoa vaikuttaisivat mahdollisimman vähän ohjelmiston muihin osiin.

CORBA tarjoaa selkeän komponenttiperustaisen arkkitehtuurin ja piilottaa verkkotason ohjelmoinnin. Lisäksi se mahdollistaa ohjelmiston rajapintojen kuvaamisen yhtenäisellä tavalla, mikä vähentää ohjelmistojen integrointityötä. CORBAn käyttöönotto vaatii kuitenkin panostusta koulutukseen, eikä sen kaupallisten toteutusten suorituskyky vielä täytä kovien reaaliaikajärjestelmien asettamia vaatimuksia.

Tämän työn yhteydessä tehty joustavan valmistusjärjestelmän perusohjelmisto toteutettiin kaupallisella ORB-toteutuksella (Orbix) ja oliotietokannalla (Objectivity). Sitä kokeiltiin todellisessa toimintaympäristössä, missä käyttöjärjestelmänä oli Windows NT 4.0 ja siirtomediana Ethernet-lähiverkko.

Holappa, Mikko S. CORBAn soveltaminen joustavan valmistusjärjestelmän perusohjelmistoon [Applying CORBA in the basic software for a flexible manufacturing system]. Espoo 1998, Valtion teknillinen tutkimuskeskus, VTT Tiedotteita – Meddelanden – Research Notes 1911. 95 p.

Keywords manufacturing systems, FMS, software design, software components, CORBA

Abstract

The purpose of this work is to examine CORBA, some of its implementations and its suitability for controlling soft real time applications. Special attention is paid to its application in implementing the basic software components for a flexible manufacturing system.

The increasing size of software systems and the number of different integrations between them has made them hard to maintain. This has created a demand for developing scalable, reusable software architectures and development methods that use common, standardized interfaces between software components. In order to avoid integration and complexity problems, the components of a software system should be as independent as possible. If the dependencies between software components are described in a standard way, changes made to one component will have less effect on others.

CORBA introduces a clear, component-based architecture and hides network-level programming. Its interface definition language alleviates software integration. However, moving to CORBA requires investments in education, and the performance of current commercial implementations is not adequate for hard real-time systems.

This diploma thesis also describes the basic software for a flexible manufacturing system. This was implemented with a commercial CORBA-compliant product (Orbix) and an object database management system (Objectivity) running on a Windows NT 4.0 operating system in an Ethernet local area network. The software has been tested in a real environment.

Alkulause

Tämä työ tehtiin VTT Elektroniikan Teknologian kehittämiskeskus (TEKES) -rahoitteisessa ARTTU-tavoitetutkimusprojektissa. Projektisssa mukana olleita yritysosapuolia olivat ABB Transmit OY ja OY Mercantile AB Fastems, joista jälkimmäinen tarjosi projektiin konkreettisen sovellusympäristön.

Valvojana tässä työssä on toiminut apulaisprofessori Juha Röning Oulun yliopiston tietokonetekniikan laboratoriosta, jolle tässä yhteydessä haluan esittää kiitokseni joustavasta yhteistyöstä ja asiantuntevasta opastuksesta.

Työn ohjaajana työnantajan puolelta on toiminut Eila Niemelä, jonka panos työn syntymiseen on ollut merkittävä. Samoin tekn. lis. Harri Perunka ansaitsee kiitokset työhön myötävaikuttamisesta.

Erityiset kiitokset haluan esittää myös OY Mercantile Fastems AB:n Jani Granholmille hänen humoristisesta otteestaan ongelmaviidakon raivaamisessa. Samoin työhuoneeni asukit Arno Tuominen, Jarmo Lumpus ja Tomi Korpipää ansaitsevat maininnan kärsivällisyydestään ja auttavaisuudestaan.

Terveisten lähettämismahdollisuuksien ollessa näinkin harvinaiset haluan kiitellä myös kotiväkeäni ja tuttaviani. Kiitos.

Oulussa 12.3.1998

Mikko Holappa

SISÄLLYSLUETTELO

TIIVISTELMÄ **3**

ABSTRACT **4**

ALKULAUSE **5**

SISÄLLYSLUETTELO **6**

LYHENTEIDEN SELITYKSET **9**

1. JOHDANTO **11**

2. JOUSTAVAT VALMISTUSJÄRJESTELMÄT **12**

2.1 Johdatus joustaviin valmistusjärjestelmiin 12
2.2 Joustavan valmistusjärjestelmän tyypillinen rakenne 13
2.3 Tuotevariaatiot 16
2.4 Ohjelmistovaatimukset 17

3. CORBA-POHJAINEN HAJAUTUSALUSTA **19**

3.1 Käsitteet ja määritelmät 19
3.2 Syyt yleisen hajautusalustan käytölle 19
3.3 CORBAn tavoitteet 20
3.4 IDL-kuvauskieli 21
3.4.1 Olioviittaukset 24
3.4.2 Perintämekanismi 25
3.5 CORBAn arkkitehtuuri ja toiminnallisuus 26
3.5.1 Operaatiokutsut 28
3.5.2 ORB:n rakenne 28
3.5.3 ORB:n kutsutyypit 30
3.5.4 Oliosovitin 31
3.5.5 Rajapintavarasto 32
3.5.6 Toteutusvarasto 33
3.5.7 Toteutusten rekisteröinti 33
3.5.8 Dynaaminen kutsurajapinta 34
3.5.9 Yhteensopivuus 35
3.6 CORBA-palvelut 37

4. ORB-TOTEUTUKSET **40**

4.1 Orbix 40
4.1.1 Rakenne 40

4.1.2 Ohjelmiston kehitys 41
4.1.3 Olioiden toteutus 42
4.1.4 Palvelinten rekisteröinti 42
4.1.5 Operaatiokutsujen käsittely 43
4.1.6 Olioiden nimeäminen ja yhteyksien muodostaminen 44
4.1.7 DII 45
4.1.8 Lisäominaisuuksia 46
4.1.9 Resurssien käyttö ja hallinta 48
4.1.10 Suorituskyky 49
4.1.11 Kirjastot 51
4.1.12 Sovellusten siirrettävyys 52
4.1.13 Käyttökokemuksia 53
4.2 Chorus/COOL ORB 53
4.2.1 Toteutuksen rakenne 54
4.2.2 Resurssien käyttö ja hallinta 56
4.2.3 Suorituskyky 56
4.2.4 Kirjastot 58
4.2.5 Käyttökokemuksia 58
4.3 Muita ORB-toteutuksia 58

5. CORBAN SOVELTAMINEN JOUSTAVAAN VALMISTUSJÄRJESTELMÄÄN 60

5.1 Tarkoitus 60
5.2 CORBAn soveltamisen rajaus 60
5.3 Esimerkkijärjestelmän rakenne 60
5.4 Ohjelmistoarkkitehtuuri 61
5.4.1 ECA 61
5.4.2 Ohjelmistokomponentit 62
5.4.3 Käyttöliittymät 64
5.4.4 Tietokanta 65
5.5 Esimerkkijärjestelmän toiminnan looginen kuvaus 66
5.6 Ohjelmiston komponenttien hajautus 67
5.7 Esimerkkijärjestelmän liitynnät 69
5.8 Sovellusohjelmiston komponentointi 69
5.9 Kommunikointiprotokollat 71
5.9.1 Socket-yhteyden käyttö MFC-luokkakirjastolla 73
5.10 Laitteistoliitynnät 74
5.11 Sovittimet 76
5.12 Sovelluksen ja tietokannan integrointi 79
5.12.1 Tapahtumien hallinta 81
5.13 CORBA-järjestelmien toteutusperiaatteita 82
5.14 Jatkokehitys 85

5.14.1 Suorituskyvyn optimointi ... 85
5.14.2 Tietokannan ja tapahtumien hallinta ... 86
5.14.3 Ohjelmiston komponentointi ... 86

6. TULOSTEN ARVIOINTI ... 88

6.1 CORBAn vaatima koulutustarve... 88
6.2 CORBAn soveltuvuus joustaviin valmistusjärjestelmiin... 88
6.2.1 Paikkariippumattomuus ... 89
6.2.2 Resurssien käyttö... 90
6.2.3 Tuotevariointi ... 90
6.2.4 Hajautusalustan jatkokehitystarpeet ja tulevaisuuden näkymät ... 90

7. YHTEENVETO ... 92

LÄHTEET ... 93

LYHENTEIDEN SELITYKSET

API Application Programming Interface. Sovellusrajapinta.

BOA Basic Object Adapter. Perusoliosovitin.

CASE Computer Aided Software Engineering. Tietokoneavusteinen suunnittelu.

COM Component Object Model. Microsoftin komponenttioliomalli.

CONS Connection-mode Network Service. Yhteystyyppinen verkkopalvelu.

CORBA Common Object Request Broker Architecture. Yleinen ORB-arkkitehtuuri.

DCOM Distributed Component Object Model. Microsoft:n olioperustainen hajautusalusta.

DII Dynamic Invocation Interface. Dynaaminen kutsurajapinta.

DSI Dynamic Skeleton Interface. Dynaaminen palvelinrajapinta.

ECA Event, Condition, Action. Tapahtuma, ehto, toiminto.

FM Flexible Manufacturing. Joustava valmistus.

FMS Flexible Manufacturing System. Joustava valmistusjärjestelmä.

GIOP General inter-ORB Protocol. Yleinen ORB-yhteysprotokolla.

IDL Interface Definition Language. Rajapintojen kuvauskieli.

IFR Interface Repository. Rajapintavarasto.

IIOP Internet inter-ORB Protocol. Internet-verkon ORB-yhteysprotokolla.

IMR Implementation Repository. Toteutusvarasto.

IOR Interoperable Object Reference. Yhteensopiva olioviittaus.

IP Internet Protocol. Internet-protokolla.

IPC Inter Process Communication. Prosessien välinen kommunikointi.

LAN Local Area Network. Lähiverkko.

MFC Microsoft Foundation Classes. Microsoftin valmiit perusluokat.

NSAP Network Service Access Point. Verkkopalvelun yhteyskohta.

NSDU Network Service Data Unit. Verkkopalvelun tietoyksikkö.

OCX OLE Custom Control. Käyttäjän määrittelemä OLE-ohjain.

OMA Object Management Architecture. Olionhallinta-arkkitehtuuri.

OMG Object Management Group. CORBAn määritellyt ohjelmistoalan ryhmittymä.

ORB Object Request Broker. Oliokutsuvälittäjä.

OSI Open Systems Interconnection. Protokollastandardi.

OTS Object Transaction Service. Olioiden tapahtumanhallintapalvelu.

PC Personal Computer. Henkilökohtainen tietokone.

PLC Programmable Logic Controller. Ohjelmoitava logiikka.

POA Portable Object Adapter. Siirrettävä oliosovitin.

POS Persistent Object Service. Olioiden pysyvyyspalvelu.

RFC Request for Comment. Standardoimismenettely.

SII Static Invocation Interface. Staattinen kutsurajapinta.

TCP Transmission Control Protocol. Yhteysperustainen tiedonsiirtoprotokolla.

1. JOHDANTO

Nykyisten ohjelmistojen ongelmana on usein hallittavuuden puute ohjelmiston koon kasvaessa. Heikosta komponentoinnista, modulaarisuusasteesta ja ohjelmiston osien välisestä integroinnista johtuen pieni muutos osaan ohjelmistoa vaatii suuria ja hankalasti paikannettavissa olevia muutoksia muihin osiin järjestelmää. Lisäksi ohjelmiston kehitysvaiheessa ennakoidaan harvoin riittävästi tulevia muutos- ja laajennustarpeita. Järjestelmän toiminnallisten ominaisuuksien ja vanhojen osien lukumäärän lisääntyessä törmätään laajennettavuusongelmaan: suunnittelussa käytetyt arkkitehtuuri ja integrointiratkaisut eivät enää mahdollista järjestelmän jatkokehittämistä esimerkiksi suorituskyvyn putoamisen tai resurssien loppumisen vuoksi. Vaikka järjestelmä laajantamisen myötä säilyttäisikin toimintakykynsä, ohjelmiston kompleksisuus ei enää ole ihmisen hallittavissa.

Joustava valmistusjärjestelmä (Flexible Manufacturing System, FMS) on kompleksinen tietokoneohjattu laitteisto- ja ohjelmistokokoonpano. Sen tarkoitus on valmistaa haluttuja tuotteita lyhyillä läpimenoajoilla ja alhaisilla yksikkökustannuksilla [2]. FMS:t ovat niiden valmistajan kannalta jokaiselle asiakkaalle räätälöityjä työläitä projekteja. Ilman järkevien ohjelmistoarkkitehtuurien ja komponentointimenetelmien käyttöä yllä mainitut ongelmat alentavat ohjelmistokehityksen tuottavuutta ja johtavat aikataulun ja budjetin ylittymiseen.

CORBA-spesifikaatio (Common Object Request Broker Architecture) määrittelee oliosuuntautuneen ohjelmistoarkkitehtuurin, joka tarjoaa raamit ohjelmiston komponentoinnin toteuttamiseen ja komponenttien väliseen kommunikointiin [5]. CORBA-spesifikaation mukaisten ohjelmistojen pitäisi olla helppoja toteuttaa ja integroida, joustavia ja mukautuvia, helppoja uudelleenkäyttää, laajennettavia sekä kohtuullisella vaivalla ylläpidettäviä.

Tässä diplomityössä tutkittiin CORBAn käyttämistä ja soveltamista joustavan valmistusjärjestelmän ohjelmistoväylänä. Tarkastelun alla olivat CORBA-spesifikaatioon perustuvan hajautetun ohjelmiston toteuttaminen, CORBAn ohjelmistokehitykseen ja ohjelmiston suorituskykyyn mukanaan tuomien muutosten arvioiminen sekä sen mahdollisten puutteiden ja rajoitteiden selvittäminen.

2. JOUSTAVAT VALMISTUSJÄRJESTELMÄT

Tässä luvussa esitellään lyhyesti joustavia valmistusjärjestelmiä, jotta lukija saa käsityksen siitä, millaiseen ympäristöön ja sovellusalueeseen työn kokeellinen osa pohjautuu.

2.1 Johdatus joustaviin valmistusjärjestelmiin

Joustava valmistusjärjestelmä (FM-järjestelmä) on laitteisto- ja ohjelmistokokonaisuus, jonka tarkoitus on valmistaa raaka-aineista valmiita tuotteita. FM-järjestelmä on vaihtelevassa määrin automatisoitu valmistusprosessi, jossa ihmisen tehtävä on osittainen ohjaus ja käsityövaiheiden suorittaminen.

Jäykissä tuotantolinjoissa tuotteen läpikäymä valmistusprosessi on alusta loppuun asti ennalta määrätty eikä järjestelmän konfigurointi uuden tuotteen valmistukseen onnistu ilman merkittäviä muutoksia tuotantolinjaan ja sen ohjaukseen. FM-järjestelmillä voidaan valmistaa useita erilaisia tuotteita vaihtelevilla valmistusprosessin vaiheistuksilla ja reiteillä. Vaihtoehtoisten läpäisyreittien lukumäärän kasvaessa läpimenoaikojen keskiarvot ja niiden hajonnat pienenevät [1, s. 5]. Valmistettavan tuotteen, järjestelmän olosuhteiden tai laitteiston muuttuessa järjestelmä ei vaadi kohtuutonta uudelleenkonfigurointia eikä järjestelmän suorituskyky laske liikaa, vaan järjestelmä mukautuu muutoksiin.

Kun FM-järjestelmässä on tuki tilausten käsittelylle, kappaleiden valmistaminen on mahdollista silloin, kun niille on kysyntää. Edistyneet ajastusalgoritmit järjestelmän ohjauksessa nostavat kalliiden koneiden käyttöastetta, ja tehokkaiden virheenkäsittelymekanismien avulla järjestelmän vaurioituminen sekä uudelleenkäynnistämisen aiheuttamat kustannukset saadaan minimoitua. Lisäksi valmistuksen yksikkökustannukset laskevat valmistusprosessin systematisoinnin ja resurssien käytön optimoinnin myötä. Eräs FM-järjestelmän määritelmä on seuraava [2, s. 12]:

"The full FMS installation is one in which a process is put under total computer control to produce a variety of products within the stated capability and to a pre-determined schedule."

Uusien teknologioiden mukanaan tuomien ongelmien ja kustannusten vastapainoksi FM-järjestelmillä on tarjota esimerkiksi seuraavia etuja verrattuna perinteiseen valmistusprosessiin [2, s. 19 - 23]:

- Nopea vaste markkinoiden vaatimuksiin. Tuotetta voidaan valmistaa silloin, kun sille on kysyntää. Valmistettavan tuotteen tyyppiä ja määrää on helpompi hallita.

- Tuotteiden tasaisempi ja parempi laatu. Automaatioasteen noustessa ihmisen tekemien virheiden määrä laskee.

- Vaihtoehtoisten reittien lukumäärän kasvaessa läpimenoajat lyhenevät.

- Pääresurssien, kuten koneiden ja työkalujen, käyttöaste kasvaa ajastusalgoritmien ja automaatioasteen nousun myötä.

- Ihmisen suorittamien työvaiheiden vähentyessä työvoimakustannukset alentuvat.

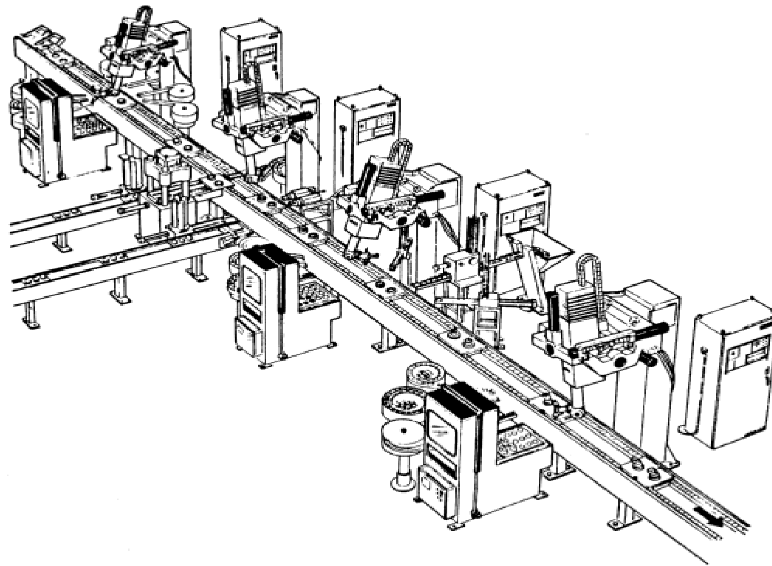
- Integroitavuus CAD-työkaluihin helpottuu.

- Järjestelmän kirjanpito ja inventointi helpottuvat integroitujen tietojärjestelmien ja pienempien varastojen myötä.

- Työkalujen asetusajat lyhenevät automatisoidun työkalujen käsittelyn ja tietokoneen ohjaaman rinnakkaisen toiminnan avulla.

2.2 Joustavan valmistusjärjestelmän tyypillinen rakenne

FM-järjestelmän laitteisto koostuu kuljettimista, varastoista ja joukosta laitteita sekä käsityöasemia. Kuljetin siirtää kuljetusalustan laitteelle, missä alustalla sijaitsevien kappaleiden työstö tapahtuu. Alustojen käsittelyä varten järjestelmässä on latausasemia, joilla ihminen tai robotti varustaa alustan valmiiksi työstökiertoa varten sekä purkaa kierrosta valmistuneen alustan. Kuva 1 esittää linjamuotoista FM-järjestelmää.



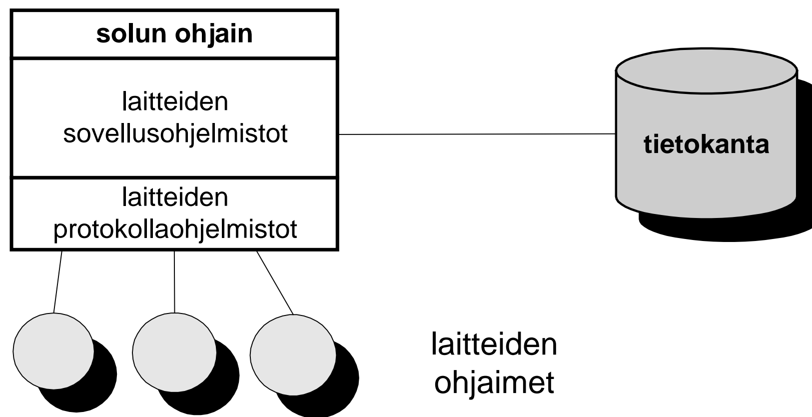
Kuva 1. Linjamuotoinen FM-järjestelmä [3].

FM-järjestelmää ohjaava ohjelmisto voidaan toteuttaa usealla erilaisella arkkitehtuurilla. Arkkitehtuurin valintaan vaikuttavat muun muassa tarvittavan kommunikoinnin määrä, vasteaikavaatimukset, järjestelmän laajuus ja ohjauksen kohdentaminen. Erään määritelmän mukaan arkkitehtuurit voidaan jakaa karkeasti seuraaviin kolmeen ryhmään: keskitettyyn, verkotettuun ja täysin hajautettuun arkkitehtuuriin [4].

Keskitetyssä arkkitehtuurissa solun[1] ohjausjärjestelmä on toteutettu täysin yhdellä, esimerkiksi yleiskäyttöisellä minitietokoneella. Tietokone käsittelee tietokantaa ja vastaa kaikista solun ohjaustoiminnoista. Lisäksi kaikkien solun laitteiden vaatimat sovellus- ja protokollatehtävät suoritetaan tällä keskuskoneella. Ongelmana on laitteiden kommunikointimekanismien monipuolisuuden aiheuttama kompleksisuus solun ohjaimen laitteistossa ja ohjelmistossa ja kaikkien laitteiden johdottaminen mahdollisesti häiriöiltä suojattavaan keskuskoneeseen. Laitteen vasteaikaan ei vaikuta vain tiedonsiirtoyhteyden nopeus vaan myös solun ohjaimen nopeus sen vastatessa kaikista järjestelmän ohjaustoiminnoista. Yhden prosessorin ohjatessa kaikkia sovelluksia sovellusten prioriteettien ja ajastusalgoritmin valinnan merkitys kasvaa. Kuva 2 esittää keskitettyä järjestelmää.

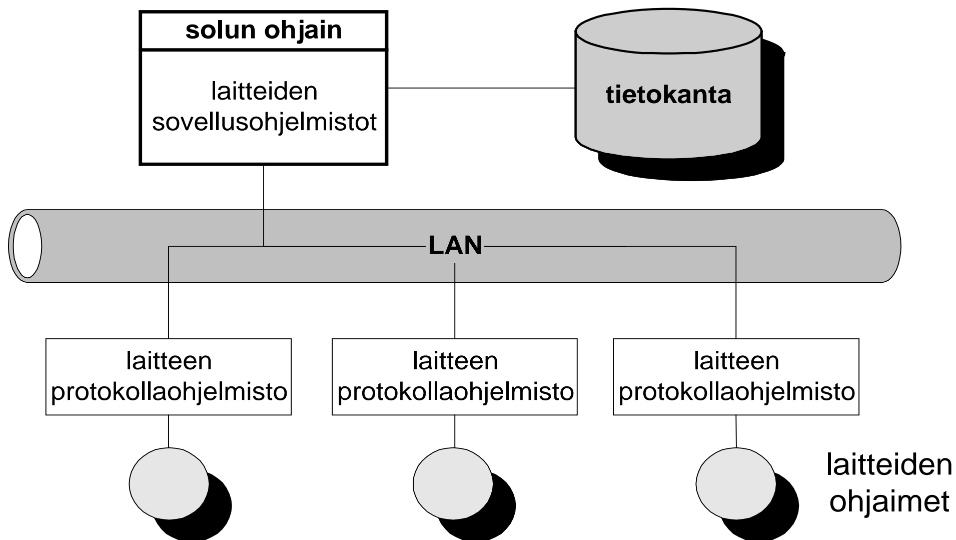
---

[1] Tässä solu tarkoittaa yhden tai useamman laitteen ja materiaalin varastointipaikan muodostamaa loogista kokonaisuutta.



Kuva 2. Keskitetty solunohjausjärjestelmä.

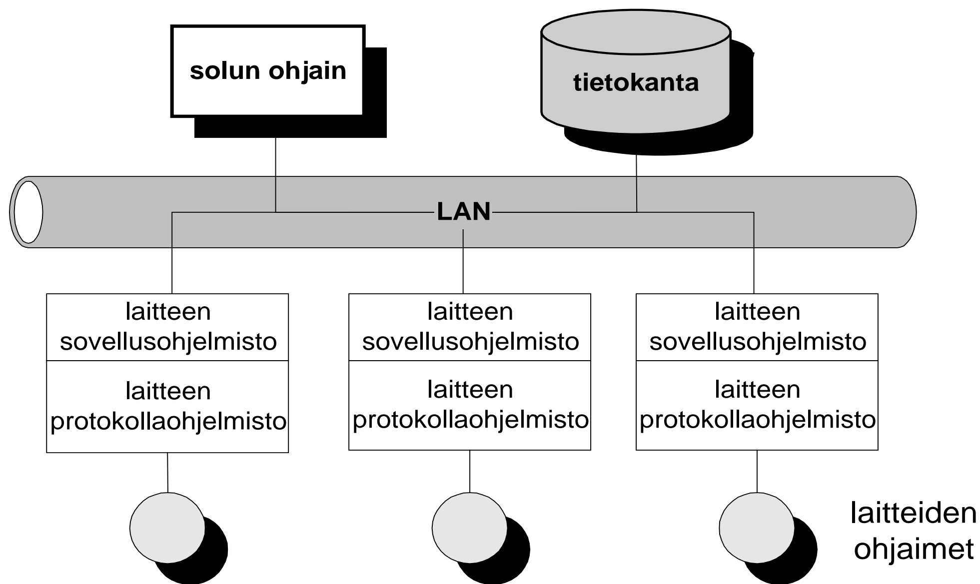
Verkotettu arkkitehtuuri käyttää nopeaa, virheenkorjausprotokollalla varustettua lähiverkkoa ja mikroprosessoreita ohjausjärjestelmän hajautukseen. Solun ohjaimena toimii minitietokone tai mahdollisesti yksi tai useampi mikrotietokone. Laiteriippuvasta ohjelmistosta protokollaosuus on hajautettu lähiverkkoon kiinnittyville mikroprosessoreille. Sovellusohjelmat sisältävä solun ohjain on vastuussa ohjaustoiminnoista ja tietokannan hallinnasta. Kun sovellus antaa ohjauskomennon kohdelaitteelle, se lähettää viestin lähiverkon yli mikroprosessorissa suorittuvalle protokollaohjelmistolle. Mikroprosessorilta viesti välittyy edelleen kohdelaitteelle.Kuva 3 esittää verkotettua järjestelmää.



Kuva 3. Verkotettu solunohjausjärjestelmä.

Täysin hajautetussa arkkitehtuurissa solun ohjain, tietokanta ja kaikki laitteiden sovellusohjelmistoja suorittavat yksiköt on kiinnitetty suoraan lähiverkkoon. Solun ohjain on vastuussa vain toiminnoista, jotka liittyvät solun ohjaamiseen korkeamman

tason kokonaisuutena, ja laitteen sovellus- ja protokollaohjelmisto on kohdennettu laitetta ohjaavalle suoritinyksikölle. Tällä tavoin yksittäiset laitteet saadaan toisistaan riippumattomiksi, ohjelmiston kehittäminen modularisoituu ja laitteiden sijainnit ovat läpinäkyviä sovellusohjelmiston kannalta. Laitetta ohjaavan sovelluksen ja laitteen itsensä välinen kommunikointi ei enää riipu lähiverkon kuormituksesta tai nopeudesta eikä solun ohjaimen vasteajoista. Kuva 4 esittää täysin hajautettua arkkitehtuuria.



Kuva 4. Täysin hajautettu solunohjausjärjestelmä.

2.3 Tuotevariaatiot

FM-järjestelmän toimittajan kannalta jokainen järjestelmätilaus on yleensä oma projektinsa analysointi-, suunnittelu-, toteutus- ja testausvaiheineen. Tämä johtuu siitä, että varsinaista järjestelmän "prototyyppiä" ei ole, vaan laajuudeltaan ja toiminnallisuudeltaan erilaisia järjestelmiä on periaatteessa ääretön määrä. Erilaiset koneet, niiden lukumäärä, ohjaukseen käytettävä tiedonsiirtoväylä ja -protokolla ja asiakkaan halu maksaa erilaisista ominaisuuksista ovat merkittävimpiä muuttujia, jotka aiheuttavat tuotevariaatioiden lukumäärän kasvamisen erittäin suureksi. Tapa toteuttaa järjestelmä tuotettavan artikkelin tai artikkeleiden ympärille haittaa myös prosessin systematisoitumista.

Järjestelmissä esiintyvä variaatioiden suuri määrä asettaa vaatimuksia järjestelmien ohjausohjelmistojen toteuttamisessa käytettäville ohjelmistoarkkitehtuureille ja suunnittelumenetelmille. Arkkitehtuurin ja toteutustyökalujen tulee tukea järjestelmän ajonaikaista joustavuutta, minimoida ohjelmien jatkokehitystarpeiden mukanaan tuomia kustannuksia sekä mahdollistaa laajennettavat järjestelmät.

Muokkauskustannuksia voidaan vähentää luomalla sovellusympäristölle sovelluskehys (framework), jonka määrittelemiä konsepteja kaikki sovelluksen komponentit noudattavat. Kun järjestelmän komponentit jaetaan modulaarisiin, itsenäisesti muunneltaviin yksiköihin, niiden looginen toiminnallisuus on paikallistettavissa helpommin. Siten ohjelmiston muuntaminen erilaisiin kohdejärjestelmiin sopivaksi voidaan hoitaa vähemmällä työllä. Erilaisten järjestelmien välisiä eroavaisuuksia ja samankaltaisuuksia ei ole kuitenkaan mahdollista kartoittaa projektin puitteissa, vaan ohjelmiston komponentoinnin tulee olla projekteista erillinen prosessi.

2.4 Ohjelmistovaatimukset

Haastattelujen mukaan FM-järjestelmien luonne asettaa seuraavia vaatimuksia järjestelmää ohjaavalle ohjelmistolle:

- **Suorituskyky**. Useimmat FM-järjestelmät voidaan luokitella pehmeiksi reaaliaikajärjestelmiksi, joissa vasteaikojen ylittyminen ei aiheuta vakavia seurauksia, kuten ihmishenkien menetyksiä. Järjestelmän ohjaajalle näkyvien viiveiden eliminoimiseksi vasteaikojen tulee silti olla millisekuntien luokkaa ja järjestelmien pitää pystyä siirtämään kohtalaisen suuria tietomääriä.

- **Turvallisuus**. Ohjelmiston loogisten virheiden aiheuttamat aineelliset vahingot voivat nousta suuriksi. Virhetilanteet käsitellään usein laitteistotasolla, mutta kalliiden uudelleenkäynnistysten välttämiseksi perusteellinen virhetilanteiden havaitseminen ja käsittely ovat tarpeen myös ylemmällä tasolla.

- **Palautettavuus**. Sähkökatkosten ja muiden yllättävien tilanteiden varalta järjestelmän tilan pitää olla kokonaisuudessaan palautettavissa ja uudelleenkäynnistettävissä katkosta edeltäneeseen tilaan.

- **Joustavuus**. Järjestelmän toiminnan pitää olla konfiguroitavissa ajon aikana siten, ettei sovelluskomponentteihin tarvitse tehdä muutoksia.

- **Laajennettavuus**. Ohjelmistoarkkitehtuurin tulee tukea järjestelmän laajennettavuutta sekä horisontaalisesti että vertikaalisesti. Horisontaalinen laajennettavuus tarkoittaa järjestelmän komponenttien lukumäärän kasvattamista ja vertikaalinen laajennettavuus järjestelmän toiminnallisuuden lisäämistä.

- **Muokattavuus**. Ohjelmiston suunnittelussa tulee ottaa huomioon tulevien muutosten ja jatkokehityksen aiheuttama muutostyö. Modulaarinen, oliosuuntautunut mallinnus ja toteutus helpottavat järjestelmien muokkaamista eristämällä komponentit

toisistaan, minimoimalla komponenttien väliset riippuvuudet ja edistämällä uudelleenkäyttöä.

- **Tietorakenteiden hallittavuus.** Tuotteiden valmistusohjeiden ja asiakkaiden tekemien tilausten tietorakenteet saattavat olla hyvin monimutkaisia. Tietojärjestelmän tulee tarjota helposti hallittava tuki tietoyksiköiden välisille assosiaatioille.

Ohjelmistolle asetettavat vaatimukset jakaantuvat kahteen osa-alueeseen: ohjelmiston ajonaikaisen toiminnan ja suorituskyvyn tulee olla riittävän laadukasta, ja toisaalta ohjelmistonkehitykseen käytettävien työkalujen tulee tarjota tuki mahdollisimman tuottavalle työlle. CORBA ja sen toteutukset pyrkivät täyttämään näitä molempia tavoitteita.

3. CORBA-POHJAINEN HAJAUTUSALUSTA

3.1 Käsitteet ja määritelmät

Teollisuusstandardiksi luokiteltava OMG:n CORBA-spesifikaatio määrittelee oliosuuntautuneen viitekehyksen (reference model) hajautetuille ja heterogeenisille ohjelmistoille. CORBA-standardin mukainen ORB-tuote (Object Request Broker) on ohjelmistoväylä, joka tarjoaa mekanismit hajautettujen olioiden väliseen kommunikointiin ja helpottaa hajautettujen järjestelmien kehittämistä automatisoimalla alemman tason verkko-ohjelmoinnin. CORBA-standardin mukaisten ohjelmistojen tulisi kyetä kommunikoimaan riippumatta siitä, mikä on osapuolten sijainti, toteutusympäristön laitteisto, käyttöjärjestelmä tai ohjelmointikieli. ORB on vastuussa olion paikantamisesta, sen valmistamisesta asiakkaan pyynnön vastaanottamiseen, pyynnön välittämisestä ja mahdollisen palautusarvon palauttamisesta asiakkaalle. [5]

Jatkossa käytetään seuraavia nimityksiä hajautetun ohjelmiston komponenttien yhteydessä:

- **Rajapinta** on CORBAn IDL-kuvauskielellä tehty ohjelmistokomponentin (olion) määrittelykuvaus, joka näkyy komponenttia käyttäville asiakkaille. Rajapinta toimii arkkitehtuurin ja toteutuksen välisenä linkkinä.

- **Olio** on rajapinnan toiminnallisuuden tarjoava jollakin ohjelmointikielellä toteutettu ohjelmiston osa.

- **Prosessi** on suoritettavana oleva ohjelma, jolla on oma muistiavaruus.

- **Palvelin** on yhtä tai useampaa oliota ylläpitävä prosessi.

- **Asiakas** on yhden tai useamman rajapinnan operaatioita käyttävä ohjelmistokomponentti.

3.2 Syyt yleisen hajautusalustan käytölle

CORBA-standardin määritteleminen on lähtenyt tarpeesta kehittää paremmin hallittavissa ja ylläpidettävissä olevia ohjelmistoja hajautettuihin järjestelmiin lyhyemmällä koulutusajalla, vähemmällä työllä ja siten pienemmillä kustannuksilla. Vaikka nykyiset käyttöjärjestelmät ja laitteistot eroavaisuuksistaan huolimatta ovat kytkettävissä yhteen ja niiden välinen kommunikointi on mahdollista, sovellustasolla

näitä alustaeroja ei ole vielä onnistuttu täysin piilottamaan. Tietoverkon yli toimivaa sovellusta tehdessään ohjelmoija joutuu ottamaan huomioon nämä eroavaisuudet, mikäli haluaa tuotteensa toimivan prosessirajojen lisäksi myös käyttöjärjestelmä- ja laiterajojen yli. Kommunikoinnin toisen osapuolen toteuttavassa järjestelmässä saatetaan käyttää eri laitteistoa, käyttöjärjestelmää, verkkoprotokollaa tai ohjelmointikieltä.

Jotta ohjelmoija saisi sovelluksensa toimimaan eri ympäristöjen välillä, hänen täytyy tuntea matalan tason mekanismien toimintaa. Opeteltavia asioita ovat käyttöjärjestelmän toimintatapa, tiedonsiirtoprotokollan toiminta ja konfigurointi, järjestelmien väliset erot tietotyyppien tavujärjestyksessä sekä kääntäjän tapa käsitellä ohjelmoijan määrittelemiä tietotyyppejä. Tämän alemman tason tietämyksen hankkiminen kuluttaa varsinaisen ongelma-alueen käsittelyyn käytössä olevia resursseja.

Ohjelmistokomponenttien välisten rajapintojen määrittäminen on usein puutteellista, ja rajapinnat on kuvattu erilaisilla menetelmillä. Siten ohjelmistojen yhteenliittäminen on työlästä ja toteutusten muuttaminen vaikeaa. Hajautettujen sovellusten välisten hyötyviestien pakkaaminen yleiseen muotoon (esim. socket-yhteyden yli välitettävät TCP/IP-viestit) on virhealtista, koska käännösaikainen tyypintarkistus ei ole mahdollista. Kommunikointi toteutetaan tietotasolla esimerkiksi ohjelmoijan määrittämien tietueiden avulla. Korkeammalla abstraktiotasolla kommunikointi toteutettaisiin tiettynä toiminnallisuutena, esimerkiksi funktiokutsuina. Lisäksi ohjelmistokomponentin rajapinta on usein sidottu suoraan toiminnalliseen toteutukseen, jolloin toteutuksen muuttuessa myös kaikkia rajapintaa käyttäviä komponentteja joudutaan muuttamaan.

Arkkitehtuurien standardoinnin ja niiden käytön puute tuottaa niin sanottuja *ad-hoc*-järjestelmiä, jotka ovat huonosti dokumentoituja ja kalliita ylläpitää [6]. Ohjelmistossa ei käytetä ennalta sovittuja rakenteita ja mekanismeja, vaan toiminnallisuus toteutetaan nopeasti epäyhtenäisiä menettelytapoja käyttäen. Tämän seurauksena saadaan ohjelmistoja, jotka ovat vaikeita ymmärtää, muunnella ja uudelleenkäyttää.

3.3 CORBAn tavoitteet

CORBAn tavoitteet ovat lähteneet edellä mainituista ongelmista, joita syntyy käytettäessä perinteisiä menetelmiä hajautettujen ohjelmistojen kehittämiseen. Kunnianhimoisia, osin vielä toteutumattomia tavoitteita ja ominaisuuksia CORBA-standardiin perustuvalle ORB-tuotteelle ovat [5]:

- ympäristön (käyttöjärjestelmän, protokollan, ohjelmointikielen) läpinäkyvyys

- lähdekoodien siirrettävyys
- eri ORB-toteutusten yhteensopivuus
- rajapinnan erottaminen toteutuksesta
- uudelleenkäyttö, joustavuus ja
- komponenttien paikkaläpinäkyyys.

Osa näistä tavoitteista ja ominaisuuksista tähtää sovelluskehityksen tuottavuuden lisäämiseen yhtenäistämällä menettelytapoja ja hyödyntämällä oliosuuntautunutta suunnittelua ja mahdollisesti toteutusta (ORB:n ei tarvitse tukea olio-ohjelmointia ollakseen CORBA-yhteensopiva). Toinen näkökulma on ajonaikaisen toiminnan hyvyys, jota ORB-toteutukset pyrkivät määritelmän vaatimusten lisäksi esimerkiksi suorituskyvyn osalta optimoimaan.

3.4 IDL-kuvauskieli

IDL on CORBAn tärkein osa. Se on OMG:n vuonna 1991 määrittelemä kuvauskieli sovellusohjelmistojen rajapinnoille ja on säilynyt miltei muuttumattomana tähän päivään asti. Koska IDL on perusta jokaiselle OMG:n spesifikaatiolle, OMG:n täytyy pitää IDL stabiilina tai se joutuu muuttamaan omia standardejaan ja kaupallisia tuotteitaan. IDL määrittelee vain rajapinnan CORBA-pohjaiseen olioon: se ei rajoita millään tavalla olion toteutusta, eikä rajapintaa käyttävän asiakkaan tarvitse olla tietoinen rajapinnan toteuttavan olion yksityiskohdista. [7, s. 21 - 22]

IDL on ohjelmointikielestä riippumaton. Se tukee useampaa standardoitua kielisidontaa (language binding), joista tärkeimpiä ovat C++, C, SmallTalk, Ada, OLE (esim. Visual Basic) ja COBOL. Määrittely on menossa myös Javalle, Eiffelille ja Objective C:lle [8]. Kun rajapintamääritelmä on kirjoitettu IDL:n syntaksin mukaisesti kerran, sitä ei tarvitse uudelleenkirjoittaa toteutuskielen vaihtuessa. Standardien kielisidontojen ansiosta IDL on myös alustariippumaton kuvauskieli.

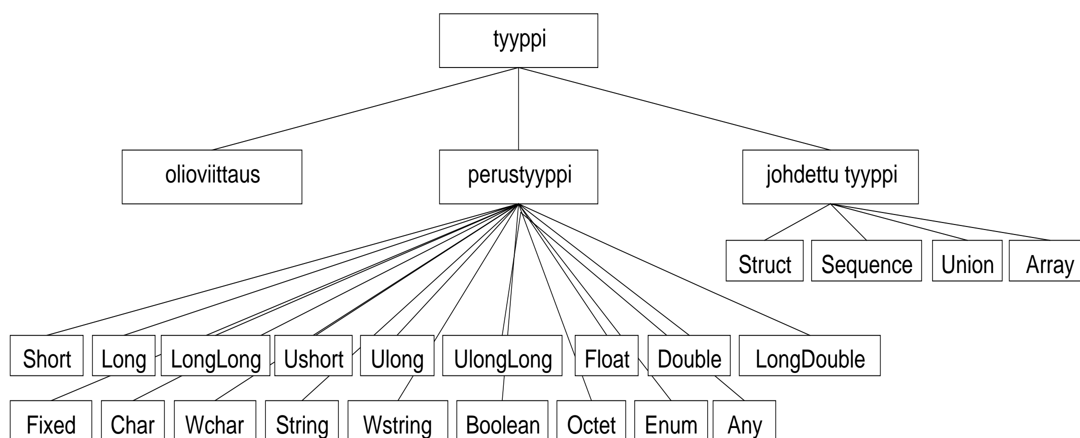
IDL-kuvauskieltä käytetään nimensä mukaisesti olioiden rajapintojen kuvaamiseen: siinä ei ole muuttujia, lausekkeita tai *if / while / for* -rakenteita. Sen syntaksi on C++-kielen ja Javan tyylinen mutta helpompi oppia, koska sen tarvitsee ilmaista vain pieni osa tavallisen ohjelmointikielen rakenteista. [8]

IDL:n tärkein rakenne on rajapinta (interface). Se vastaa C++- tai SmallTalk-kielen luokkaa tai Javan rajapintaa. Rajapinnan avulla määritellään ohjelmistokomponentin asiakkaille näkyvät operaatiot sekä attribuutit. Operaatiot vastaavat C++:n metodeja ja attribuutit tietojäseniä. Operaatioiden parametreille voidaan tietotyypin lisäksi määritellä käyttötapa. Käyttötavan mukaan parametri välittää tietoa joko asiakkaalta palvelimelle, palvelimelta asiakkaalle tai molempiin suuntiin. Taulukko 1 esittää parametrien käyttötapoja.

Taulukko 1. IDL-operaation parametrityypit.

Parametrin tyyppi	Käyttötarkoitus
in	tiedon välitys asiakkaalta palvelimelle
out	tiedon välitys palvelimelta asiakkaalle
inout	tiedon välitys sekä asiakkaalta palvelimelle että palvelimelta asiakkaalle
palautusarvo	operaation palautusarvo palvelimelta asiakkalle

Lisäksi IDL:n ominaisuuksiin kuuluvat rajapintojen perintämekanismi, poikkeuksien (exceptions) määrittely, perus- ja yhdistetietotyypit, esikääntäjän direktiivit, vakiomäärittelyt, moduulimäärittelyt ja etukäteisesittelyt. Kuva 5 esittää IDL:n tukemia tietotyyppejä.



Kuva 5. IDL-tietotyypit [8].

Jotta IDL olisi toteutusriippumaton kieli, se määrittelee jokaiselle tietotyypille arvoalueen. IDL-kielisidonta varmistaa, että IDL-tietotyyppi on jokaisessa kohdejärjestelmässä samanlainen. Taulukko 2 esittää IDL:n perustyyppien mahdolliset arvoalueet.

Taulukko 2. IDL-tietotyypit ja niiden arvoalueet [8, 9].

IDL-tietotyyppi	Kuvaus
Short	-2^{15} ... $2^{15}-1$ (16-bittinen)
Long	-2^{31} ... $2^{31}-1$ (32-bittinen)
Longlong	-2^{63} ... $2^{63}-1$ (64-bittinen)
Ushort	0 ... $2^{16} - 1$ (16-bittinen)
Ulong	0 ... $2^{32} - 1$ (32-bittinen)
UlongLong	0 ... $2^{64} - 1$ (64-bittinen)
Float	IEEE-liukuluku (32-bittinen)
Double	kaksoistarkkuuden tarjoava IEEE-luku (64-bittinen)
LongDouble	kaksoislaajennettu IEEE-luku (mantissa vähintään 64 bittiä, etumerkkibitti ja eksponentti vähintään 15 bittiä)
Fixed	kiinteän pisteen desimaalinumero 31:een merkitsevään numeroon asti
Char	merkki, 8-bittinen tietotyyppi (ASCII:n ISO-Latin-alijoukko)
Wchar	laajennettu merkki (wide character)
String	muuttuvanpituinen merkkijono (pituus päätetään ajonaikaisesti)
Wstring	laajennettu muuttuvanpituinen merkkijono (wide character string)
Boolean	TOSI tai EPÄTOSI
Octet	8-bittinen tietotyyppi, jolle ei suoriteta mitään muunnoksia siirron aikana
Enum	luetellut tyypit (järjestetty tunnistesekvenssi)
Any	tietotyyppi, joka voi esittää mitä tahansa IDL-tietotyyppiä

Rakenteisista tietotyypeistä *struct* on loogisesti samanlainen kuin tavallinen tietuetyyppi. *Sequence* on muuttuvanpituinen taulukko tietyntyyppisiä tietoalkioita, josta siirretään vain sen kulloinkin sisältämät alkiot. *Union* on kuten tavallinen yhdistetyyppi, ja *array* kuten tavallinen kiinteänpituinen taulukko, joka siirretään aina kokonaisuudessaan.

Erikoisia tietotyyppejä ovat octet, any ja olioviittaus (object reference). *Octet* on 8-bittinen tietotyyppi, jolle ei suoriteta mitään muunnosoperaatioita siirron aikana. Se on tärkeä tietotyyppi siirrettäessä esimerkiksi tiedostoja, kuvia tai muita objekteja, jotka

koostuvat binääritiedosta. Minkään muun tietotyypin “koskemattomuutta” ei taata siirron aikana. Tietotyyppi *any* voi sisältää ajon aikana minkä tahansa IDL:n perustyypin tai johdetun tietotyypin. Any-tietotyypissä on kentät tyypin tunnisteelle sekä osoitin itse tietoon. Any mahdollistaa yleiskäyttöisten operaatioiden määrittelemisen, joissa muodolliset parametrit ovat tyyppiä any, mutta varsinaiset parametrit päätetään vasta ohjelman suorittamisen aikana.

3.4.1 Olioviittaukset

Olioviittaus on tietotyyppi, joka vastaa loogisesti C++-kielen olion osoitinta. Sen voi välittää operaation parametrina, ja se määrittää yksikäsitteisesti jonkin CORBA-rajapinnan toteuttavan olion. Eri ORB-toteutukset voivat valita eri esitystavan olioviittauksille [10]. Esimerkiksi Orbixin olioviittaukseen on koodattuna olion nimi (*marker*), olion toteuttaman rajapinnan tyyppi (*interface*), oliota ylläpitävän palvelimen nimi (*server*) sekä isäntäkoneen nimi (*host*), jossa palvelinta suoritetaan. Kaikkien ORB-toteutusten pitää tarjota sama kielisidonta olioviittaukselle tietyllä ohjelmointikielellä, jotta tietyllä ohjelmointikielellä kirjoitettu ohjelma voi käsitellä olioviittauksia ORB-riippumattomalla tavalla [5].

CORBA 2.0 määrittelee myös eri ORB-toteutusten välillä toimivan olioviittauksen, IOR:n (Interoperable Object Reference). ORB-siltausten vuoksi seuraavat tiedot olioviittauksessa ovat tarpeen [5]:

- *Onko olioviittaus null?* Null-olioviittaus pitää vain siirtää, eikä sen tarvitse tukea operaatiokutsuja.

- *Mitä tyyppiä olio on?* Monet ORB-toteutukset tarvitsevat tiedon olion tyypistä.

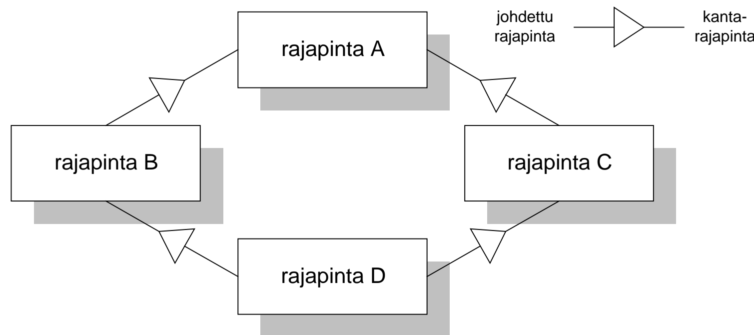
- *Mitä protokollia tuetaan?* Jotkut ORB-toteutukset tukevat olioviittauksia, jotka toimivat useammalla toimialueella (domain). Näin asiakkaille mahdollistetaan tehokkaimman kommunikointimekanismin valinta.

- *Mitä ORB-palveluja on tarjolla?* Operaatiokutsuun voi liittyä eri ORB-palveluja ja tieto palveluista voi vähentää viivettä niiden valinnassa.

TCP/IP:n päällä toimivan Internet-kohtaisen yhdysprotokollan (Internet Inter-ORB Protocol, IIOP) tapauksessa IOR sisältää palvelimen isäntäkoneen IP-osoitteen sekä TCP/IP-portin, jonka kautta olion toteuttavaan palvelimeen voidaan ottaa yhteys.

3.4.2 Perintämekanismi

IDL tukee rajapintojen moniperintää. Rajapinta voidaan johtaa toisista rajapinnoista, joita kutsutaan kantarajapinnoiksi. Johdettu rajapinta perii kaikki kantarajapinnan elementit (vakiot, tyypit, attribuutit, poikkeukset, operaatiot) ja voi lisätä uusia elementtejä. Kuva 6 esittää esimerkin moniperinnästä. [5, s. 3,-,15]



Kuva 6. Esimerkki moniperinnästä.

Moniperinnän yhteydessä ei ole sallittua periä useampaa rajapintaa, joilla on sama operaation tai attribuutin nimi. Myös operaation tai attribuutin nimeäminen uudelleen johdetussa rajapinnassa on kiellettyä. Rajoitukset johtuvat siitä, että operaatioita käyttävät asiakkaat määrittelevät kutsutun operaation nimen kutsussa ohjelman suorituksen aikana. Tällöin jokaisella tiettyyn olioon liittyvällä operaatioilla tulee olla yksikäsitteinen nimi. Tulevat CORBA-spesifikaation versiot lieventävät mahdollisesti tätä rajoitusta. [5]

IDL-perintä eroaa huomattavasti C++-perinnästä. C++-kielellä on muunnelmia perinnässä, kuten private, protected, public ja virtual. Puhtaana rajapintojen kuvauskielenä IDL ei ota kantaa tällaisiin toteutusteknisiin asioihin. [8]

Operaatioiden ja attribuuttien ylikirjoittaminen johdetulle rajapinnalle on mahdollista rajapintojen toteutuksessa. Esimerkiksi johdetun rajapinnan C++-toteutusluokka voi määritellä kantaluokan kanssa samannimisen ja samat parametrit omaavan metodin, jonka toteutus poikkeaa kantaluokan vastaavasta. Kuva 7 esittää (Orbixin tapa) tällaista tilannetta.

```
Rajapinnat

// kantarajapinta

interface engine {

        void inc();
        void dec();

};

// johdettu rajapinta

interface turbo : engine {

        // uusi operaatio

        void boost();

};
```

```
Rajapintojen C++ -toteutukset (Orbix)

class engine_i, public virtual engineBOAImpl {                          // kantarajapinnan toteutus
public:
        double speed;
        void inc(CORBA::Environment &env) {
                speed++;
        }
        void dec(CORBA::Environment &env) {
                speed--;
        }
};

class turbo_i : public virtual turboBOAImpl, public virtual engine_i {  // johdetun rajapinnan toteutus
public:
        void boost(CORBA::Environment &env) {
                speed += 4;
        }
        void inc(CORBA::Environment &env) {                             // inc( ) -operaation ylikirjoitus
                speed += 2;
        }
};
```

Kuva 7. Koodiesimerkki IDL- ja C++-perinnästä.

Jos asiakas ottaa yhteyden esimerkin mukaiseen engine-tyypin rajapinnan toteuttavaan olioon, sen saama olioviittaus saattaa viitata joko johdetun tai kantaluokan rajapinnan toteuttavaan olioon. Asiakkaan kutsuessa operaatiota inc(), kutsutaan olioviittauksen viittaaman todellisen olion toteutusta operaatiolle inc(). Tämä on esimerkki polymorfisesta sidonnasta, jossa asiakkaan ei tarvitse tietää kohdeolion toteutuksen tarkkaa tyyppiä [6].

3.5 CORBAn arkkitehtuuri ja toiminnallisuus

Avain CORBA-arkkitehtuurin rakenteen ymmärtämiseen on CORBA-viitemalli (CORBA Reference Model), joka koostuu seuraavista komponenteista [5]:

- **Oliokutsuvälittäjä (ORB)** mahdollistaa hajautettujen olioiden väliset paikkaläpinäkyvät kutsut, kutsujen vastaanottamisen ja vasteiden saamisen kutsuihin. ORB on perusta hajautetuista olioista koostuvien sovellusten rakentamiselle ja niiden väliselle kommunikoinnille heterogeenisissa järjestelmissä.

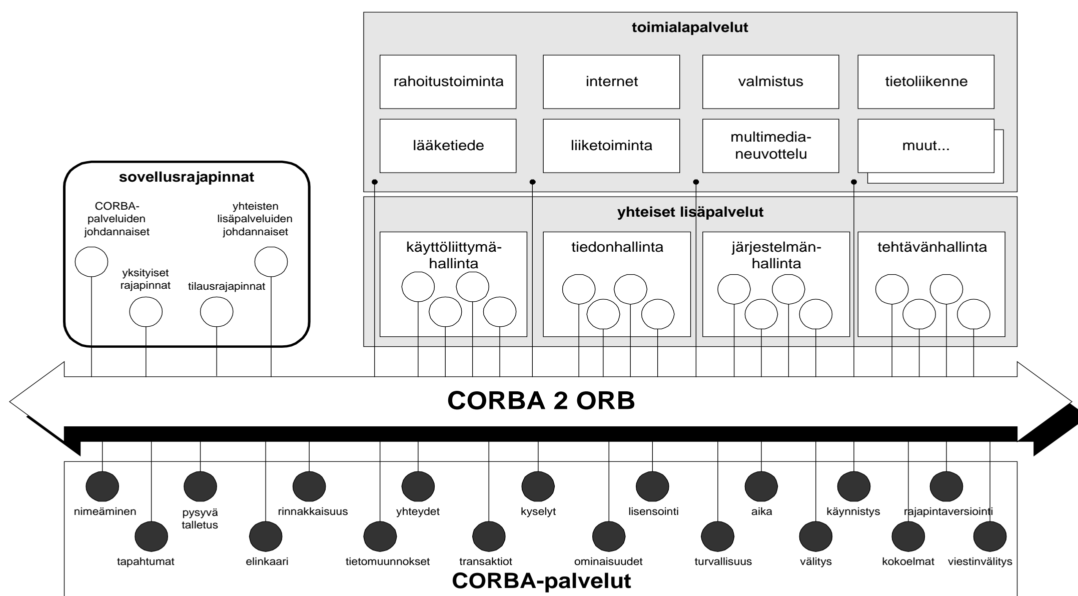
- **CORBA-palvelut (CORBA Services)** ovat palveluja, jotka tukevat perustoimintoja sovellusolioiden toteuttamista ja käyttöä varten. Palvelut ovat sovellusalasta riippumattomia ja niille on oma määritelmänsä *CORBAservices: Common Object Services Specification*.

- **Yhteiset lisäpalvelut (CORBA Facilities)** ovat palveluja, jotka ovat jaettavissa eri sovellusalojen kesken. Ne eivät ole yhtä olennaisia eivätkä yhtä matalan tason palveluja kuin oliopalvelut. Näille lisäpalveluille on oma määritelmänsä *CORBAfacilities: Common Facilities*.

- **Toimialapalvelut (CORBA Domains)** ovat sovellusalakohtaisia ylemmän tason palveluja. Merkittävimpiä sovellusaloja ovat tietoliikenne, valmistustekniikka ja lääketiede.

- **Sovellusrajapinnat (Application Interfaces)** ovat yksittäisen tuottajan kehitystyön tulosta. Tuottaja määrittelee sovellusolioiden rajapinnat ja tekee niille toteutukset. Sovellusoliot vastaavat perinteisiä sovelluksia, joten OMG ei ole standardoinut niitä. Sovellusoliot muodostavat CORBA-viitemallin ylimmän kerroksen ja voivat uudelleenkäyttää kaikkia muita viitemallin osia.

Kuva 8 esittää CORBA-viitemallia OMAn (Object Management Architecture) mukaisesti.



Kuva 8. Olionhallinta-arkkitehtuuri [7].

CORBA-palveluista suurin osa on määritelty täysin jo vuonna 1995, mutta joidenkin palvelujen (välitys-, kokoelma-, käynnistys-, rajapintaversiointi- ja viestinvälityspalvelujen) on arvioitu valmistuvan vuosina 1996 ja 1997 [7]. Maaliskuussa 1998 OMG:n www-sivulta[2] löytyivät määritelmät kaikille muille palveluille paitsi käynnistys-, rajapintaversionti- ja viestinvälityspalveluille. Joillekin palveluista on jo toteutuksia, mutta luultavasti kaikki ORB-tuottajat eivät tule toteuttamaan kaikkia CORBA-palveluja [7]. Tällöin vaihtoehtoina ovat palvelujen

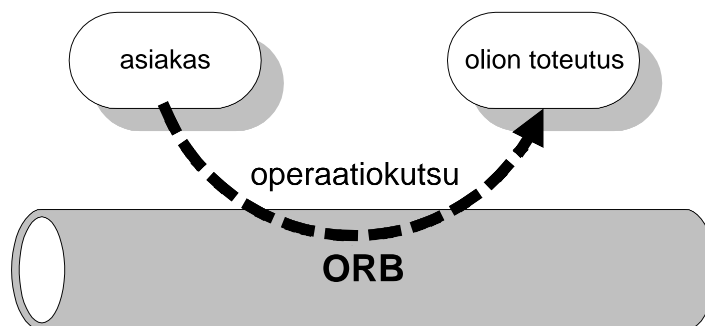
[2] CORBA-palveluiden määrittelyt löytyivät maaliskuussa 1998 osoitteesta *http://www.omg.org/corba/sectran1.htm.*

toteuttaminen itse tai tilaaminen joltain muulta osapuolelta kuin käytetyn ORB:n tuottajalta.

Yhteisten lisäpalvelujen ja erityisesti toimialapalvelujen määrittely ei ole yhtä pitkällä kuin CORBA-palvelujen määrittely. Ne ovat ylemmän tason komponentteja, jotka mahdollisesti käyttävät CORBA-palveluja. Niiden välinen raja ei ole tarkka, sillä niiden tarkoitus on lähinnä edistää suunnittelun uudelleenkäyttöä. [7]

3.5.1 Operaatiokutsut

Kuva 9 esittää asiakkaan lähettämää operaatiokutsua jonkin rajapinnan toteuttavalle oliolle. ORB on vastuussa mekanismeista, joita tarvitaan olion toteutuksen paikantamiseen, sen valmistelemiseen kutsun vastaanottoa varten ja pyynnön muodostavan tiedon käsittelemiseen. Asiakkaan näkemä olion rajapinta on riippumaton kohdeolion sijainnista, sen toteutuskielestä, ympäristöstä sekä kaikesta muusta, mikä ei käy ilmi rajapinnan IDL-määrittelyssä. [5]

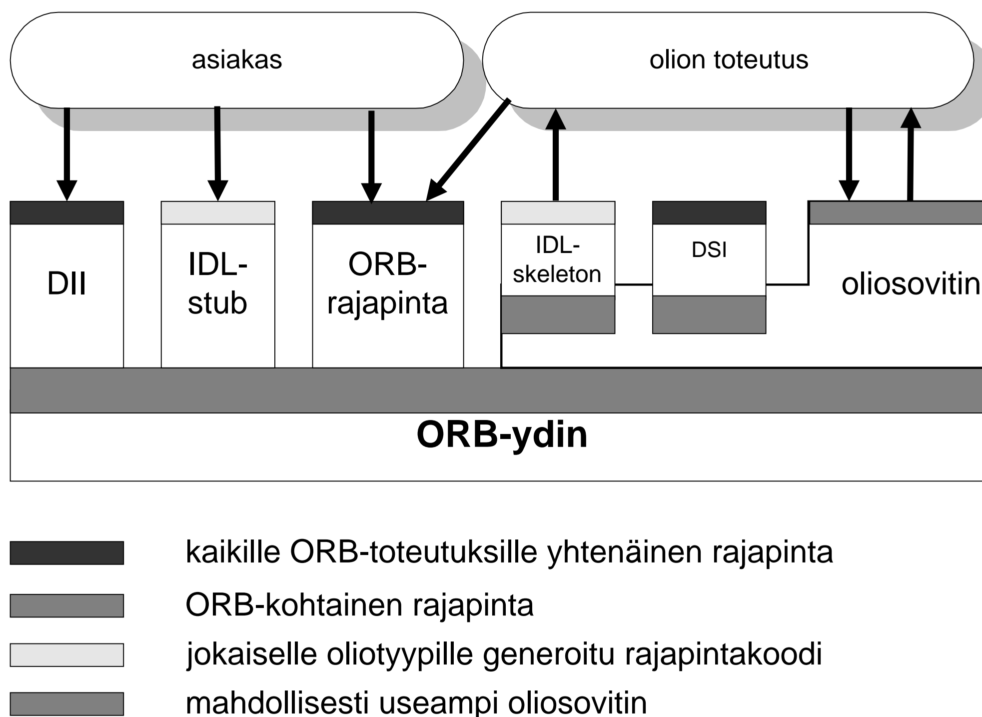


Kuva 9. ORB:n kautta lähetetty operaatiokutsu.

Sama kutsumekanismi on käyttökelpoinen riippumatta siitä, sijaitseeko kohdeolio samassa vai eri muistiavaruudessa tai koneessa kuin kutsun tekevä asiakas. Asiakas on kiinnostunut vain loogisesta operaatiosta ja sen mahdollisista vaikutuksista, ei toteutuksesta millään tasolla. [5]

3.5.2 ORB:n rakenne

Kuva 10 esittää ORB:n rakenteen. Se esittää alaspäin suuntautuvilla nuolilla sovellusten käytössä olevat ORB:n rajapinnat ja ylöspäin suuntautuvilla nuolilla ORB:n sovelluksiin kohdistamat toiminnot. [5]



Kuva 10. ORB:n rajapintojen rakenne.

Asiakasohjelma voi käyttää kutsun suorittamiseen dynaamista kutsurajapintaa (Dynamic Invocation Interface, DII) tai IDL-kääntäjän rajapintamääritelmästä luomaa staattista kutsurajapintaa (Static Invocation Interface, SII). DII on kutsumekanismi, jota käytettäessä asiakkaan ei tarvitse sisältää IDL-kääntäjän luomaa stub-koodia. DII-kutsu muodostetaan ohjelman suorituksen aikana määrittämällä kutsun kohdeolio, kutsuttava operaatio ja sen tarvitsemat parametrit. SII:tä käytetettäessä asiakasohjelma pitää kääntää stub-koodin kanssa. Asiakasohjelma voi käyttää myös ORB:n tarjoamia palveluja. [5]

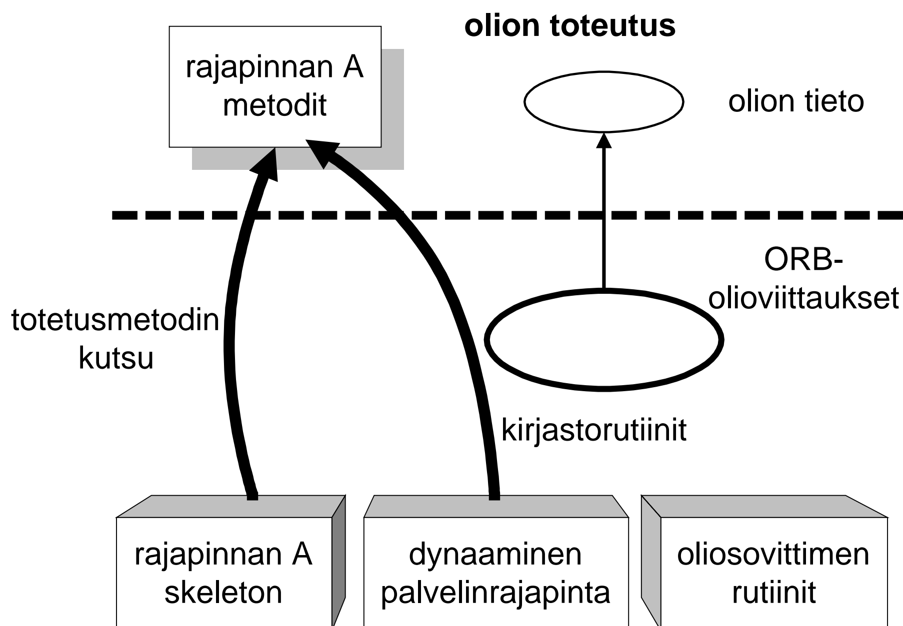
Olion toteutus vastaanottaa operaatiokutsun joko IDL-kääntäjän rajapintamääritelmästä luoman skeleton-koodin tai dynaamisen palvelinrajapinnan (Dynamic Skeleton Interface, DSI) kautta. DSI on yleinen rajapinta, sillä toteutusolion ei tarvitse käännösaikana tietää toteuttamansa rajapinnan tyyppiä. Olion toteutus voi käyttää oliosovittimen ja ORB:n tarjoamia palveluja. [5]

ORB:n rajapinta on suora yhteys ORB:n tarjoamiin palveluihin. Se on samanlainen kaikille ORB-toteutuksille ja riippumaton olion rajapinnasta ja oliosovittimesta. Koska suurin osa ORB:n toiminnallisuudesta tarjotaan sovelluksille oliosovittimen, stubien, skeletonien tai dynaamisen kutsurajapinnan kautta, vain muutama operaatio on yhteinen

kaikille oliotyypeille. Nämä operaatiot ovat sekä asiakkaan että oliototeutusten käytettävissä. [5]

Olion toteutus vastaa varsinaisen olion tilasta ja käyttäytymisestä. Sovellusalueen tarpeista riippuen olion totetus voidaan rakentaa eri tavoin. Olion toteutus voi pelkkien operaatioiden toteuttamisen lisäksi määritellä keinot olion aktivointiin ja deaktivointiin sekä käyttää muiden olioiden sekä teknologioiden (tietokannat, kommunikointimekanismit tms.) tarjoamia mahdollisuuksia. Olion toteutus käyttää ORB:n palveluja suoraan tai oliosovittimen kautta. [5]

Kun asiakas suorittaa operaatiokutsun olioviittauksen avulla, ORB-ydin, oliosovitin ja skeleton-koodi vastaavat kutsun välittämisestä varsinaiselle operaation toteuttavalle kohdeolion metodille. Metodin parametri määrittää kutsuttavan toteutusluokan instanssin eli olion, jonka avulla metodi paikantaa kyseisen olion tiedon. Operaatiokutsun yhteydessä tulevat parametrit välitetään toteutusmetodille, ja metodin suorituksen jälkeen palautusparametrit ja palautusarvo siirtyvät takaisin asiakkaalle. Kuva 11 esittää tätä mekanismia ja olion toteutuksen rakennetta. [5]



Kuva 11. Tyypillisen oliototeutuksen rakenne.

3.5.3 ORB:n kutsutyypit

Asiakas voi suorittaa operaatiokutsun DII:n tai SII:n avulla. SII:n käyttö edellyttää staattista stub-koodin mukaanottamista asiakkaan käännökseen, mutta tarjoaa

vastineeksi vahvan käännösaikaisen tyypintarkistuksen. DII:n avulla riippuvuus stub-koodista voidaan välttää ja kutsun suoritus saada joustavammaksi, mutta DII on hankalampi käyttää ja siirtää virheiden havaitsemisen tai vaikutukset ajonaikaiseksi.

Oletuksena IDL-rajapintakuvauksessa määritellyt operaatiot ovat synkronisia: operaation kutsuja odottaa, kunnes operaation vastaanottaja on käsitellyt kutsun ja mahdolliset palautusarvot on vastaanotettu. Tarvittaessa operaatio voidaan määritellä IDL-tasolla yksisuuntaiseksi *(oneway)*, jolloin operaation kutsuja ei jää odottamaan palvelimen kutsun käsittelyä vaan jatkaa toimintaansa välittömästi kutsun lähettämisen jälkeen. CORBA määrittelee yksisuuntaisen kutsun operaatioksi, jolle käytetään *best-effort*-semantiikkaa: onnistunut kutsu suoritetaan tarkalleen kerran, mutta toisaalta virhetilanteessa korkeintaan kerran. Yksisuuntaiselle kutsulle ei voi antaa out- tai inout-parametreja, se ei voi aiheuttaa käyttäjän määrittelemää poikkeusta eikä sillä voi olla palautusarvoa. [5]

Takeita yksisuuntaisen kutsun onnistumisesta ei siis ole. CORBA jättää ORB-toteutuksen vastuulle yksisuuntaisten kutsujen toteuttamisen. Tämän vuoksi sovellukset eivät voi tietää tarkalleen, mitä yksisuuntainen kutsu missäkin ORB-toteutuksessa takaa. Siksi niiden turhaa käyttöä tulisi välttää. Jotkut ORB:t suorittavat yksisuuntaiset kutsut luotettavasti välittämällä ne samaa yhteysperustaista protokollaa käyttäen kuin normaalit operaatiokutsut. Yksittäinen yksisuuntainen kutsu voidaan suorittaa luotettavasti, mutta suuri määrä nopeasti lähetettyjä yksisuuntaisia kutsuja voi ylikuormittaa palvelimen ja pakottaa kutsujan odottamaan palvelimen palautumista. [8, s. 48 - 49]

CORBA määrittelee lisäksi DII:n yhteydessä käytettävät viivästetyt synkroniset (deferred synchronous) kutsut. Kutsun lähettämiseen käytetään funktiota *send()*, jolloin kutsujan toiminta voi jatkua heti kutsun lähettämisen jälkeen. Toinen tähän ryhmään kuuluva funktio on *send_multiple_requests()*, joka suorittaa useamman operaatiokutsun rinnakkain. Rinnakkaisuuden aste on järjestelmäriippuvainen eikä operaatioiden suoritusjärjestyksestä ole takeita. Operaation valmistumista voi myöhemmin tarkastella kiertokyselytyyliin (polling) funktioilla *get_response()* tai *get_next_response()*. [5]

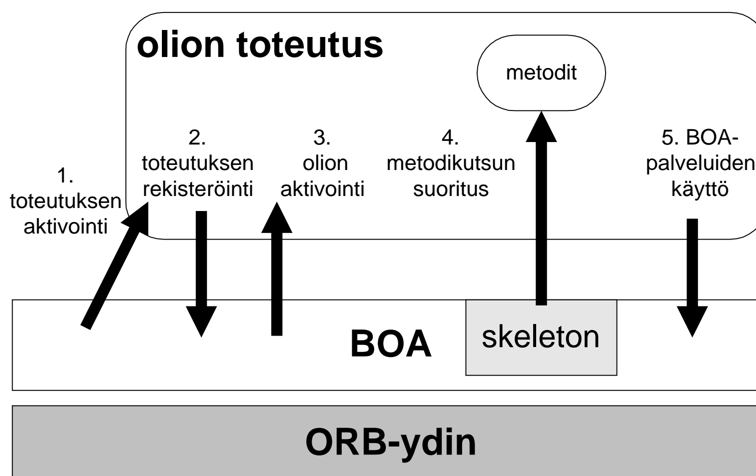
3.5.4 Oliosovitin

Olion toteutus käyttää ORB:n palveluja pääasiassa oliosovittimen avulla. Oliosovittimen tehtäviä ovat olioviittausten luominen ja tulkinta, metodikutsujen suorittaminen, kutsujen turvallisuuden varmistaminen, olion ja palvelimen aktivointi, deaktivointi ja rekisteröinti sekä olioviittausten yhdistäminen olioiden toteutuksiin. Oliosovitin suorittaa nämä toiminnot käyttämällä ORB-ydintä tai muiden lisäkomponenttien palveluja. Olioiden erilaisten toiminnallisuuksien ja erikoisvaatimusten vuoksi ORB-

ytimen on vaikea tarjota yhtä ainoaa rajapintaa, joka soveltuu ja on lisäksi tehokas eri tavoilla toteutetuille olioille. Siksi ORB voi toteuttaa useamman, erityyppisille kohdeolioille tarkoitetun oliosovittimen. Käytetty oliosovitin on asiakkaan kannalta läpinäkyvä. [5]

CORBA-spesifikaatio määrittelee pakollisen *perusoliosovittimen*. (Basic Object Adapter, BOA). BOA on rajapinta, joka on laajasti käytettävissä ja joka tukee suurta joukkoa tyypillisiä oliototeutuksia. BOAn tehtävät ovat samat kuin yleisen oliosovittimen, mutta lisänä on operaatiokutsun suorittajan tunnistaminen. Vaikka BOAlla on yleensä ORB-riippuvainen toteutus, sitä käyttävien olioiden pitäisi toimia kaikissa tiettyä kielisidontaa tukevissa ORB-toteutuksissa. [5]

Kuva 12 esittää BOAn rakennetta ja sen vuorovaikutusta palvelimen kanssa. BOA käynnistää palvelimen toteuttavan ohjelman (1), jonka jälkeen palvelin kertoo BOAlle suorittaneensa alustusrutiinit ja olevansa valmis vastaanottamaan operaatiokutsuja (2). Kun ensimmäinen operaatiokutsu jollekin kohdeoliolle vastaanotetaan, BOA pyytää palvelinta aktivoimaan kohdeolion (3). Seuraavilla operaatiokutsuilla BOA kutsuu toteutusolion metodeja käyttämällä rajapintakohtaista skeleton-koodia (4). Palvelin voi tarvittaessa käyttää BOAn palveluja (5). [5]



Kuva 12. BOAn rakenne ja toiminta.

3.5.5 Rajapintavarasto

Rajapintavarasto (Interface Repository, IFR) on palvelu, joka ylläpitää ja jakaa pysyvästi talletettua tietoa järjestelmän IDL-moduuleista, rajapintakuvauksista ja muista IDL-tyypeistä ohjelmien suorituksen aikana [8]. ORB voi käyttää IFRn palveluja operaatiokutsujen suorittamiseen. Selvittämällä rajapinnan operaatiot ja niiden vaatimat

parametrit IFRn avulla sovellus voi käyttää olioita, joiden rajapinnat eivät olleet tiedossa sovelluksen kääntämisen aikana. IFRn pääasialliset käyttötarpeet ovatkin operaatiokutsujen suorittaminen DIIn avulla ja ORB-toteutusten yhteenliittämisessä tarvittava rajapintaominaisuuksien kysely. Lisäksi suunnittelijoiden ja ohjelmoijien käytössä olevat selaimet, CASE-työkalut (Computer Aided Software Engineering) ja yhdyskäytävät voivat hyödyntää IFR:n tarjoamaa tietoa [8].

3.5.6 Toteutusvarasto

Koska BOA kommunikoi oliototeutusten kanssa ja aktivoi niitä käyttäen ORBn ulkopuolelle luokiteltavia käyttöjärjestelmän mekanismeja, se tarvitsee järjestelmäkohtaista tietoa. Tämän järjestelmäkohtaisen tiedon ylläpitämiseen BOA määrittelee käsitteen *toteutusvarasto* (Implementation Repository, IMR). Koska sidontamekanismi ohjelman ja BOAn sekä ORBn välillä on luonnostaan järjestelmästä ja ohjelmointikielestä riippuva, CORBA ei määrittele IMRn toteutusta. [5]

ORB käyttää IMR:n ylläpitämää tietoa olioiden toteutusten paikantamiseen ja aktivoimiseen. Tyypillisesti myös olioiden toteutusten asennus ja niiden aktivointitapojen hallinta tehdään IMRn avulla. Lisäksi sitä voidaan käyttää kaiken muun toteutuksiin liittyvän lisätiedon, kuten ohjelmien virheidenkorjaukseen liittyvän tiedon, järjestelmän hallintatiedon ja resurssien kohdentamistiedon, hallintaan. [5]

3.5.7 Toteutusten rekisteröinti

BOA olettaa palvelinten ja olioiden toteutusten aktivointiin liittyvän kuvaustiedon löytyvän IMRstä. Operaatiokutsun seurauksena BOA voi suorittaa palvelimen tai olion toteutuksen aktivoinnin, jolloin BOA aktivoi toteutukset niiden aktivointitavan perusteella. Kaikkien BOA-toteutusten täytyy tukea seuraavia neljää aktivointitapaa:

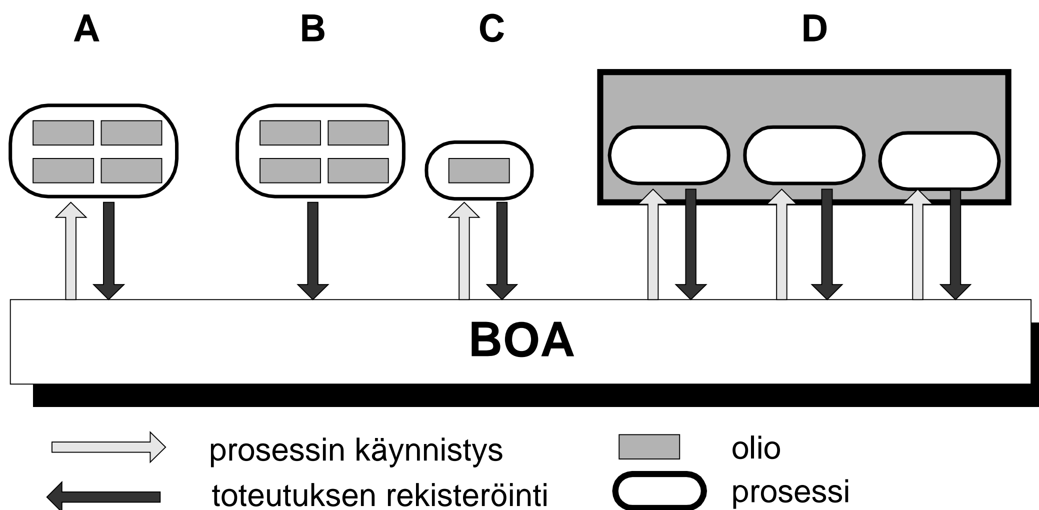
- **Jaettu palvelin (shared server)**, missä useampi olio jakaa saman palvelimen. Palvelin aktivoidaan, kun mihin tahansa sen ylläpitämistä olioista kohdistuu operaatiokutsu. Tämä on yleisimmin käytetty aktivointimoodi.

- **Jakamaton palvelin (unshared server)**, missä vain yksi olio kerrallaan voi olla aktiivinen palvelimessa. Jokaiselle oliolle luodaan oma palvelinprosessi, mikä voi olla tarkoituksenmukaista esimerkiksi, kun olio edustaa kokonaista sovellusta tai kun palvelin tarvitsee toiset poissulkevan pääsyn jaettuun resurssiin.

- **Palvelin-per-metodi (server-per-method)**, missä jokainen metodikutsu aiheuttaa uuden palvelimen käynnistämisen. Useampi palvelin voi olla aktiivinen samanaikaisesti samalle oliolle tai samalle metodille. Tämä on käyttökelpoinen mekanismi esimerkiksi silloin, kun metodin kutsuminen on harvinaista ja metodin tarvitsemat resurssit on vapautettava heti kutsun suorittamisen jälkeen.

- **Pysyvä palvelin (persistent server)**, missä palvelin aktivoidaan BOA:n ulkopuolelta, esimerkiksi käyttäjän komennosta. Kun pysyvä palvelin on rekisteröitynyt BOAlle, sitä käsitellään kuten jaettua palvelinta.

Kuva 13 esittää näitä toteutuksen aktivointitapoja. A on BOAn käynnistämä jaettu palvelin, B on pysyvä palvelin, C on jakamaton palvelin ja D on palvelin-per-metodi-tyyppinen palvelin. [5]



Kuva 13. Toteutuksen aktivointitavat.

3.5.8 Dynaaminen kutsurajapinta

Staattisen kutsumekanismin lisäksi CORBA tarjoaa dynaamisen kutsumekanismin asiakkaiden käyttöön. DII tarjoaa yleisen rajapinnan, jonka avulla mille tahansa oliolle voi lähettää minkä tahansa operaatiokutsun. Asiakas ei tarvitse IDL-kääntäjän kohderajapinnasta luomaa stub-koodia operaatiokutsun lähettämiseen, vaan se voi käyttää DII:n avulla käännöshetkellä tuntemattomien rajapintojen operaatioita. [8]

DII tarjoaa ylimääräisiä kutsutyyppejä asynkroniseen kommunikointiin (kohta 3.5.3) ja mahdollisuuden joustaviin kutsuihin, jotka voidaan määritellä ohjelman suorituksen aikana. Parametrien syöttäminen DII:llä tehtävään kutsuun on kuitenkin kömpelöä, jos

pitää varautua erilaisten parametrien antamiseen operaatiolle. Operaation vaatimat parametrit ja niiden tyypit pitää ottaa selville esimerkiksi IFR:stä ja sijoittaminen kutsuun edellyttää jonkinlaista silmukan ja *switch-case*-rakenteen yhdistelmää. Mikäli käytetään parametrittomia tai parametreiltaan samantyyppisiä operaatioita, DII:n käyttö on joustavaa. Kohdassa 4.1.7 tarkastellaan DII:tä Orbixin toteuttamana.

3.5.9 Yhteensopivuus

CORBA-spesifikaation yhteensopivuusosuus (interoperability) määrittelee mekanismit, joilla heterogeeniset CORBA-yhteensopivat ORB-toteutukset saadaan yhdistettyä toimivaksi kokonaisuudeksi. Tavoitteena on piilottaa erilaisten toimialueiden rajat: keskenään kommunikoivien ORB-toteutusten ei tarvitse tietää mitään yksityiskohtia toisena osapuolena olevasta ORB-toteutuksesta tai sen toimintaympäristöstä. Määritelmän mukaiset yhteensopivuuden elementit ovat [5]

- ORB-yhteensopivuusarkkitehtuuri
- ORB yhdyssiltojen[3] tukeminen
- yleinen ORB-yhdysprotokolla (General Inter-ORB Protocol, GIOP) ja IIOP.

ORB-yhteensopivuusarkkitehtuuri tarjoaa käsitteellisen viitekehyksen yhteensopivuuden elementtien määrittelylle ja yhteensopivuuden kannalta tärkeiden pisteiden tunnistamiseen. Se kuvaa myös mekanismit ja noudatettavat säännöt, joita yhteensopivuus eri ORB-tuotteiden välillä edellyttää. Lisäksi se määrittelee ORB-alueiden välittömän ja välillisen siltauksen. Välittömässä siltauksessa kommunikoinnin kannalta olennaiset elementit muunnetaan toimialueiden[4] rajalla sisäisestä esitysmuodosta toiseen. Välillisessä siltauksessa muunnos toimialueiden sisäisten esitysmuotojen välillä tapahtuu yleisen esitysmuodon kautta. Tämä yleinen esitysmuoto voi olla esimerkiksi kahden ORB:n keskenään sopima tai maailmanlaajuisesti hyväksytty. Yleisiä esitysmuotoja voi olla useaa eri tyyppiä, jotka ovat käyttötarkoituksensa mukaan muotoiltu ja optimoitu. [5]

Kun kaksi ORB-toteutusta ovat samalla toimialueella, ne voivat kommunikoida suoraan. Jos operaatiokutsun sisältämä tieto joutuu vaihtamaan toimialuetta, kutsun täytyy kulkea sillan kautta. Sillan tehtävä on varmistaa, että operaatiokutsun sisältö ja semantiikka

---

[3] Termi “silta” on tässä käsitteellinen ja viittaa vain eri esitystapojen välillä tehtävän muunnokseen.
[4] Tässä arkkitehtuurissa toimialue *(domain)* tarkoittaa erillistä vaikutusaluetta, jonka sisällä tietyt ominaisuudet ja yhteiset säännöt ovat voimassa.

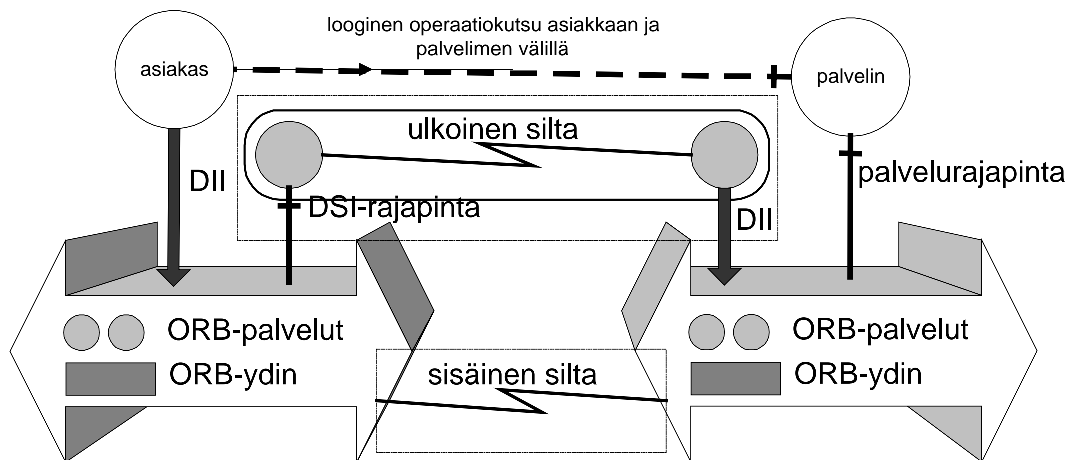
muunnetaan lähettävän ORB:n esitysmuodosta vastaanottavan ORB:n esitysmuotoon. Tällä tavoin minkä tahansa ORB:n käyttäjä näkee vain oman esitystapansa mukaisen sisällön ja semantiikan. [5]

ORB-yhdyssiltojen tukea voidaan soveltaa myös yhteistoimintaan muiden kuin CORBA-järjestelmien, esimerkiksi Microsoftin Component Object Model (COM) -teknologian, kanssa.

GIOP määrittelee standardin siirtomuodon (alemman tason tiedonesitystavan) ja joukon viestien esitystapoja kommunikointiin eri ORB-tuotteiden välillä. GIOP on suunniteltu erityisesti usean eri ORBn väliseen vuorovaikutukseen ja toimimaan minkä tahansa yhteysperustaisen siirtoprotokollan päällä. GIOPn suunnittelu mahdollistaa siirrettävien ja suorituskykyisten toteutusten rakentamisen, jotka ovat mahdollisimman vähän riippuvaisia muista tukiohjelmistoista kuin alla olevasta siirtokerroksesta. [5]

IIOP on TCP/IP:n päällä toimiva GIOP. Se on pakollinen kaikille ORB-toteutuksille. IIOPn suhde GIOPhen on sama kuin yksittäisen kielisidonnan suhde IDL-kuvauskieleen: GIOP voidaan toteuttaa monelle erilaiselle siirtokerrokselle, aivan kuten IDL voidaan kääntää useammalle eri toteutuskielelle. [5]

Yhteensopivuuden takaavat sillat voidaan toteuttaa joko ORB:n sisäisillä mekanismeilla tai ORB-ytimen yläpuolella toimivalla kerroksella. Kuva 14 esittää näitä kahta siltauksen toteutustapaa. Sisäiset sillat (in-line bridges) rakennetaan ORB:n sisäisellä sovellusrajapinnalla. Se on siltauksen suorin muoto, mutta edellyttää merkittäviä muutoksia ORB:n toteutukseen. Ulkoiset sillat (request-level bridges) käyttävät CORBAn määrittelemää sovellusrajapintaa ja dynaamista palvelinrajapintaa operaatiokutsujen lähettämiseen ja vastaanottamiseen. Ulkoisilla silloilla totetutettu siltausmekanismi edellyttää kummassakin ORB:ssa ylimääräisen komponentin, “puoli-sillan”, toteuttamista. [5]



Kuva 14. Kahden ORBn välinen siltaus.

3.6 CORBA-palvelut

CORBA-standardin mukainen ORB-toteutus tarjoaa mekanismit sovellusolioiden hajauttamiselle, muttei sellaisenaan tarjoa riittävästi tukea tietyntyyppisten järjestelmien toteuttamiseen. Eri sovellusalojen korkeamman tason palvelujen tarpeiden vuoksi OMG on määritellyt joukon lisäpalveluja. CORBA-palvelujen toteutuksia ei ole tarkoitettu loppukäyttäjille, vaan ne auttavat ohjelmiston suunnittelijoita ja ohjelmoijia tarjoamalla ratkaisuja yleisiin, sovellusriippumattomiin ongelmiin [8, s. 375]. Esimerkiksi tietoliikenne, valmistusjärjestelmät ja rahoitus ovat sovellusaloja, joilla tarvitaan sovellusalakohtaisesti toteutettuja lisäpalveluja [8, s. 375].

CORBA-palvelut on pelkkä määritelmä lisäpalveluille. Palvelujen toteutus jätetään ORB-tuottajan tai muun osapuolen vastuulle. Lisäpalvelut ovat tyypillisesti jääneet ORB-tuottajilla vähemmälle huomiolle ORB-ytimen vaatiman jatkokehityksen ja osittain kesken olevan palvelumäärittelyn vuoksi. Monet ohjelmistonkehittäjät rakentavat sovelluksensa huomioimatta CORBA-palveluja ja päätyvät tilanteeseen, jossa ovat toteuttaneet CORBA-palvelujen perustoimintoja [7]. Tämä menettelytapa tulee usein kalliimmaksi kuin ostaa valmis toteutus palvelulle. Kuva 8 sivulla 27 esittää CORBA-palvelut.

Seuraavassa kuvataan lyhyesti tärkeimpien CORBA-palvelujen tehtävät. Tarkemmat selostukset palvelujen merkityksestä, toiminnasta ja nykytilasta löytyvät CORBA-palvelujen määrittelyistä (*http://www.omg.org/corba/csindx.htm*) tai alan kirjallisuudesta [7, 8, 10].

- **Nimipalvelun (Naming Service)** avulla sovellukset hankkivat olioviittauksia olioiden loogisten nimien perusteella. Nimipalvelulla on rajapinta, jota sovellukset

voivat käyttää olioiden nimihierarkioiden luomiseen, navigoimiseen ja olioviittausten saamiseen. [10]

- **Tapahtumapalvelu (Event Service)** tarjoaa asynkronisen kommunikoinnin anonyymien olioiden välillä. Kommunikointi perustuu olioiden luomiin tapahtumiin, jotka välittyvät tapahtumista kiinnostuneille osapuolille tapahtumapalvelun kautta. Tapahtumapalvelun avulla oliot voivat dynaamisesti rekisteröidä kiinnostuksensa tai sen loppumisen tietyntyyppisiin tapahtumiin. Sovellukset pysyvät löysästi kytkettyinä: tapahtuman luojan ei tarvitse tietää, mitkä osapuolet ovat kiinnostuneita sen luomasta tapahtumasta. *Push*-moodissa tapahtuman tuottaja alustaa tapahtumatiedon siirron, ja *pull*-moodissa sen tekevät tapahtumista kiinnostuneet osapuolet. Tapahtumapalvelun toteutus voi halutessaan puskuroida ja tallettaa tapahtumia myöhempää käyttöä varten ja voi mahdollistaa myös tapahtuman tuottajan suoran kommunikoinnin tapahtumista kiinnostuneiden osapuolten kanssa. [10]

- **Pysyvyyspalvelu (Persistent Object Service, POS)** mahdollistaa CORBA-olioiden tilan tallentamisen pysyvästi. POS määrittelee pysyvyyspalvelun loogisen arkkitehtuurin, minkä puitteissa palvelu on toteutettava. Se jättää avoimeksi sen, miten tallettaminen hoidetaan, joten esimerkiksi relaatiotietokannat, oliotietokannat ja tiedostojärjestelmät ovat mahdollisia fyysisiä talletustapoja. Joko olion toteutus tai asiakas voi päättää, milloin olion tila talletetaan pysyvästi tai luetaan pysyvästä varastosta. [8]

- **Rinnakkaisuuspalvelun (Concurrency Service)** avulla oliot voivat koordinoida pääsynsä jaettuihin resursseihin lukkoja käyttämällä. Palvelu määrittelee useita lukitustapoja eri pääsytapojen ja lukituksen hienojakoisuuden (granularity) mukaan. Lukitusta voidaan tarvita sekä rinnakkaisille säikeille että transaktioille. Rinnakkaisuuspalvelun rajapinnat eivät tyypillisesti (määritelmä ei kuitenkaan vaadi sitä) ole asiakkaille näkyviä toteutuksen eristämiseksi, vaan operaation toteutus hoitaa lukitusten käsittelyn. Jos haluttu resurssi on jo lukittu, operaation totetutus jää odottamaan lukkojen vapautumista. Rinnakkaisten säikeiden tapauksessa lukkoa pitävä olio vapauttaa lukon, kun se ei sitä enää tarvitse, mutta transaktioiden tapauksessa lukot vapautetaan automaattisesti transaktion päättyessä. Rinnakkaisuuspalvelua voidaan käyttää yhdessä transaktiopalvelun kanssa synkronoimalla näiden kahden palvelun toiminta. [8]

- **Transaktiopalvelun (Object Transaction Service, OTS)** tarjoaman rajapinnan avulla asiakkaat voivat suorittaa olioiden toteutuksiin kohdistuvia atomisia transaktioita. Asiakas voi aloittaa transaktion, käsitellä joukkoa olioita ja päättää, haluaako se hyväksyä vai peruuttaa transaktion. OTS:n toteutus voi halutessaan tukea

myös sisäkkäisiä transaktioita. OTS mahdollistaa rinnakkaisuuden hallinnan, tiedon yhtenäisyyden säilymisen ja virhetilanteista palautumisen CORBA-järjestelmissä. Transaktiot toimivat *kaikki-tai-ei-mitään*-periaatteella: jos yksikin transaktioon kuuluvista operaatioista epäonnistuu, kaikkien muidenkin operaatioiden vaikutukset peruutetaan. OTS käyttää *kaksivaihehyväksyntää* (two-phase commit) tapahtumien hyväksymiseen. [8]

- **Turvallisuuspalvelu (Security Service)** varmistaa, että tietoa voivat käsitellä vain siihen oikeutetut osapuolet ja vain oikeuksiensa määrittämällä tavalla. Näiden vaatimusten täyttämiseksi turvallisuuspalvelu voi tunnistaa operaatiokutsujen suorittajat, rajoittaa palvelinolioiden kutsujen suorittamista, välittää kutsun suorittajan oikeuksia eteenpäin, ylläpitää lokia tapahtumista, todistaa toimintojen suorittajan ja suojata verkossa välitettävät viestit. [8]

4. ORB-TOTEUTUKSET

Tässä luvussa käsitellään tarkemmin kahta käytännössä kokeiltua CORBAn mukaista toteutusta, Orbixia ja Chorus/COOL ORB:ta. Muihin ORB-toteutuksiin annetaan viitteitä, joista lisätietoa on hankittavissa.

4.1 Orbix

Orbix on IONA Technologies -yhtymän tuottama, ehkä yleisimmin käytetty CORBA-standardin mukainen ORB-toteutus. Se tarjoaa ohjelmoijalle paljon eri mahdollisuuksia ja omilla lisäominaisuuksillaan täydentää CORBA-spesifikaation ORB:ltä vaatimia piirteitä.

Tässä kohdassa esitellään Orbix-ohjelmiston rakennetta, ominaisuuksia ja toimintaa. Tiedot on saatu pääosin Orbixin manuaaleista [11, 12], muilta osin viittaukset lähdeteoksiin esitetään tekstissä.

4.1.1 Rakenne

Orbix-ohjelmisto koostuu Orbix-daemon -prosessista *(orbixd)*, asiakas- ja palvelinkirjastoista, rajapintavarastosta, IDL-kääntäjästä ja useammasta työkalusta. Suurin osa toiminnallisuudesta on piilotettu sovellusten käyttämiin kirjastoihin.

- **Orbix-daemon** tarvitaan jokaisessa palvelimia ylläpitävässä koneessa. Orbix-daemon odottaa tulevia operaatiokutsuja, aktivoi palvelimen, ellei se jo ole aktiivinen, ja muodostaa yhteyden asiakkaan ja palvelimen välille. Se ei ota osaa myöhempään asiakkaan ja palvelimen väliseen kommunikointiin. Orbix-daemon tarvitaan myös asiakaskoneissa, jotka käyttävät Orbixin oletuspaikantimen palveluja. Orbix-daemon on Orbixin toteutus IMR:lle, mutta se hoitaa myös osaa BOAn tehtävistä.

- **Rajapintavarasto** on Orbix-sovellus, josta sovellukset voivat hakea tietoa järjestelmän olioiden rajapintaominaisuuksista ohjelman ajon aikana.

- **IDL-kääntäjä** muuntaa IDL-rajapintamääritelmien operaatiot, attribuutit ja muut tietotyypit C++-kääntäjän ymmärtämään muotoon.

- **Työkalut** ovat lähinnä järjestelmän konfigurointitiedon hallintaan tehtyjä sovelluksia. Niillä voidaan esimerkiksi rekisteröidä palvelimia, määritellä ympäristömuuttujia ja seurata järjestelmän tilaa.

4.1.2 Ohjelmiston kehitys

Asiakas-palvelinsovelluksen tekeminen Orbixilla koostuu seuraavista vaiheista:

1. Määritellään olion rajapinta IDL-kielellä.

2. Käännetään IDL-rajapinta IDL-kääntäjällä C++-kieliseksi asiakkaan ja palvelimen käännökseen mukaan otettavaksi stub- ja skeleton-koodiksi.

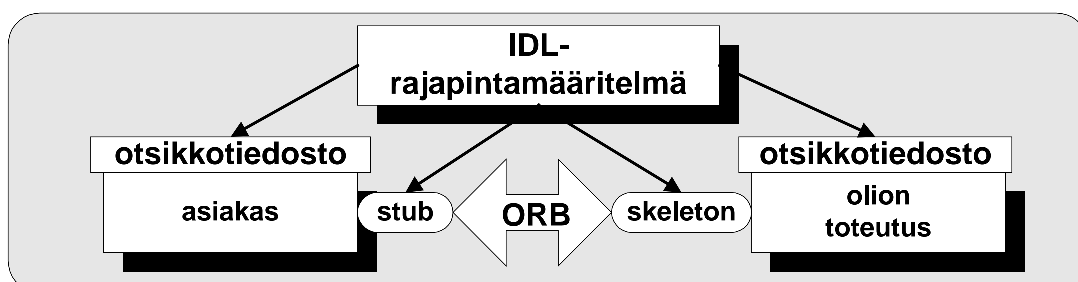
3. Toteutetaan rajapinnan toiminnallisuuden tarjoava olio C++-kielellä.

4. Toteutetaan olion ylläpitävä palvelin C++-kielellä.

5. Rekisteröidään palvelin *putit*-työkalulla. Rekisteröinti on tarpeen, mikäli palvelin halutaan laukaista automaattisesti operaatiokutsun yhteydessä.

6. Toteutetaan asiakas C++-kielellä.

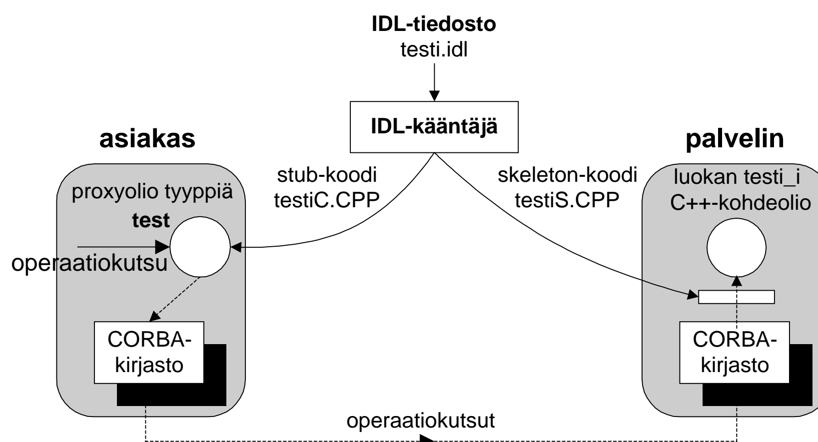
IDL-kääntäjä luo rajapintamääritelmästä yhteisen otsikkotiedoston asiakas- ja palvelinsovellukselle. Lisäksi asiakkaalle ja palvelimelle luodaan omat lähdekooditiedostot, joissa on ohjelmoijalle läpinäkyvä toteutus olioiden hajautukseen. Kuva 15 esittää IDL-kääntäjän luomien tiedostojen asemaa sovelluksessa.



Kuva 15. IDL-kääntäjän luoma koodi.

Orbixilla IDL-kääntäjän luoma skeleton-koodi koodi sisältää myös asiakkaalle luodun stub-koodin. Asiakkaaseen mukaanotettavaa stub-koodia nimitetään myös proxy-koodiksi, koska sen avulla asiakkaan muistiavaruuteen luodaan kohdeolion paikallinen edustaja, proxy, jonka kautta kaikki asiakkaan tekemät operaatiokutsut välitetään kohde-

oliolle. Kuva 16 esittää operaatiokutsujen välittämistä asiakkaan muistiavaruuteen luodun proxy-olion avulla.



Kuva 16. Operaatiokutsut proxyn kautta.

4.1.3 Olioiden toteutus

Orbix tarjoaa kaksi CORBAn määrittelemää tapaa CORBA-olioiden toteuttamiseen: BOA- ja TIE-mekanismit. Näillä mekanismeilla sidotaan toteutusluokka yhteen tai useampaan rajapintaan. BOA:n tapauksessa toteutusluokka perii IDL-kääntäjän rajanpinnasta generoiman C++-luokan *rajapintaBOAImpl*. TIEn tapauksessa käytetään makroa *DEF_TIE_rajapinta(toteutusluokka)*, joka luo ylimääräisen luokkamääritelmän rajapinnan ja toteutusluokan liittämiseksi yhteen. Tämä luokka sisältää osoitinkentän, jonka avulla operaatiokutsut välitetään toteutusoliolle.[8]

Olemassa olevan koodin toteuttaminen CORBA-pakkaaja-luokkien (wrapper) avulla on joustavampaa BOA-mekanismilla, mutta useamman rajapinnan toteuttaminen yhdellä C++-luokalla on helpompaa TIE-mekanismia käyttäen. TIE-mekanismi mahdollistaa oliokohtaisen suodatuksen, mutta vaikeuttaa muistinhallintaa luomalla yhden olion enemmän muistiin. Tätä oliota ei tuhota automaattisesti ohjelmoijan suorittaman toteutusolion tuhoamisen yhteydessä. Kummallakin mekanismilla on hyvät puolensa, mutta valinta näiden välillä on lähinnä makuasia [8].

4.1.4 Palvelinten rekisteröinti

Jos halutaan, että palvelin on käynnistettävissä (ladattavissa levyltä muistiin) automaattisesti asiakkaan kutsun yhteydessä, se täytyy rekisteröidä Orbixille *putit*-työkalulla. Kun palvelin rekisteröidään, Orbix luo levylle tiedoston, johon se tallettaa

tiedon rekisteröitävästä palvelimesta. Tiedosto määrittelee esimerkiksi palvelimen aktivointimoodin ja omistajan sekä tiedon siitä, kuka on oikeutettu käyttämään palvelinta. Mikäli palvelimen käynnistäminen vasta operaatiokutsun yhteydessä on sovelluksen kannalta liian hidas toimenpide, palvelimet voidaan käynnistää komentoriviltä. Palvelimet voidaan rekisteröidä jollakin CORBAn määrittelemällä aktivointitavalla (kohta 3.5.7). Orbix toteuttaa CORBAn määrittelemän pysyvän palvelimen samanlaisena jaetun aktivointimoodin palvelimen kanssa. Erona on se, että pysyvää palvelinta ei rekisteröidä *putit*-työkalulla, eikä se siten ole käynnistettävissä automaattisesti. Orbix tarjoaa jaetulle ja jakamattomalle palvelimelle vielä seuraavat kolme lisämoodia:

1) *Asiakaskohtainen;* jokaiselle loppukäyttäjälle luodaan oma palvelinprosessi käyttäjätunnuksen mukaan.

2) *Asiakasprosessikohtainen;* jokaiselle asiakasprosessille luodaan oma palvelinprosessi prosessin tunnisteen mukaan.

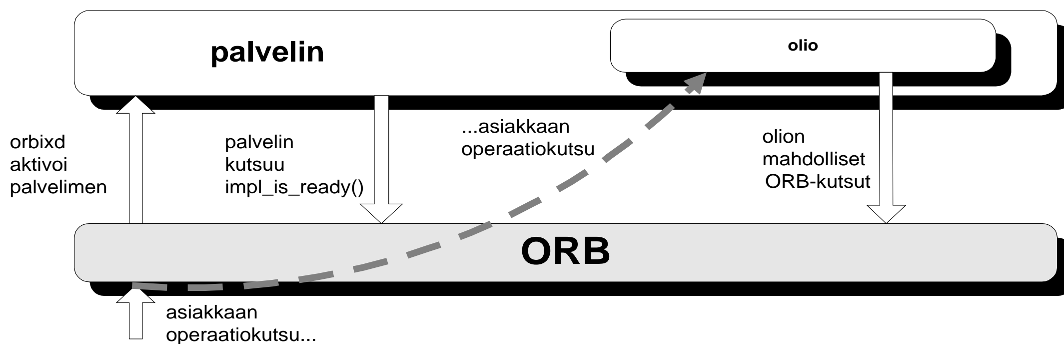
3) *Monen asiakkaan moodi (oletus);* eri loppukäyttäjät jakavat saman palvelinprosessin.

4.1.5 Operaatiokutsujen käsittely

Oletuksena jokaisella palvelinprosessilla on yksi säie, joka käsittelee tulevat palvelupyynnöt. Orbix takaa, että vain yksi pyyntö on aktiivinen prosessissa kerrallaan puskuroimalla tulevat operaatiokutsut odottamaan palvelimen vapautumista. Tarvittaessa jokaiselle operaatiokutsulle voidaan luoda oma säikeensä.

Orbix-daemon käyttää yhden TCP/IP-portin. Jokaiselle aktivoidulle palvelimelle varataan myös TCP/IP-portti. Kun asiakas ottaa yhteyden palvelimeen, sille palautetaan osoitin IDL-kääntäjän luoman C++-luokan instanssiin. Jos kohdeolio on eri muistiavaruudessa, osoitin viittaa hajautettua kohdeoliota edustavaan proxy-olioon.

Palvelimen aktivoituessa palvelinprosessi kutsuu metodia *impl_is_ready().* Tämä kertoo Orbix-daemonille, että palvelin on alustanut olionsa ja on valmis vastaanottamaan asiakkaiden pyyntöjä. Kuva 17 esittää tätä ORB:n ja palvelimen välistä kommunikointia.



Kuva 17. Kommunikointi Orbixin ja palvelimen tai olion välillä.

Funktiota *impl_is_ready()* voidaan kutsua sellaisella parametrin arvolla, että palvelinohjelma jää ikuiseen silmukkaan odottamaan asiakkaiden palvelupyyntöjä ja käsittelee pyynnöt automaattisesti. Mikäli pyynnöt halutaan käsitellä manuaalisesti, funktiolle *impl_is_ready()* annetaan parametrina 0 (nolla). Tällöin se rekisteröi itsensä Orbixille ja palaa funktiosta heti. Tämän jälkeen ohjelmoija on vastuussa tulevien palvelupyyntöjen käsittelystä käyttämällä Orbixin sovellusrajapinnan (Application Programming Interface, API) tarjoamia kutsuja viestien olemassaolon tarkistamiseen ja lukemiseen.

4.1.6 Olioiden nimeäminen ja yhteyksien muodostaminen

Jokaisella oliolla on yksikäsitteinen tunniste (*marker*) palvelimessaan. Marker on Orbixin oma toteutus CORBAn määrittelemälle olion viitetiedolle (reference data). Orbix luo automaattisesti tunnisteen oliolle, mikäli ohjelmoija ei halua itse määrätä sitä.

Asiakas ottaa yhteyden palvelinolioon funktiolla *rajapinta::_bind()* määrittelemällä kohdeolion, palvelimen ja isäntäkoneen nimen. Olion ja palvelimen nimi on sama, jota käytettiin palvelun rekisteröinnissä *putit*-työkalulla. Isäntäkoneen nimi on Internet-nimi: looginen nimi, kuten *ele307,* tai IP-osoite, kuten 130.188.93.107.

Ottaessaan yhteyden kohdeolioon asiakas voi kertoa minkä tahansa yhdistelmän olion, palvelimen ja isäntäkoneen nimistä:

- *Palvelimen nimeä ei anneta:* oletusarvona palvelimen nimi on sama, kuin *rajapinta::_bind()* -kutsun rajapinta. Tämä on selvä puute olion rajapinnan ja sen toteutuksen eristämisen periaatteen noudattamisessa. Käytännössä tämä pakottaa valitsemaan olion toteuttavan palvelimen nimeksi olion rajapinnan nimen tai määrittämään kohdeolion palvelimen nimen *_bind()*-kutsussa.

- *Olion nimeä ei anneta*: oletuksena on mikä tahansa kohdepalvelimen olio, joka toteuttaa halutun rajapinnan.

- *Isäntäkoneen nimeä ei anneta*: Orbix käyttää oletuspaikannintaan löytääkseen palvelimen sijainnin. Paikannin lukee levyllä olevan tiedoston ja päivittää palvelinten sijaintitiedon muistissa. Ellei tiedostoa ole, vanha sijaintitieto säilyy ennallaan.

Orbix tukee myös asiakkaan ja palvelimen sijoittamista samaan muistiavaruuteen. API-kutsu *CORBA::Orbix.collocated(TRUE)* aiheuttaa virtuaalisten C++-kutsujen käyttämisen asiakkaan ja palvelimen välillä, eikä Orbix yritä ohjata kutsua kutsujan muistiavaruuden ulkopuolelle. Tämä on käyttökelpoinen mekanismi suorituskyvyn parantamiseksi tilanteissa, joissa halutaan käyttää IDL-rajapintoja olioiden välillä, mutta olioita ei ole hajautettu. Hajautuksen päättäminen vasta ajon aikana on myös mahdollista käyttämällä kutsua esimerkiksi komentoriviparametrien mukaan. CORBA ei määrittele tätä ominaisuutta.

4.1.7 DII

CORBA-standardin mukaisesti Orbix tarjoaa dynaamisen kutsumekanismin asiakkaille. Palvelimelle DII:n käyttö on läpinäkyvää: se ei tiedä, lähettikö asiakas operaatiokutsun staattisella vai dynaamisella kutsumekanismilla.

Orbix tarjoaa oman, vuotyyppisen mekanismin DII:n käyttöön CORBA-yhteensopivan tavan lisäksi. Molemmilla tavoilla käytettynä DII näkyy asiakkaalle pieniä poikkeuksia lukuun ottamatta samalla tavalla.

1. Hankitaan olioviittaus kohdeolioon (esim. operaatiokutsun tuloksena tai tiedostosta).

2. Selvitetään IFR:n avulla olion toteuttaman rajapinnan ominaisuudet.

3. Muodostetaan kutsuolio, jolle määritellään kutsuttavan operaation nimi merkkijonomuodossa.

4. Syötetään parametrit ja niiden tyypit (*in, out, inout*) kutsuolioon.

5. Suoritetaan dynaaminen operaatiokutsu, jolle annetaan kutsuolio parametrina.

6. Tutkitaan mahdolliset kutsun palautusarvot.

DII:n käyttö on perusteltua lähinnä seuraavissa tapauksissa:

1) Jos halutaan välttää operaatiokutsun tulosten odottaminen. DII:n viivästetty kutsumuoto mahdollistaa tulosten tarkastelun myöhemmin. Myös monen rinnakkaisen operaatiokutsun suorittaminen on mahdollista.

2) Tehtäessä kutsuja rajapintoihin, joiden tyypit ja ominaisuudet saadaan tietää vasta ohjelman suorituksen aikana.

4.1.8 Lisäominaisuuksia

Seuraavat Orbixin tarjoamat ominaisuudet eivät ole CORBAn määrittelemiä, joten ne eivät ole siirrettävissä sellaisenaan muihin ORB-toteutuksiin. Ne esitellään tässä, koska ne ovat hyvä esimerkki Orbixin lisäpiirteistä ja tarjoavat ohjelmoijalle valmiita keinoja edistyneempien sovellusten toteuttamiseen.

Suodattimet

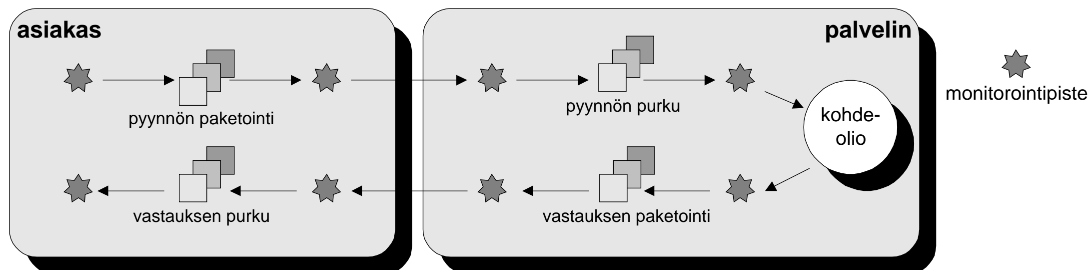
Suodattimet tarjoavat ohjelmoijalle mahdollisuuden suorittaa ylimääräistä koodia ennen tai jälkeen operaatiokutsun tai attribuutin käytön. Tässä koodissa voidaan suorittaa esimerkiksi turvallisuuden kannalta tarpeellisia varmistuksia, tarjota apua ohjelmiston virheiden määrittämiseen, ylläpitää lokitietoa, pakata ylimääräistä tietoa operaatiokutsuihin, aloittaa tietokantatransaktio tai käynnistää uusi säie operaatiokutsun käsittelyyn.

Orbix tukee kahdentyyppisiä suodattimia, prosessikohtaisia ja oliokohtaisia. Prosessikohtaiset suodattimet näkevät kaikki operaatio- ja attribuuttikutsut, jotka tulevat tai lähtevät prosessin muistiavaruudesta. Siksi ne eivät sovellu prosessin sisäisten kutsujen seuraamiseen, kun asiakas ja palvelin sijaitsevat samassa muistiavaruudessa. Oliokohtainen suodatin on yhdistetty tiettyyn olioon, ja sitä voidaan käyttää myös prosessin sisäisten operaatiokutsujen yhteydessä. Oliokohtainen suodatin toimii vain TIE-mekanismia käytettäessä ja ainoastaan palvelimessa.

Ohjelmoija toteuttaa suodattimet määrittelemällä luokan, joka perii Orbix:in luokasta *CORBA::Filter*, ja luomalla instanssin tästä luokasta. Jokaiselle prosessille voidaan luoda suodattimien ketju, joiden ylikirjoitettuja käsittelymetodeja kutsutaan vuorollaan. Ohjelmoija voi määrittää suodattimen käsittelymetodin palautusarvolla, halutaanko operaatiokutsua jatkaa vai tuleeko se keskeyttää ja palauttaa poikkeus asiakkaalle.

Oliokohtaisella suodattimella on vain kaksi monitorointipistettä, ennen kohdeolion metodin kutsumista ja metodin kutsumisen jälkeen. Prosessikohtaisilla suodattimilla puolestaan on kahdeksan monitorointipistettä, joista kutsun tekijän puolella on neljä ja palvelimen puolella neljä. Asiakkaan puolella monitorointipisteitä on kaksi kutsun

lähettämisen ja vastaanottamisen yhteydessä, kummassakin tapauksessa ennen ja jälkeen kutsupaketin käsittelyn. Sama pätee palvelimen puolella pyynnön vastaanottamisen ja sen palauttamisen suhteen. Kuva 18 esittää prosessikohtaisen suodattimen monitorointipisteitä.



Kuva 18. Prosessikohtaisen suodattimen monitorointipisteet. . [8].

Älykkäät proxyt

IDL-kääntäjä luo rajapintaa käyttäville asiakkaille stub-koodin. Tämä stub-koodi sisältää rajapinnan proxy-luokan. Asiakkaan tehdessä operaatiokutsun rajapinnan toteuttavalle oliolle kutsu välitetään proxy-luokan instanssin kautta kohdeoliolle.

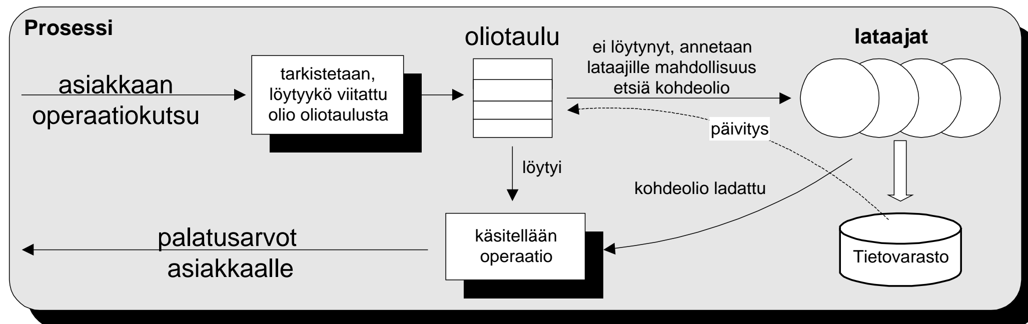
Orbix mahdollistaa proxy-luokkien määrittelyn myös manuaalisesti. Tämä on joissakin tilanteissa käyttökelpoinen ominaisuus rajapintojen toteuttajalle, joka voi tarjoata asiakkaille älykkään proxy-luokan otsikkotiedoston ja toteutuksen objektitiedoston. Älykkään proxyn koodi voi esimerkiksi parantaa suorituskykyä toimimalla välimuistina, jota kohdeolio päivittää. Se voi puskuroida yksittäisiä operaatiokutsuja ja lähettää ne myöhemmin yhdessä tai suorittaa kuormituksen tasausta valitsemalla kohdeolion.

Älykkäät proxyt toteutetaan määrittelemällä luokka, joka perii IDL-kääntäjän generoiman proxy-luokan (sama toimenpide pitää tehdä myös proxy-luokan instansseja luovalle proxy-factory-luokalle). Sen lisäksi tarvitsee vain luoda yksi globaali proxy-factory-luokan instanssi. Asiakasohjelmat pitää kääntää tämän uuden luokan määritelmän ja toteutuksen kanssa, mutta niiden ei tarvitse tietää oletus-proxyn korvanneesta älykkäästä proxysta mitään. [8]

Lataajat

Operaatiokutsun saapuessa prosessille Orbix tarkistaa, löytyykö kohdeolio prosessin ylläpitämästä oliotaulusta. Mikäli kohdeoliota ei löydy eli se ei ole aktiivisena prosessin muistissa, oletuksena palautetaan poikkeus kutsun tehneelle asiakkaalle. Orbixin lataajat mahdollistavat toisenlaisen menettelyn: prosessiin asennetuille lataajille annetaan

mahdollisuus ladata kohdeolio muistiin. Kuva 19 esittää Orbixin lataajien toimintaperiaatetta.



Kuva 19. Orbixin lataajien toiminta.

Lataaja voidaan toteuttaa määrittämällä luokka, joka perii Orbixin valmiista luokasta *CORBA::LoaderClass,* ja luomalla siitä instanssi. Tällöin se rekisteröityy Orbixille ja sitä kutsutaan oletuksena automaattisesti asiakkaan viitatessa ei-aktiiviseen olioon. Lataaja saa parametrina kohdeolion ja sen toteuttaman rajapinnan nimet, joiden avulla se voi etsiä kohdeolion tietovarastosta ja ladata sen muistiin.

Lataajamekanismi on tärkeä apukeino, kun CORBA-olioita talletetaan pysyvästi esimerkiksi tietokantaan. Lataaja voidaan toteuttaa siten, että se lukee viitatun kohdeolion tietokannasta, tiedostosta tai jostain muusta pysyvästä varastosta prosessin muistiin. Kun oliota ei enää tarvita tai on muuten tarkoituksenmukaista, lataaja voi päivittää olion uuden tilan tietovarastoon ja vapauttaa olion käyttämät resurssit.

4.1.9 Resurssien käyttö ja hallinta

Keskusmuistin käyttö

Keskusmuistin käytön tarkastelussa on käytetty tämän työn esimerkkisovelluksen ohjelmistokomponentteja (luku 0). Keskusmuistin käyttö on mitattu Windows NT -käyttöjärjestelmän Task Manager -ohjelmalla. Tarkasteluun otetut kaupalliset ohjelmat ovat Orbix-daemon ja Objectivityn Lock Server. Lock Server tarvitaan valvomaan tietokannan käyttöä. Sovellusohjelmiston keskusmuistin käyttö on esitetty tyypilliselle esimerkkisovelluksen Orbix-sovellukselle ja Trigger Scheduler -sovellukselle, joka käyttää sekä Orbixin että Objectivityn kirjastoja. Taulukko 3 esittää ohjelmakohtaisen keskusmuistin käytön.

Taulukko 3. Esimerkkisovelluksen pääkomponenttien keskusmuistin käyttö.

Komponentti	Muistin käyttö [kB]	Tiedoston koko [kB]
orbixd	700	568
Trigger Scheduler	3500	158 (dynaaminen linkitys)
Orbix-sovellus	900	78 (dynaaminen linkitys) 480 (staattinen linkitys)
Objectivity Lock Manager	800	304

Jos Orbixin kirjastot linkitetään dynaamisesti, suoritettavan tiedoston koko pienenee noin kuudesosaan. Keskusmuistin käyttö ei juurikaan muutu, sillä sovelluksen käyttäessä DLL-kirjastoja sille varataan oma tietoalue - vain DLL-kirjaston koodi jaetaan muiden sovellusten kesken ja ladataan muistiin vain kerran.

Kommunikointiyhteydet

Orbix on toteutettu TCP/IP:n päälle ja käyttää kommunikointiin socket-yhteyksiä. Orbix-daemon kuuntelee oletuksena porttia 1570, jonka kautta sovellukset kommunikoivat sen kanssa. Kun asiakas ottaa yhteyden palvelimeen, Orbix-daemon välittää tarvittavan tiedon asiakkaan ja palvelimen välille luotavasta socket-yhteydestä. Sen jälkeen asiakas ja palvelin kommunikoivat ilman Orbix-daemonia. Koska siirtoprotokollana on TCP, yhteys on luotettava, yhteysperustainen kommunikaatiokanava.

Käyttöjärjestelmä ylläpitää tietoa socket-yhteyksistä tiedostokuvaajien avulla [15]. Tämä rajoittaa aktiivisten yhteyksien lukumäärää ja sovellusten toteuttamistapaa käyttöjärjestelmissä, joissa yhtäaikaisten tiedostokuvaajien suurin sallittu määrä on pieni (esim. 512).

Orbix tarjoaa useita API-kutsuja yhteyksien ominaisuuksien konfigurointiin. Näillä voidaan määrittää käytetyt menettelytavat ja oletukset yhteyksien luomisessa ja niiden lopettamisessa.

4.1.10 Suorituskyky

Orbixin suorituskykyä arvioitiin suorittamalla operaatiokutsuja yhden ja kahden koneen tapauksessa. Kutsujen sekä niiden parametrien tyyppiä ja kokoa vaihdeltiin. Kaikissa viesteissä parametrina käytettiin tietuetta, jonka ainoana kenttänä oli long-tyyppistä

tietoa sisältävä taulukko. Eri viestikoot saatiin vaihtelemalla taulukon kokoa, ja vasteajat saatiin suorittamalla operaatiokutsu 1 000 kertaa peräkkäin ja laskemalla keskiarvo. Taulukko 4 esittää Orbix-version QNX 4.22A[5] 1.3.3.B suorituskykytestin tuloksina saatuja vasteaikoja. Taulukoissa merkintä "tko11/tko105" tarkoittaa asiakasohjelman sijaintia koneessa tko11 ja palvelinohjelman sijaintia koneessa tko105. Kutsut ovat tavallisia kaksisuuntaisia kutsuja, ellei muuta mainita.

QNX-testien tapauksessa kone tko11 oli sulautettu PC/104, jossa oli Intel 486/100 MHz -prosessori, 8 megatavua keskusmuistia ja NE2000 Ethernet -verkkokortti. Kone tko105 oli tavallinen PC, jossa oli Intel 486/66 MHz -prosessori, 32 megatavua keskusmuistia ja NE2000Plus Ethernet -verkkokortti. Tiedonsiirtomedia oli 10BaseT Ethernet -lähiverkko, jossa testisolmut oli eristetty lähiverkon muista osista sillalla.

Taulukko 4. Orbix QNX 4.22A -version suoritusajat.

Kutsun tyyppi	Asiakas / palvelin [ms]			
	tko11 / tko11	tko105 / tko105	tko11 / tko105	tko105 / tko11
olioviittauksen saaminen	10	7	10	10
yksisuuntainen kutsu ilman parametreja	2	1	2	2
viestikoko 4 tavua	8	5	7	7
viestikoko 512 tavua	8	5	8	9
viestikoko 1024 tavua	9	5	10	10
viestikoko 2048 tavua	10	6	12	12
viestikoko 4096 tavua	12	8	17	17

Yksisuuntaiset kutsut olivat kaksi kertaa nopeampia kuin tavalliset kutsut pienellä viestikoolla. Kun viestikoko ylitti yhden kilotavun (1 kB), yksisuuntaisten kutsujen suoritusajat alkoivat vaihdella paljon, jopa 100 %. Syytä tiedusteltiin Orbixin tukipalvelusta saamatta sieltä kuitenkaan ilmiön selittävää vastausta. Mahdollisesti kyseessä oli palvelimen tukkeutuminen (kohta 0).

Myös dynaamista kutsumekanismia kokeiltiin Orbixin QNX-versiolla. Suoritetut testit olivat suppeammat kuin staattisella kutsumekansimilla, mutta kutsut kestivät pääsääntöisesti kaksi kertaa kauemmin kuin staattisella kutsumekanismilla.

---

[5] QNX on QNX Software Systems Ltd:n reaaliaikakäyttöjärjestelmä.

Orbixin Windows NT -testien tapauksessa kone ele307 oli PC, jossa oli Intel Pentium Pro 200 MHz -prosessori, 64 megatavua keskusmuistia ja 3Com Fast EtherLink XL PCI 10/100 Mb -verkkokortti (3C905). Koneessa ele302 oli Intel Pentium Pro 180 MHz -prosessori, 96 megatavua keskusmuistia ja 3Com EtherLink XL -verkkokortti (3C900). Järjestelyt olivat samat kuin QNX-testien yhteydessä, mutta siltaa ei käytetty. Taulukko 5 esittää testitulokset Orbix 2.2:n Windows NT -versiolle. Taulukkoon on merkitty kahdenkymmenen suorituskerran suurin ja pienin arvo.

Taulukko 5. Orbix 2.2 Windows NT -version suoritusajat.

Kutsun tyyppi	Asiakas / palvelin [ms]			
	ele307 / ele307	ele307 / ele302	ele302 / ele302	ele302 / ele307
olioviittauksen saaminen	60	60	80	80
yksisuuntainen kutsu ilman parametreja	0.5 - 1.3	0.3 - 0.4	0.5 - 0.7	0.2 - 0.4
viestikoko 4 tavua	2.5 - 3.2	3.0 - 3.4	3.3 - 3.7	2.9 - 3.1
viestikoko 512 tavua	2.6 - 3.1	2.8 - 4.0	2.3 - 3.3	2.9 - 3.2
viestikoko 1024 tavua	2.5 - 3.2	3.5 - 4.8	2.8 - 3.3	3.4 - 3.6
viestikoko 2048 tavua	2.7 - 3.2	4.2 - 5.7	3.0 - 3.6	4.1 - 4.3
viestikoko 4096 tavua	3.0 - 3.6	6.1 - 7.1	3.2 - 3.6	5.9 - 6.2

Myös yksisuuntaisten kutsujen suoritusaikoja mitattiin 4096 tavun viestikokoon asti, ja ne olivat noin kaksi kertaa nopeampia kuin tavalliset kutsut.

Yhteenvetona voidaan todeta kutsujen keston olevan millisekuntien luokkaa molemmissa Orbix-toteutuksissa. Jos asiakas ja palvelin sijaitsevat samassa muistiavaruudessa, niiden väliset operaatiokutsut voidaan muuttaa suorituskyvyn parantamiseksi tavallisiksi C++-virtuaalimetodien kutsuiksi käyttämällä funktiota *CORBA::Orbix.setCollocation(TRUE).* Tämän mekanismin vasteaikoja ei kuitenkaan mitattu.

4.1.11 Kirjastot

Orbix tarjoaa asennuslevyllään kirjastoja kolmessa hakemistossa: ML-hakemiston kirjastoja käytetään yksisäikeisten sovellusten tekemiseen, MT-hakemiston kirjastoja

monisäikeisten sovellusten tekemiseen ja MD-hakemiston kirjastoja monisäikeisten, dynaamisesti linkitettävien sovellusten tekemiseen. Oletuksena kovalevylle asentuvat MD-hakemiston kirjastot, joita esimerkkisovelluksessa on käytetty. Orbix tukee näitä käyttötarpeita, koska Microsoft Visual C++ 4.2 -kääntäjä jaottelee kirjastonsa näin.

Näistä hakemistoista löytyvät seuraavat kirjastotiedostot:

- **ITC.lib**: yksisäikeisten, merkkipohjaisten sovellusten tekoon
- **ITM.lib**: monisäikeisten, merkkipohjaisten sovellusten tekoon
- **ITG.lib**: yksisäikeisten graafisten käyttöliittymäsovellusten tekoon, jotka samalla toimivat CORBA-palvelimina. ITG.lib hoitaa automaattisesti CORBA-viestien vastaanoton ja käsittelyn palvelimessa ilman ohjelmoijan suorittamaa manuaalista kyselyä.

Näistä kolmesta kirjastosta on olemassa dynaamisesti linkitettävät versiot *ITCI.lib*, *ITMI.lib* ja *ITGI.lib*. Esimerkkisovelluksessa on käytetty kirjastoja *ITGI.lib* ja *ITMI.lib*.

Ohjelmiston sisältämät kirjastot on dokumentoitu Orbix-manuaaleissa puutteellisesti. Tieto on hankittu Internet-verkon uutisryhmistä ja osittain Orbix:in tukipalvelusta. Selkeää ohjetta monisäikeisen, CORBA-palvelimena toimivan graafisen käyttöliittymäsovelluksen tekemiseen ei ole saatu IONAlta. Uutispalstoilla se neuvotaan tekemään *ITM(I).lib*-kirjaston kanssa siten, että CORBA-viestien käsittelyfunktioita kutsutaan ajastimen avulla tai manuaalisesti omassa säikeessään. Kirjastoa *ITG(I).lib* ei ole suojattu rinnakkaisen käytön varalta, joten sitä ei suositella käytettäväksi monisäikeisissä sovelluksissa.

4.1.12 Sovellusten siirrettävyys

Ohjelmistokomponenttien kuvauksissa käytetään IDL-rajapintakuvausta ja toteutus koodataan C++-kielellä. IDL-rajapintakuvausten osalta ohjelmisto on siirrettävissä muihin ORB-ympäristöihin, sillä IDL-rajapintakuvaus on käännettävissä myös muilla kuin Orbixin IDL-kääntäjällä. IDL-rajapinnan C++-toteutus saattaa vaatia muutoksia IDL-kääntäjien mahdollisten eroavaisuuksien vuoksi. Eri ORB-tuotteiden API-funktioita ei ole täysin määritelty CORBA-spesifikaatiossa, ja siten funktioiden toteutukset saattavat vaihdella. Jos sovelluksista tehdään API-kutsuja vähän, siirtovaiva jää kohtuulliseksi.

Suurin osa esimerkkisovelluksen ei-siirrettävästä koodista koostuu Orbix:n oman, rajapintakohtaisen *_bind()*-metodin käytöstä, jolla asiakkaat hankkivat olioviittauksen rajapinnan toteuttavaan kohdeolioon. Tämä *_bind()*-funktio ei myöskään toimi kommunikoitaessa jonkin muun ORB:n kanssa IIOP-mekanismilla, vaan on käytettävä erillisenä tuotteena myytävää Orbixin nimipalvelua (OrbixNames). Tällä hetkellä OrbixNames v. 1.0.3 on saatavissa ilmaiseksi Windows NT 4.0 -käyttöjärjestelmälle.

Siirtotyötä voidaan vähentää toteuttamalla olioviittauksen hankkiminen eri tavoin esimerkiksi makroilla, joista esikääntäjän direktiivien avulla valitaan sovellusympäristöön sopiva.

4.1.13 Käyttökokemuksia

Orbix on uutispalstojen kirjoituksista päätellen laajasti käytetty ja käyttökokemusten mukaan paljon ominaisuuksia tarjoava ORB-toteutus. Ominaisuudet tarjoavat ohjelmoijalle runsaasti valmiita apukeinoja edistyneempien piirteiden toteuttamiseen, ja laajasta käyttäjäkunnasta on konkreettista hyötyä uutispalstoilta saatavan tiedon muodossa.

Suurimmat puuttet Orbixilla oli manuaaleissaan. Manuaalien jäsentely oli heikkoa ja tieto yhteenliittyvistä asioista oli pieninä palasina. Erityistiedot käytettävästä käyttöjärjestelmästä ja kääntäjästä olivat minimaaliset elleivät olemattomat, ja tiedon puute Orbixin kirjastoista hidasti alkuvaiheessa ohjelmankehitystä.

Maksullinen tukipalvelu oli hidas ja yleensä sieltä ei saanut riittävän tarkkaa vastausta esitettyyn kysymykseen. Tämä lienee ymmärrettävää, sillä ohjelmistoa kehittävällä joukolla ei ole resursseja toimia asiakaspalvelussa ja muu henkilöstö ei tiedä toteutuksen yksityiskohtia riittävän tarkasti. Teknisiin yksityiskohtiin liittyvät kysymykset jäivät tyypillisesti vaille vastausta, luultavasti alan kovan kilpailun vuoksi.

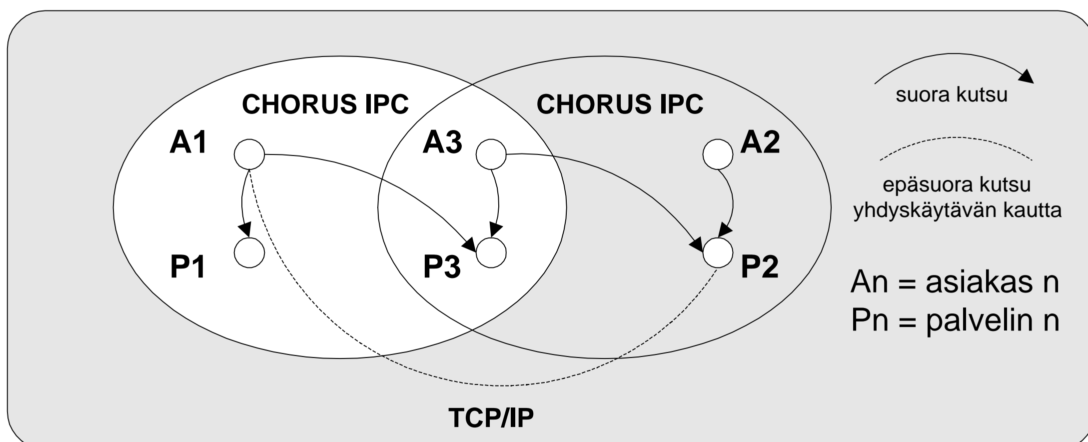
4.2 Chorus/COOL ORB

Chorus/COOL ORB on Chorus Systemsin CORBA 2.0 -määritelmän mukainen ORB-toteutus. Windows NT, Windows 95, Linux, Solaris, SunOS, SCO OpenServer, AIX ja HP/UX ovat käyttöjärjestelmiä, joille Chorus/COOL ORB on toteutettu Choruksen oman käyttöjärjestelmän CHORUS/ClassiX:n lisäksi. [13] Tässä kohdassa tarkastellaan Chorus/COOL ORB v. r4.1:n monisäikeistä versiota Windows NT -käyttöjärjestelmälle.

4.2.1 Toteutuksen rakenne

Chorus/COOL ORB toteuttaa CORBAn määrittelemistä piirteistä C++-kielisidonnan, DII:n, DSI:n, IFR:n ja IIOP:n. IMR:n toteutus puuttuu vielä versiosta r4.1. [13]

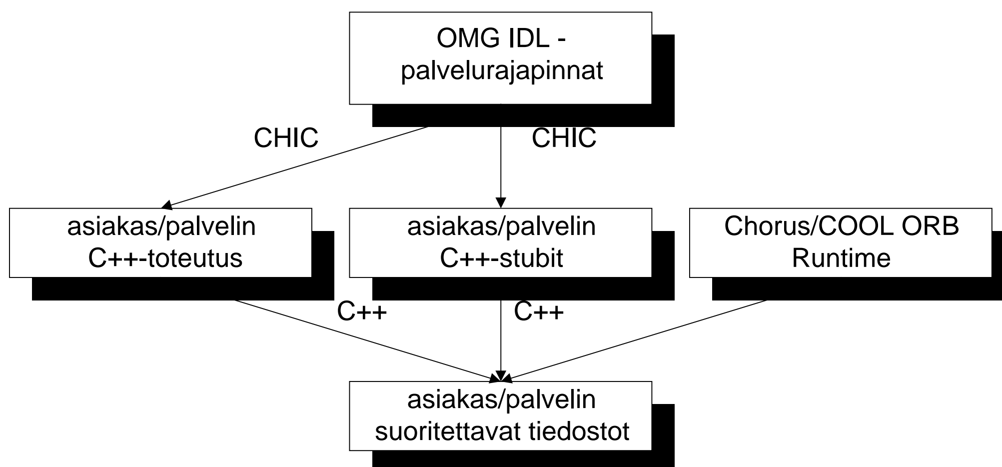
Chorus/COOL ORB on suunniteltu tukemaan erilaisia siirtoprotokollia. Tällä hetkellä tuki on CHORUS IPC:lle (Inter Process Communication) Choruksen omassa käyttöjärjestelmässä ja TCP/IP:lle muissa käyttöjärjestelmissä. Kuva 20 esittää näiden eri siirtoprotokollien käyttöä. Chorus/COOL ORB tukee IIOP:n lisäksi toista kommunikointiprotokollaa, COOL-protokollaa, joka on vähemmän resursseja kuluttava kuin IIOP ja soveltuu siten paremmin sulautettuihin sovelluksiin. Chorus/COOL ORB:n toteutus Choruksen omalle käyttöjärjestelmälle hyödyntää käyttöjärjestelmän mikroydinarkkitehtuuria ja sen , kuten säikeitä, synkronointiprimitiivejä ja CHORUS IPC:tä. [13]



Kuva 20. Chorus/COOL ORB:n tukemat siirtoprotokollat.

Oliota käytetään CORBA-spesifikaation mukaisesti olioviittauksen avulla, joka hankitaan tuotteen sisäänrakennetun nimipalvelun avulla tai saadaan metodin parametrina tai palautusarvona.

Chorus/COOL ORB:n kehitystyökaluista tärkein on CHIC (Chorus IDL Compiler), joka kääntää IDL-rajapintamääritelmät C++-kielen mukaisiksi tyyppi- ja luokkamäärittelyiksi sekä metodeiksi. Kääntäjästä on sekä graafinen että merkkipohjainen toteutus. CHIC:n luoma koodi jakaantuu asiakkaan käyttämiin ja palvelimen käyttämiin osiin. Kuva 21 esittää CHIC:n luomia lähdekoodeja. Graafisten käyttöliittymäsovellusten tekeminen on mahdollista UNIX-käyttöjärjestelmiin X Windowsilla ja Windows-käyttöjärjestelmiin käyttöliittymien kehitystyökaluilla. [13]



Kuva 21. CHIC:n IDL-rajapintamääritelmästä generoimat koodit.

Orbixilla toteutusluokan sitominen rajapintaan tehdään toteutustasolla perimällä luokasta *rajapintaBOAImpl* tai käyttämällä makroa *DEF_TIE(toteutusluokka, rajapinta)* [11]. Chorus/COOL ORB:tä käytettäessä sidonta tehdään antamalla CHIC:lle käännöksen määrittävässä tiedostossa (makefile) parametrina rajapinnan ja toteutusluokan nimet [13]. Näin CHIC osaa luoda skeleton-koodin tälle <rajapinta, luokka> -parille [13]. Toteutuskoodista kutsutaan funktiota *COOL_bind()*, jolla ohjelmassa luodulle oliolle luodaan olioviittaus [13].

Chorus/COOL ORB Runtime tarjoaa peruspalveluina suoritusympäristön olioille, olioiden aktivointimekanismit, operaatiokutsujen suorittamisen, synkronoinnin (semaforit, lukot) ja ilmoitukset vieraista pyynnöistä. Lisäksi se tarjoaa seuraavat IDL-kuvauskielellä määritellyt palvelut:

- **Nimipalvelu (Naming Service)**, joka mahdollistaa olioiden symbolisten nimien ja olioviittausten assosiaatioiden hallinnan.

- **Solmupalvelu (Node Service),** joka tarvitaan jokaisessa solmussa (koneessa), jossa Chorus/COOL ORB -sovellusta suoritetaan. Sitä tarvitaan sovellusolioiden ja hajautettujen synkronointiolioiden luomiseen.

- **Synkronointipalvelu (Synchronisation Service)**, joka mahdollistaa hajautettujen semafori-, mutex- ja lukko-olioiden hallinnan monen lukijan ja yhden kirjoittajan yhtäaikaiseen toimintaan.

- **Toimialapalvelu (Domain Service),** joka on toimialan sisäisten solmujen sekä eri toimialojen yhdistämispiste. Toimiala on samantyyppisen toiminnallisuuden omaavien solmujen muodostama alue.

- **Ryhmäpalvelu (Group Service)**, joka mahdollistaa asiakkaan operaatiokutsun läpinäkyvän kohdistamisen yhdelle tai useammalle palvelimelle. Ryhmäpalvelu tukee ryhmien dynaamista konfiguroimista, kuten palvelinten lisäämistä ryhmään ja niiden poistamista ryhmästä. Sillä on erilaisia toteutuksia operaatiokutsujen välittämiseen palvelimille ja vastausten vastaanottamiseen.

Näiden palvelujen lisäksi ajonaikainen tuki sisältää solmu- ja toimialakohtaiset konfigurointitiedostot (*node.cf, domain.cf*), rajapintavaraston ylläpitäjän *(Interface Repository Manager)* ja työkalut suoritusympäristön hallintaan.

4.2.2 Resurssien käyttö ja hallinta

Chorus/COOL Runtime suorittaa optimointia, kun asiakas ja palvelin ovat samassa muistiavaruudessa. Tällöin käytetään suoraa kutsua viestinvälityksen sijaan. Sovellusohjelmien käännökseen mukaan otettavan koodin määrää voidaan säätää antamalla CHIC:lle haluttuja parametreja IDL-rajapinnan kääntämisen yhteydessä (esim. virtuaaliperinnän tai kaukokutsujen poisjättäminen). Taulukko 6 esittää Windows NT 4.0 -käyttöjärjestelmän Task Manager -ohjelman avulla saadut arvot eri komponettien keskusmuistin käytölle, kun IDL-käännöksessä käytettiin oletusarvoja.

Taulukko 6. Chorus/COOL ORB:n komponenttien keskusmuistin käyttö.

Komponentti	Muistin käyttö [kB]	Tiedoston koko [kB]
nodeManager	610	226
domainManager	620	249
groupManager	600	200
intRepManager	650	285
palvelinsovellus	600	18 (dynaaminen linkitys)

4.2.3 Suorituskyky

Taulukko 7 esittää tuloksia, jotka saatiin mittaamalla operaatiokutsuun kuluva aika kahden Chorus/COOL ORB r4.1 Windows NT -sovelluksen välillä. Testijärjestelyssä kone tko123 oli kannettava tietokone, jossa oli Intel Pentium 70 MHz -prosessori, 16 megatavua keskusmuistia ja 3Com Etherlink III -verkkokortti. Kone tko105 ja taulukon tulkinta ovat muuten samat kuin Orbixin QNX-testien yhteydessä, mutta *ow* tarkoittaa yksisuuntaista kutsua, *in* normaalia kaksisuuntaista operaatiota, jolla on in-tyyppinen parametri, ja *out* sekä *inout* vastaavasti out-parametrin ja inout-parametrin omaavia

kaksisuuntaisia operaatioita. Operaatiokutsujen ajat on saatu tekemällä peräkkäin 100 kutsua ja laskemalla niistä keskiarvo. Taulukkoon on merkitty kunkin operaation testeissä vaatima suurin ja pienin aika.

Taulukko 7. Chorus/COOL ORB r4.1 Windows NT -version suoritusajat.

Kutsun tyyppi	Asiakas / palvelin [ms]			
	tko123 / tko123	tko105 / tko105	tko123 / tko105	tko105 / tko123
olioviittauksen saaminen	10 - 40	20 - 50	40 - 80	40 - 80
viestikoko 4 tavua	ow: 2- 3 in: 8 - 9 out: 8 - 9 inout: 8 - 9	ow: 3 - 4 in: 13 - 14 out: 13 - 14 inout: 13 - 14	ow: 3 - 5 in: 7 - 10 out: 8 - 9 inout: 8 - 9	ow: 4 - 5 in: 7 - 9 out: 8 - 9 inout: 8 - 9
viestikoko 1024 tavua	ow: 5 -6 in: 9 - 10 out: 9 - 10 inout: 9 - 10	ow: 8 - 9 in: 14 - 15 out: 14 - 15 inout: 15 - 16	ow: 4 - 5 in: 9 - 10 out: 9 - 10 inout: 12 - 13	ow: 5 - 6 in: 10 - 11 out: 10 - 11 inout: 13 - 14
viestikoko 2048 tavua	ow: 6 - 7 in: 10 - 11 out: 10 - 11 inout: 11 - 12	ow: 9 - 10 in: 15 - 16 out: 15 - 16 inout: 18 - 19	ow: 6 - 7 in: 11 - 12 out: 12 - 13 inout: 16 - 17	ow: 7 - 8 in: 12 - 14 out: 12 - 13 inout: 17 - 18
viestikoko 4096 tavua	ow: 9 - 10 in: 12 - 13 out: 12 - 13 inout: 15 - 16	ow: 14 - 15 in: 18 - 19 out: 18 - 19 inout: 23 - 24	ow: 10 - 11 in: 14 - 15 out: 17 - 18 inout: 24 - 25	ow: 12 - 13 in: 18 - 19 out: 15 - 16 inout: 25 - 27

Tulosten perusteella parametrin tyypillä (in, out, inout) ei ole merkitystä niin kauan, kun viestin koko on alle yhden kilotavun. Sen jälkeen inout-parametrin käsittelyyn kuluva aika kasvaa enemmän kuin in- ja out-tyyppisten parametrien vaatima aika.

Osa kutsuista katosi, kun yksisuuntaisia kutsuja lähetettiin nopeasti peräkkäin. Lisäksi yksisuuntaiset kutsut käyttäytyivät epämääräisesti kokeiltaessa operaatioiden kutsumista

suuremmilla viesteillä (viestikoko 8kB - 12 kB). Suoritusajat kasvoivat kymmenkertaisiksi paikoitellen, mutta muutokset eivät olleet systemaattisia. Syytä tähän käyttäytymiseen ei pystytty selvittämään, mutta mahdollisesti kysymyksessä oli palvelimen ylikuormittuminen (ks. kohta 3.5.3).

4.2.4 Kirjastot

Chorus/COOL ORB tarjoaa sekä yksi- että monisäikeiset kirjastot käyttöjärjestelmästä ja tilatusta tuotteesta riippuen. Myös tarvittavat kääntäjät on määritelty eri käyttöjärjestelmille. Windows NT -käyttöjärjestelmälle on toteutettu sekä yksi- että monisäikeinen versio, ja tarvittava kääntäjä on Microsoft Visual C++ 4.0.

4.2.5 Käyttökokemuksia

Chorus/COOL ORB:n ohjelmiston dokumentteineen sai hankittua vaivattomasti Internet-verkon kautta. Asentaminen oli kohtuullisen helppoa ja valmiit esimerkit auttoivat ensimmäisen sovelluksen tekemisessä, mikä ei eronnut mainittavasti Orbix-sovelluksen tekemisestä. Nidottuja manuaaleja olisi ollut mukavampi käyttää.

Chorus/COOL ORB:n mukana tuleva nimipalvelu tarjoaa mahdollisuuden olioiden nimiin, jotka ovat täysin riippumattomia kohdeolion toteutustavasta. Orbixilla kohdeolion toteuttavan palvelimen nimi on määritettävä *_bind()*-kutsussa. IMR:n puuttuminen häiritsee, jos palvelimen automaattinen käynnistäminen operaatiokutsun yhteydessä on tarpeen.

Rajapinnan toteutusluokan määritteleminen CHIC:lle IDL-käännöksen yhteydessä on kömpelöä ja rikkoo rajapinnan ja toteutuksen eristämisen periaatetta. Muutenkin Chorus/COOL ORB vaikuttaa ominaisuuksiltaan puutteellisemmalta kuin Orbix.

4.3 Muita ORB-toteutuksia

Muita ORB-toteutuksia ja niiden rakennetta ei tarkastella tässä työssä, vaan jotkin niistä esitellään taulukon muodossa valmistajiensa kanssa. Taulukko 8 esittää osaa helmikuussa 1998 markkinoilta löytyvistä ORB-toteutuksista.

Taulukko 8. ORB-toteutuksia.

Toteutus	Valmistaja	Yhteystiedot
Orbix	IONA Technologies	http://www-irl.iona.com/index.html
Chorus/COOL	Chorus Systems	http://www.chorus.com/
Visibroker	Visigenic Software	http://www.visigenic.com/
CORBAplus	Expersoft	http://www.expersoft.com/
HP ORB Plus	Hewlett-Packard	http://www.hp.com/gsy/orbplus.html
DAIS	ICL	http://www.iclsoft.com/sbs/daismenu.html
NEO	SUN Microsystems	http://www.sun.com/solaris/neo/
OmniORB2	ORL	http://www.orl.co.uk/omniORB/omniORB.html
SOM	IBM	http://www.software.ibm.com/ad/somobjects/
ObjectBus	TIBCO	http://www.tibco.com/products/object_bus.html
OmniBroker	OOC Inc.	http://www.ooc.com/ob.html

Taulukko ei ole kattava, sillä uusia toteutuksia syntyy jatkuvasti ja vanhoja lopetetaan tai fuusioidaan. Osa ORB-toteutuksista on ilmaisia, toisista saa ilmaiseksi vain evaluointiversion ja jotkut toteutukset ovat kaikilta osin kaupallisia. Jotkut ORB:t toteuttavat viimeisimmän CORBA-määritelmän vain osittain tai tekevät laajennuksia tai muunnoksia määritelmän ominaisuuksiin esimerkiksi suorituskyvyn parantamiseksi.

ORB-toteutusten CORBA-määritelmän ulkopuolelta toteuttamat ominaisuudet haittaavat sovellusten siirtämistä järjestelmästä toiseen, mutta usein ne tarjoavat erinomaisia keinoja järjestelmän toiminnallisuuden laajentamiseen. Esimerkiksi Orbixin lataajat ja suodattimet tarjoavat asiakkaan kannalta läpinäkyvän tavan lisätoimintojen toteuttamiseksi palvelimeen.

Tarjolla olevien ORB-toteutusten kirjo on siis hyvin laaja. Markkinoilta löytyy kuitenkin runsaasti myös ilmaisia ORB-toteutuksia. Niiden avulla alkuinvestoinnit saa pieniksi, käsitteet ja ongelma-alueet tulevat tutuiksi ja mikäli toteutus on pitkälle CORBA-yhteensopiva, siirtyminen toiseen ORB-tuotteeseen on helppoa.

5. CORBAN SOVELTAMINEN JOUSTAVAAN VALMISTUSJÄRJESTELMÄÄN

Tässä luvussa tarkastellaan CORBA-standardia ja erityisesti IONA Technologiesin Orbix-toteutusta joustavan valmistusjärjestelmän ohjausohjelmiston kehittämisessä. Luvussa kuvattu ohjelmisto on toteutettu OY Mercantile AB Fastemsin esimerkkisovellukseen.

5.1 Tarkoitus

Esimerkkisovelluksen tarkoituksena oli toteuttaa FM-järjestelmää ohjaava perusohjelmisto CORBA-arkkitehtuuriin, ECA-konseptiin [14] ja oliotietokantaan perustuen. Tämän toteutuksen avulla tuli selvittää, miten CORBA soveltuu ohjelmiston komponenttien kommunikointiväyläksi ja hajautusalustaksi. Lisäksi tarkoituksena oli tutkia, mitä uutta CORBA voi tarjota nykyisiin työkaluihin ja menetelmiin verrattuna sekä mahdollisia CORBA-arkkitehtuurin tai sen toteutuksen puutteita ja rajoituksia.

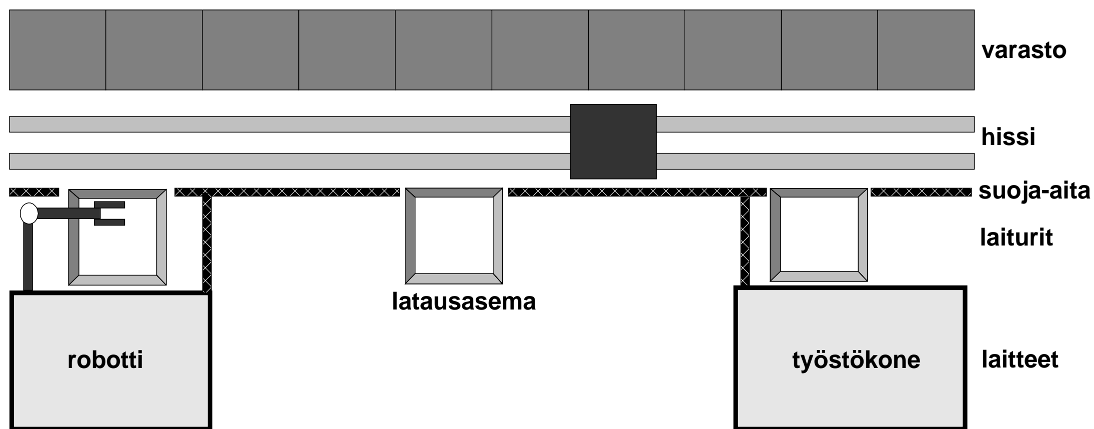
5.2 CORBAn soveltamisen rajaus

Esimerkkisovelluksen toteuttamisessa käytetty ORB-tuote oli IONA Technologiesin Orbix. Siksi tässä työssä ei tarkastella muiden ORB-toteututen soveltuvuutta FM-järjestelmiin. Käytetty Orbixin versio oli Orbix 2.2c01 MT Windows NT -käyttöjärjestelmälle ja Microsoft Visual C++ 4.2 -kääntäjälle.

Esimerkkisovelluksessa CORBAn rooli on Windows NT -tason sovellusten välisen hajautetun kommunikoinnin mahdollistaminen ja komponenttiperustaisen arkkitehtuurin toteuttaminen.

5.3 Esimerkkijärjestelmän rakenne

Esimerkkijärjestelmä koostuu varastosta, työstettäviä alustoja siirtävästä hissistä, kahdesta laitteesta ja latausasemasta. Hissi siirtää alustoja varaston sekä laitteiden ja latausaseman laitureiden välillä käyttäjän ohjauksen mukaisesti. Robotti ottaa laiturillaan olevalta alustalta osan yksi kerrallaan työstettäväksi. Työstökone ottaa koko alustan laiturilta työstötilaansa ja ajaa valitut ohjelmat alustalle. Kuva 22 esittää esimerkkijärjestelmää ylhäältä kuvattuna.



Kuva 22. Esimerkkijärjestelmä.

5.4 Ohjelmistoarkkitehtuuri

Esimerkkisovellus on toteutettu CORBAn avulla hajautettujen olioiden asiakas-palvelinarkkitehtuurina. Koska kohdejärjestelmän luonne on pitkälle tapahtumapohjainen, järjestelmän ohjaus on toteutettu osittain oliotietokantaan talletettuina ECA-sääntöinä.

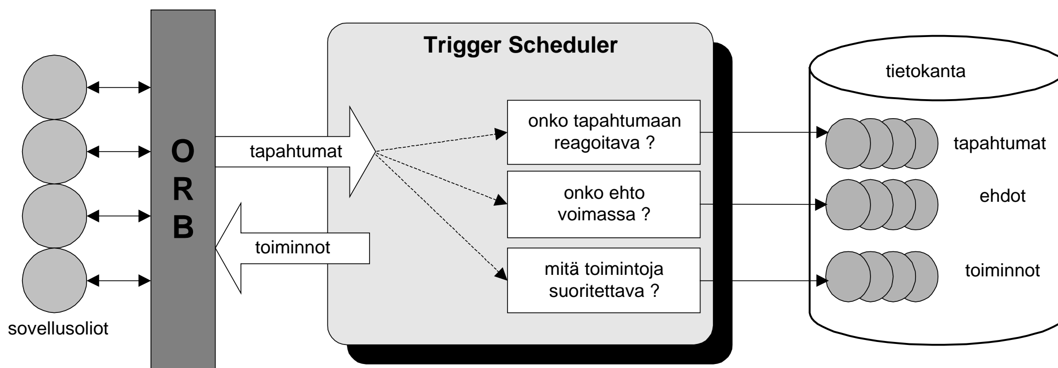
5.4.1 ECA

ECA on lyhenne sanoista event, condition ja action. Nämä kuvaavat järjestelmän tapahtumia, ehtoja tapahtumiin reagoimiseksi ja ehdon täyttyessä tapahtumien vasteeksi suoritettavia toimintoja. ECA-konsepti on käytännössä ohjelmistolla toteutettava tilakone, jossa assosiaatiot tapahtumien, ehtojen ja toimintojen välillä ovat dynaamisesti muutettavissa ohjelman ajon aikana.

Monissa järjestelmissä sovellukset käyttävät yhteistä tietokantaa ja ovat riippuvaisia järjestelmän tilasta eli tietokannan sisällöstä. Sovellusten tulee havaita järjestelmän tilanmuutokset, jotta ne voivat suorittaa tarvittavat toimenpiteet eri tilanteissa. Perinteiset *passiiviset* tietokannat tallettavat järjestelmän tilan, mutta sovellusten on suoritettava jaksoittaisesti kyselyjä tietokantaan kriittisten tilojen havaitsemiseksi. Tämä laskee järjestelmän suorituskykä, sillä yleensä riittävän suuri kyselytiheys aiheuttaa lähinnä turhia kyselyjä. Toinen menettelytapa on, että tietokantaa päivittävä sovellus tarkistaa päivityksen edellyttämät toimenpiteet. Tämä niin sanottu *proseduraalinen upotus* rikkoo kuitenkin ohjelmiston modulaarisuutta hajauttamalla järjestelmän ohjauksen sovelluksiin ja moninkertaistaa muutostyön ohjauksen muuttuessa. [14]

Aktiiviset tietokannan hallintajärjestelmät yrittävät tarjota sekä modulaarisen rakenteen että tarvittavat vasteet järjestelmän tapahtumiin. Järjestelmän tilat, niiden vaatimat toiminnot ja ajoitusvaatimukset määritellään aktiivisen tietokannan hallintajärjestelmälle, joka vastaa sovellusten toiminnan ohjaamisesta määrittelyn mukaan ilman käyttäjän tai sovellusten myötävaikutusta. [14]

Esimerkkisovelluksessa ECA-konsepti on toteutettu sovellusten ja tietokantaa käyttävän ohjausosan, Trigger Schedulerin (TS), yhteistoimintana. Järjestelmän komponenttien tilan muuttuessa ne generoivat CORBA-väylässä välitettävän tapahtuman TS:lle. TS käyttää tietokantaan talletettuja ECA-sääntöjä tarkistaakseen, miten tapahtumaan tulee reagoida. Jos tapahtumaan täytyy reagoida, TS tarkistaa, onko tapahtumaan liitetty ehto tosi. Mikäli ehto täyttyy, TS suorittaa taphtumaan ja ehtoon liitetyt toiminnot. Kuva 23 esittää yksinkertaistettuna ECA-konseptin toimintaa esimerkkisovelluksessa.



Kuva 23. ECA-konsepti esimerkkisovelluksessa.

5.4.2 Ohjelmistokomponentit

Kuva 24 esittää esimerkkisovelluksen CORBA-komponentteja. Komponenttien tehtävät ovat seuraavat:

- **Trigger Scheduler** ohjaa järjestelmän toimintaa tietokantaan talletettujen ECA-sääntöjen mukaan. Se vastaanottaa järjestelmän tapahtumat, analysoi tilanmuutokset ja aktivoi tarvittavat toiminnot.

- **Päävalvomo** tarjoaa graafisen käyttöliittymän järjestelmän ohjaukseen. Se mahdollistaa järjestelmän tilan seuraamisen ja alustojen manuaalisen siirtämisen. Päävalvomo toimii pelkästään CORBA-asiakkaan roolissa.

- **Hissivuorontaja** ylläpitää tietoa järjestelmän vapaista laitteista ja siirtoa odottavista alustoista. Näiden tietojen perusteella se tekee päätökset siirroista ja välittää siirtotehtävän Hissikoneelle.

- **Hissikone** on varsinainen siirtojen suorittaja. Se välittää siirtokäskyt Sovittimen avulla järjestelmän laitteita ohjaavalle ohjelmoitavalle logiikalle.

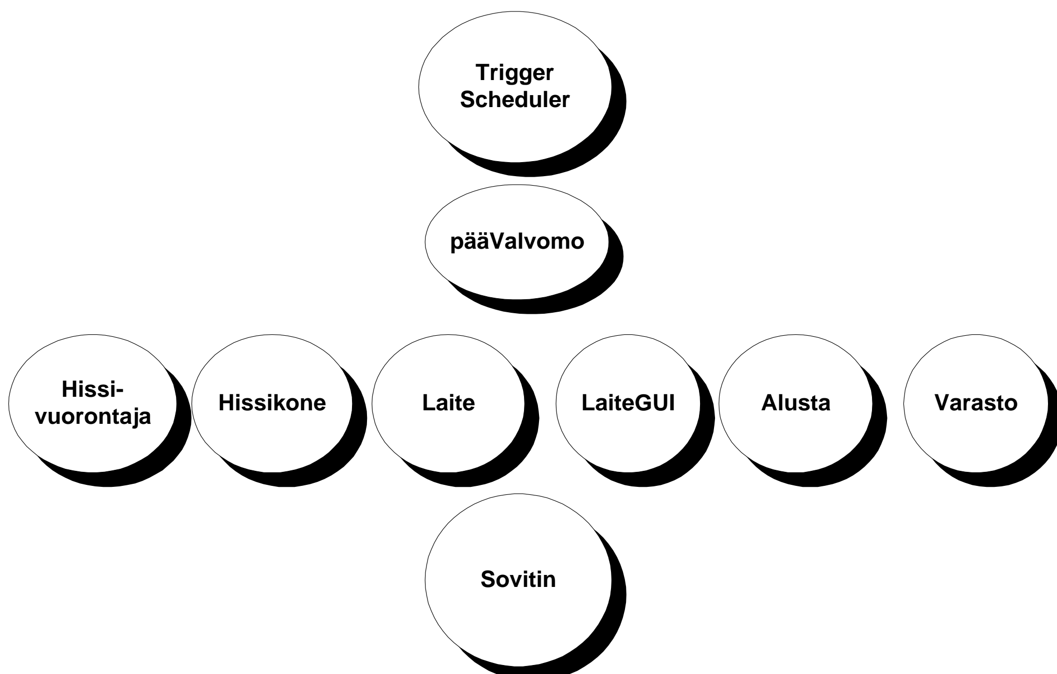
- **Laite** on työstökoneen tai latausaseman toimintaa ohjaava ja tilatietoa ylläpitävä komponentti. Työstökone välittää ohjauskomennot Sovittimen kautta ohjelmoitavalle logiikalle.

- **LaiteGUI** on graafinen käyttöliittymä laitteen tilan seuraamiseen ja ohjaamiseen. Se kohdistaa käyttäjän antamat komennot kohteeksi valitulle Laite-oliolle.

- **Alusta** edustaa järjestelmässä siirreltävää kuljetusalustaa.

- **Varasto** ylläpitää tietoa järjestelmän varastossa olevista alustoista.

- **Sovitin** toimii viestinvälittäjänä CORBA-komponenttien ja ohjelmoitavan logiikan välillä. Se muuntaa CORBA-viestit logiikan ymmärtämään muotoon ja päinvastoin.



Kuva 24. Esimerkkisovelluksen CORBA-komponentit.

Laite-luokasta on johdettu kaksi aliluokkaa, Työstökone ja Latausasema. Näillä kahdella järjestelmän todellista laitetta edustavalla aliluokalla on yhteisiä toimintoja, kuten tilan

ja alustan päivittäminen ja lukeminen. Lisäksi työstökoneella on lisätoimintoja, kuten työstäminen ja ohjelman päivitys. Laite-kantaluokan alustan päivittämiseen käytetty looginen operaatio vaatii myös erilaista toteutusta johdetuissa luokissa: työstökone voi työstää sille tuodun alustan, mutta latausasemalle riittää tilatiedon päivittäminen.

5.4.3 Käyttöliittymät

Perusohjelmiston käyttöliittymät ovat päävalvomon ja varaston sekä laitteiden ja alustojen hallinnan graafiset käyttöliittymät. Laitteet, Hissivuorontaja, Hissikone sekä Varasto on toteutettu merkkipohjaisina sovelluksina, jotka tulostavat ikkunaansa tilatietoa. Suurin osa muista sovelluksista on käynnistettävissä graafisen päävalikon avulla. Vaihtoehtoisesti sovellukset voidaan käynnistää komentoriviltä.

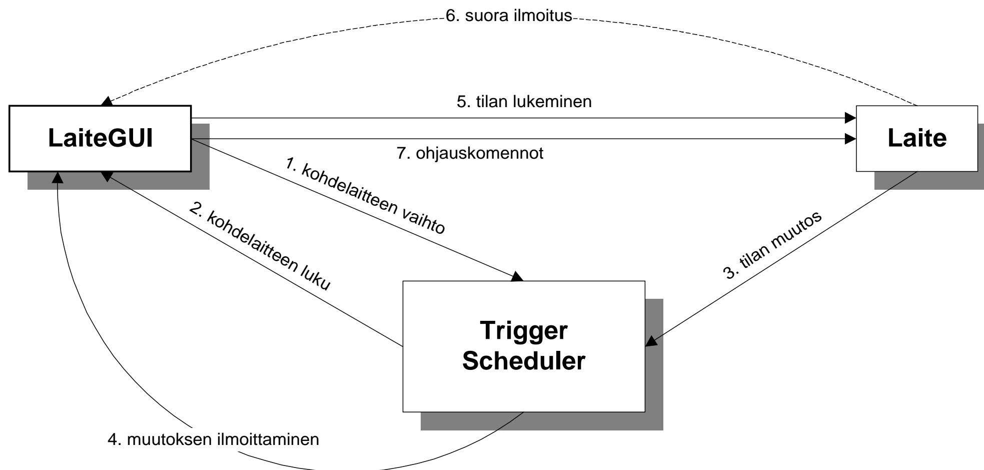
Kaikki käyttöliittymät toteutettiin dialogipohjaisina MFC-sovelluksina, koska tämä toteutustapa sopi paremmin sovellusten luonteelle ja oli helpompi käyttää ja ymmärtää kuin dokumenttipohjaiset MFC-sovellukset.

Tutkimuksen kohteena olivat keinot, joilla CORBA-palvelin voidaan toteuttaa graafisessa käyttöliittymäsovelluksessa ja sovelluksen CORBA- ja käyttöliittymäosuus saadaan mahdollisimman riippumattomiksi toisistaan ohjelman siirrettävyysvaivan minimoimiseksi. Toteutuksessa päädyttiin ratkaisuun, jossa käyttöliittymäprosessi luo instanssin CORBA-oliosta ja antaa Orbixin kirjaston *(ITGI.lib)* käsitellä saapuvat CORBA-asiakaspyynnöt automaattisesti. Vaihtoehtoina olivat manuaalinen CORBA-viestien kysely ja omien säikeiden luominen CORBA-viestien käsittelylle ja käyttöliittymän toiminnoille.

CORBA-asiakasroolissa toimivien käyttöliittymien CORBA-koodi kannattaa eristää oman CORBA-asiakasluokan sisään. Tämä helpottaa muutos- ja kehitystyötä, varsinkin jos käyttöliittymiä toteuttaa eri henkilö kuin CORBA-kommunikointia. Liiallinen modulaarisuuden toteuttaminen sisäkkäisten olioiden avulla heikentää kuitenkin koodin ymmärrettävyyttä ja suorituskykyä.

Laitteiden käyttöliittymäsovellukset toteutettiin erillisinä laitteita ohjaavista sovelluksista, jotta käyttöliittymän toimintojen tai teknologian muuttuessa itse laitetta ohjaaviin komponentteihin tarvitsisi tehdä mahdollisimman vähän muutoksia. Laitteen käyttöliittymä käyttää laitteen tarjoamaa CORBA-rajapintaa laitteen ohjaukseen ja sen tilan lukemiseen tilan muuttuessa. Laitteen ei tarvitse tietää sitä ohjaavasta käyttöliittymästä mitään, sillä sen tilanmuutokset lähettävät tapahtuman TS:lle, joka puolestaan kertoo laitteen käyttöliittymille laitteen tilan muuttuneen. Jos laitteelta

halutaan lähettää viestejä suoraan käyttöliittymille, laiteolio voi käyttää käyttöliittymän tarjoamaa CORBA-rajapintaa. Kuva 25 esittää tätä kommunikointimekanismia.



Kuva 25. Laitteen ja sen käyttöliittymän välinen kommunikointi

Komponenttien eristäminen tarjoaa myös ajonaikaista joustavuutta. Käyttöliittymän ei tarvitse sijaita samassa koneessa kuin kohdelaitteen, vaikka kohdelaite ohjaisikin paikkariippuvalla tavalla (esim. sarjaportin kautta) järjestelmää. Samalle kohdelaitteelle voidaan tarvittaesssa käynnistää monta erillistä käyttöliittymää ja käyttöliittymä voidaan jättää kokonaan pois esimerkiksi muistin säästämiseksi.

Eristäminen aiheuttaa kuitenkin ylimääräistä viestinvälitystä ja sitä kautta ymmärrettävyyden heikkenemistä. Mikäli halutaan optimoida vasteajat ja pitää komponenttien välinen kommunikointi mahdollisimman selkeänä, eristäminen voidaan jättää pois prosessitasolla. CORBAn IDL-rajapintakuvausta voidaan käyttää myös prosessin sisäisten olioiden välillä käytävään kommunikointiin, joten käyttöliittymä voidaan eristää kohdelaitteestaan myös oliotasolla vasteaikojen siitä mainittavasti kärsimättä.

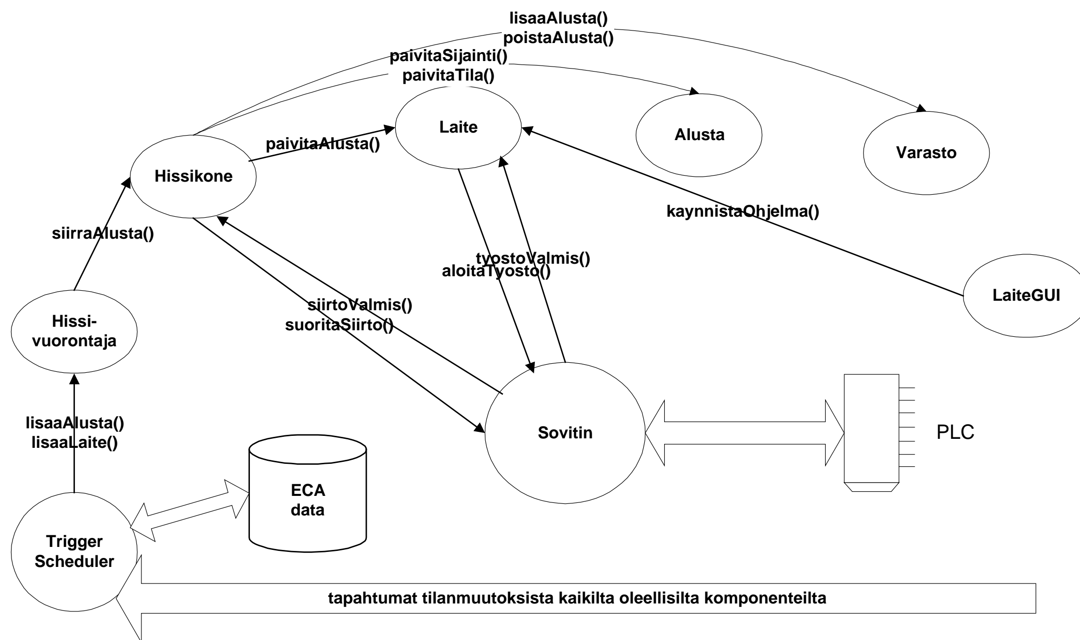
5.4.4 Tietokanta

Esimerkkisovelluksessa vain ECA-säännöt ja laitteiden ja niiden seuraamien käyttöliittymien väliset assosiaatiot talletettiin oliotietokantaan. Todellisuudessa koko järjestelmän tila tulee saada tietokantaan mahdollisten virhetilanteiden ja niistä palautumisen vuoksi. Koska järjestelmän olioiden tilaa ei esimerkkisovelluksessa talletettu tietokantaan, mahdollisia tietokannan käyttötapoja analysoidaan tarkemmin kohdassa 5.12.

5.5 Esimerkkijärjestelmän toiminnan looginen kuvaus

Mikä tahansa järjestelmän komponenteista voi ilmaista tilansa muuttuneen lähettämällä TS:lle tapahtuman. Tapahtuma sisältää tiedon tapahtuman lähettäneestä oliosta, sen toteutusluokasta sekä metodista, jonka kutsuminen on aiheuttanut tapahtuman lähettämisen. TS tarkistaa ECA-säännöistä, pitääkö vastaanotettuun tapahtumaan reagoida. Jos pitää, se tarkistaa, onko tapahtumaan liitetty ehto voimassa. Mikäli ehto on voimassa, TS suorittaa tarvittavat operaatiot.

Kuva 26 esittää tilannetta, jossa Laite- ja Alusta-olioiden luomisesta on lähetetty tapahtuma TS:lle. ECA-säännöt määrittelevät, että jos luodun alustan tila on “*odottaa siirtoa*”, alusta tulee lisätä Hissivuorontajan listaan. Jos luodun laitteen tila on “*vapaa*”, laite tulee lisätä Hissivuorontajan listaan. Alustan tai laitteen lisäämisen ja Hissikoneen vapautumisen seurauksena Hissivuorontaja valitsee siirrettävän alustan. Valinnan tehtyään Hissivuorontaja pyytää Hissikonetta siirtämään valitun alustan. Hissikonetta edustava CORBA-komponentti välittää siirtopyynnön Sovittimen kautta ohjelmoitavalle logiikalle, mistä se välittyy edelleen kohdelaitteelle. Hissikoneen saatua Sovittimelta tiedon siirron valmistumisesta, se kertoo alustan kohteelle alustan saapumisesta ja päivittää alustan sijaintitiedon. Mikäli alusta siirrettiin työstökoneelle, työstökonetta edustava Laite-olio lukee alustan tietorakenteesta alustalle ajettavat ohjelmat. Laite-olio käynnistää ohjelmien ajon antamalla kohdelaitteelle käynnistyskomennot Sovittimen ja logiikan kautta. Kun kaikki ohjelmat on ajettu, Laite-olio päivittää Alusta-olion kohteen. Näin alustan tilaksi tulee jälleen “*odottaa siirtoa*”, ja uusi tapahtuma käynnistää uuden siirtojakson.

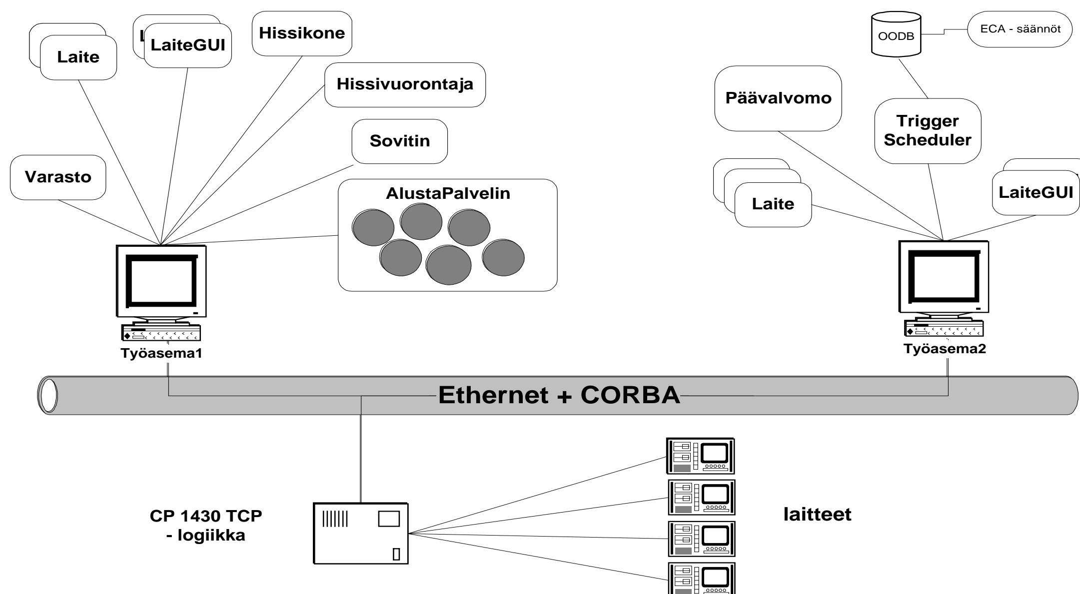


Kuva 26. Järjestelmän looginen toiminta.

Laitetta seuraavat ja ohjaavat käyttöliittymät voivat antaa komentoja laitteelle. Laite välittää käyttäjän antamat komennot Sovittimen kautta laitetasolle.

5.6 Ohjelmiston komponenttien hajautus

FM-järjestelmän prosessinohjaustaso on yleensä toteutettu ohjelmoitavilla logiikoilla, joita ohjelmisto voi ohjata esimerkiksi Ethernet-, RS-232- tai kenttäväyläliitynnän kautta. Esimerkkijärjestelmässä liityntämediana on Ethernet-lähiverkko ja verkon solmuina toimii Siemensin TCP/IP-yhteydellä varustettu logiikkakortti CP 1430 TCP ja kaksi Windows NT -työasemaa. Kuva 27 esittää yleistettynä ohjelmistokomponenttien kohdentamista järjestelmän laitteistolle.



Kuva 27. Ohjelmistokomponenttien kohdentaminen laitteistolle.

Esimerkkisovelluksessa kaikki Laite-oliot on toteutettu omina palvelinprosesseinaan. Tämä menettely kuluttaa enemmän muisti- ja yhteysresursseja kuin olioiden sijoittaminen samaan muistiavaruuteen. Tämä toteutustapa valittiin, koska laitteiden lukumäärä järjestelmässä on pieni ja niiden tulee olla mahdollisimman riippumattomia toisistaan. Lisäksi oman prosessin allokoiminen jokaiselle Laite-oliolle helpottaa virhetilanteista palautumista ja sovellusten testaamista.

Kaikista järjestelmässä liikkuvista alustoista vastaa yksi palvelinprosessi, AlustaPalvelin. Alustojen lukumäärä järjestelmässä saattaa kasvaa suureksi, ja ne ovat toteutetussa ohjelmistossa passiivisia, vain vähän toiminnallisuutta sisältäviä olioita. AlustaPalvelin tarjoaa käyttäjälle keinot alustojen luomiseen, niiden reittien määrittämiseen, tilan seuraamiseen ja manuaaliseen ohjaukseen.

Myös Hissivuorontaja ja Hissikone on toteutettu omina palvelinprosesseinaan. Kummastakin luokasta voidaan luoda useampia instansseja. Hissikone on eristetty Hissivuorontajasta, jotta yksi Hissivuorontaja voi tarvittaessa ohjata useampaa Hissikonetta. Hissivuorontajan monistaminen aiheuttaa lisää synkronointitarvetta, jotta yhdenaikaisten siirtojen ajastus on mahdollista.

Järjestelmän olioiden määrän kasvaessa suureksi kannattaa miettiä pienimpien olioiden toteuttamista muina kuin CORBA-olioina. Tällöin olioilla voi olla niitä ylläpitävä CORBA-palvelin, jonka kautta CORBA-asiakkaat käyttävät olioita. Itse oliot voivat kuitenkin olla C++- tai tietokantaolioita. Tällöin keskusmuistin käyttötarve laskee ja suorituskyky kasvaa. Palvelimen ylläpitämiin olioihin ei kuitenkaan ole enää käytössä

hajautettua olioviittausta, vaan olioita pitää käsitellä palvelimen kautta olion tunnisteen avulla.

5.7 Esimerkkijärjestelmän liitynnät

Järjestelmää ohjataan Windows NT -tasolta graafisilla käyttöliittymillä. Ohjauskäskyjä ovat lähinnä siirtokäskyt hissille ja työstökäskyt työstökoneelle. Hissin siirtokäskyt aktivoituvat automaattiajossa alustojen tilanmuutoksista ja manuaaliajossa käyttäjän päävalvomosta suorittamista Vedä & Pudota -toiminnoista (Drag & Drop). Työstökonetta ohjaa automaattiajossa työstökoneen CORBA-komponentti, mutta manuaaliajossa työstö aktivoidaan laitteen käyttöliittymältä.

Ohjelmoitava logiikka lähettää tilatietoa Windows NT -tasolle aina järjestelmän laitteiden tilan muuttuessa. Sovitin ottaa vastaan kaikki logiikalta tulevat tilatietoilmoitukset, ja tarvittava osa viesteistä välitetään eteenpäin asianomaisille CORBA-prosesseille ORB-väylää pitkin. Kuva 28 esittää esimerkkijärjestelmän tuloja ja lähtöjä.

Kuva 28. Esimerkkijärjestelmän tulot ja lähdöt.

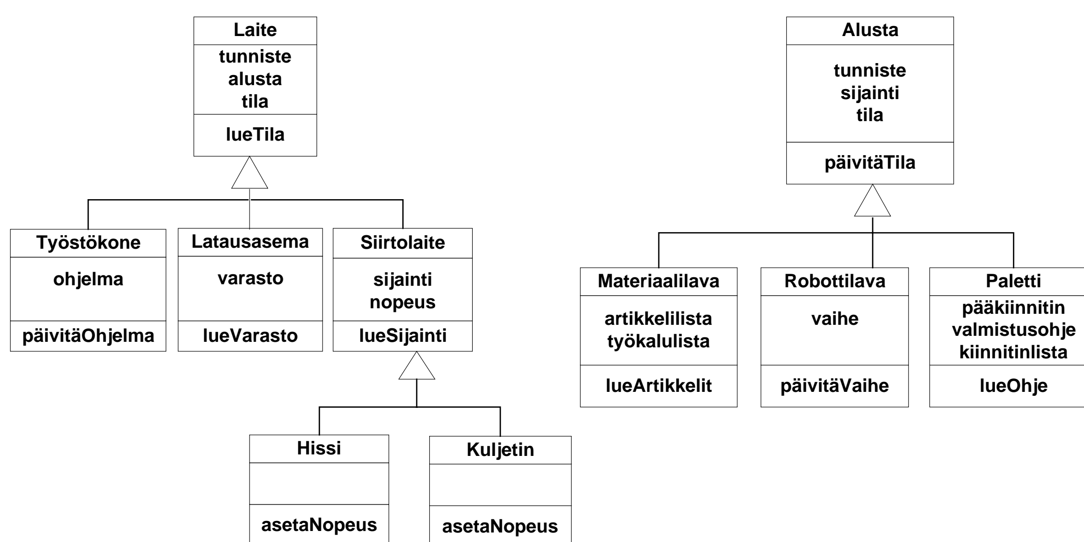
5.8 Sovellusohjelmiston komponentointi

CORBA-oliot ovat järkevästi toteutettuna valmiita komponentteja, joista uusi järjestelmä voidaan muodostaa pienellä räätälöinnillä. Tämä edellyttää aluksi mittavaa työtä, jotta eri protokollilla ja erilaisilla loogisilla variaatioilla toimivat ohjaukset saadaan toteutettua CORBA-rajapintojen taakse piilotetulla koodilla.

Olemassa olevien mekanismien toteuttamisen lisäksi muutostyötä aiheuttavat jatkuvasti markkinoille tulevat uudet ohjausmenetelmät. Esimerkiksi Siemensiltä on valmistunut

CP 1430 TCP -tyylinen kortti, joka ei enää tarvitse RFC 1006 -protokollaa[6] TCP/IP-ympäristössä. Uudet mekanismit kannattaa toteuttaa mahdollisuuksien mukaan perinnän avulla C++-luokkina, jotka piilotetaan CORBA-rajapinnan taakse.

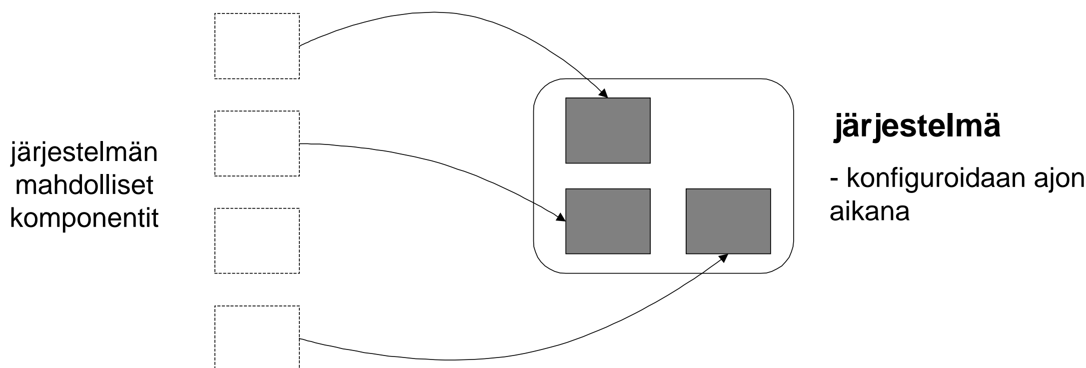
Jos mekanismit suunnitellaan perusteellisesti sovellusalueen tarpeisiin ja uusia otetaan harvoin käyttöön, sovellusten rakentaminen noudattaa niin sanottua *black-box*-menettelyä: uusi ohjelmisto kootaan valmiista komponenteista ilman komponenttien uudelleenkääntämistä [17, 18]. Järjestelmä rakennetaan tällöin valitsemalla luotavien instanssien lukumäärä ja loogiset nimet. Kun ohjelmiston looginen toiminta muuttuu tai tarvitaan uusia mekanismeja, käytetään niin sanottua *white-box*-menettelyä: valmiista komponenteista johdetaan uusia, muunneltuja tai paranneltuja komponentteja [17, 18]. Kuva 29 esittää osaa perintämekanismilla toteutettavasta järjestelmän komponenttihierarkiasta.



Kuva 29. Osa järjestelmän komponenttihierarkiaa.

CORBA-komponentoinnin onnistuminen vaikuttaa paljon järjestelmän jatkokehittämiseen. Tavoitteena on mallintaa sovellusalueen komponentit IDL-rajapintahierarkialla, jotka toteutetaan ORB-tuotteen edellyttämällä ohjelmointikielellä. Kerran toteutettuna komponentit ovat valittavissa ja muokattavissa ohjelmistonkehityksen aikana, ja äärimmilleen vietynä erilaisista komponenteista luodaan järjestelmä pelkästään konfigurointitiedon avulla ilman koodin muokkausta tai uudelleen kääntämistä. Kuva 30 esittää tätä black-box-menetelmää.

[6] RFC 1006 on protokolla, joka mahdollistaa OSI-sovellusten toimimisen TCP/IP-ympäristössä.



Kuva 30. Järjestelmän luominen black-box -menetelmällä.

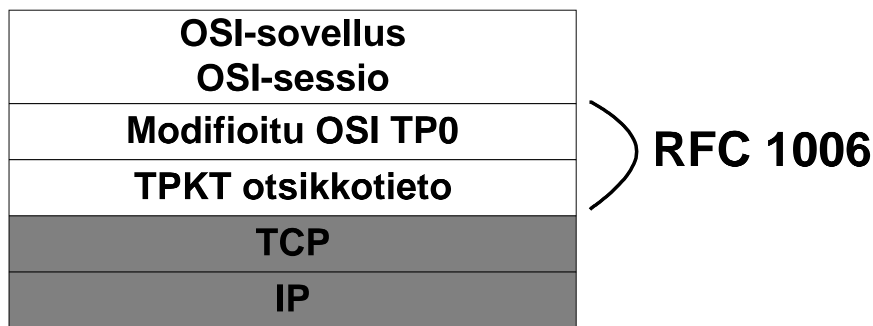
Järjestelmien ominaisuuksien jatkuva muuttuminen ja lisääntyminen mahdollistaa harvoin täydellistä black-box-ohjelmankehitystä. Komponentointityön edut tulevatkin parhaiten esille muutettaessa järjestelmän ominaisuuksia johtamalla vanhoista luokista uusia perinnän ja ylikuormituksen avulla.

5.9 Kommunikointiprotokollat

Järjestelmän hajautus esimerkkijärjestelmässä perustuu TCP/IP-protokollan käyttöön. CORBA-prosessit kommunikoivat keskenään TCP/IP:n päällä toimivalla Orbix-ohjelmistolla ja CP 1430 TCP -kortti tarjoaa TCP/IP-liitynnän, johon voidaan ottaa normaali socket-yhteys Windows NT -työasemassa suorittuvasta sovelluksesta.

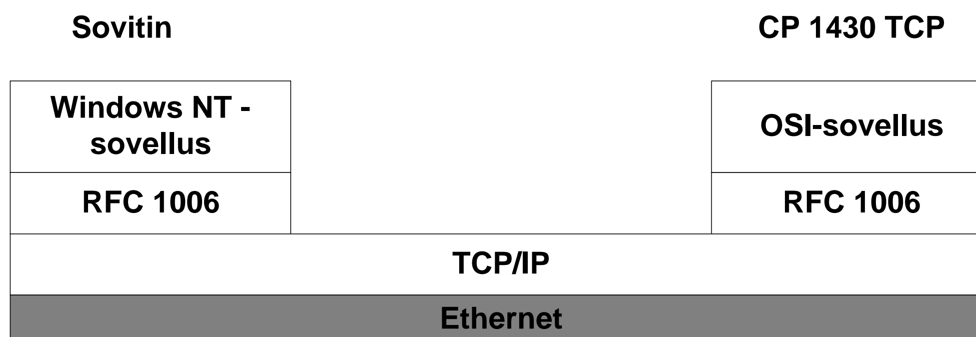
Kommunikointi kortin kanssa vaatii socket-yhteyden lisäksi RFC 1006 -protokollan noudattamista kaikessa viestinvälityksessä. Käytännössä tämä tarkoittaa jokaiseen TCP/IP-pakettiin sisällytettävää määrämuotoista otsikkotietoa. Lisäksi socket-yhteydessä asetetaan TCP_NODELAY-optio, jolla estetään lähetettävän tiedon puskurointi isommaksi kokonaisuudeksi ennen lähetystä.

RFC 1006 on protokolla, joka mahdollistaa OSI-sovellusten toimimisen TCP/IP-ympäristössä. Se tarjoaa OSI-siirtoprotokollan luokan 0 (TP0) palvelut. RFC 1006 poistaa TCP:n ja OSI CONS (Connection-mode Network Service) -protokollan välisen eron lisäämällä TCP:n tavuvirtaan protokollaotsikot (TPKT header), jolloin lähetetty TCP-tieto muuttuu OSI CONS:n ymmärtämiksi NSDU (Network Service Data Unit) -paketeiksi. OSI NSAP (Network Service Access Point) -osoitteet korvataan IP-osoitteella ja TCP/IP-porttinumerolla. Kuva 31 esittää RFC 1006:n sijainnin protokollapinossa.



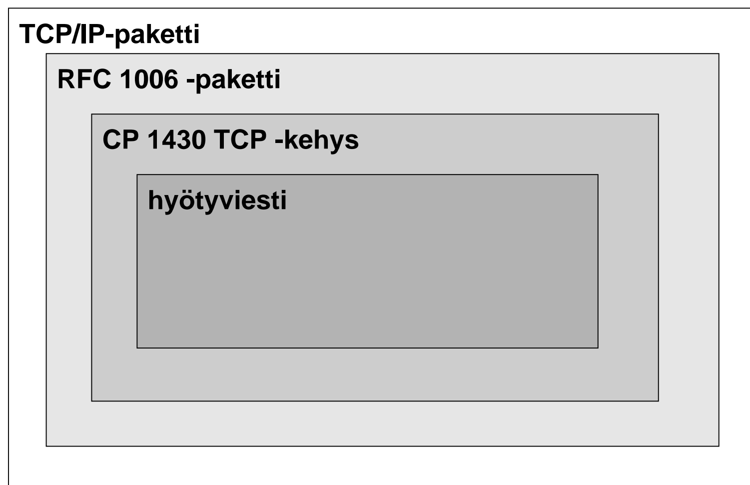
Kuva 31. RFC 1006:n sijainti protokollapinossa.

Kuva 32 esittää RFC 1006:n käyttötavan esimerkkisovelluksessa. Toteutustasolla RFC 1006 näkyy TCP/IP-pakettiin lisättävänä otsikkotietona. Yhteyden muodostamiseen käytettyjen pakettien kentissä määritellään käytettävä protokollaversio, kommunikointipaketin pituus, paketin tyyppi, kommunikoinnin osapuolet ja käytettävän viestipaketin koko. Tiedonsiirtoon käytettävän paketin kentissä määritellään käytettävä protokollaversio, tietopaketin kokonaispituus, paketin tyyppi ja siirrettävä tieto.



Kuva 32. RFC 1006:n käyttö esimerkkisovelluksessa.

Esimerkkisovelluksessa sovittimen ja CP 1430 TCP -kortin väliselle kommunikoinnille on määritelty kiinteänpituiset viestit. Viestityypistä riippuen vain osa viestin varaamasta muistista käytetään hyväksi. Tällä tavoin CP 1430 TCP -kortin konfigurointi saadaan mahdollisimman yksinkertaiseksi. Kuva 33 esittää sovittimen ja logiikkakortin välillä lähetettävän tietopaketin rakenteen. CP 1430 TCP -kehyksen käyttö ei ole tarpeen esimerkkisovelluksessa toteutetulla viestinvälityksellä, mutta muilla kortin konfiguroinneilla sitä voidaan tarvita.



Kuva 33. Sovittimen ja logiikan välillä käytettävä tietopaketti.

5.9.1 Socket-yhteyden käyttö MFC-luokkakirjastolla

Esimerkkisovelluksessa käytetty socket-yhteys logiikan ja Windows NT -tason välille on toteutettu käyttämällä MFC-luokkakirjaston valmista *CSocket*-luokkaa. Ohjelmoija määrittelee luokan, joka perii *Csocket*-luokan ominaisuudet. Ohjelmoijan määrittelemässä luokassa ylikirjoitetaan metodit, joita sovelluskehys kutsuu socketiin kohdistuvien tapahtumien yhteydessä.

Tapahtumia, joille käsittelijämetodit kirjoitetaan, ovat esimerkiksi yhteydenotto, tiedon vastaanotto ja yhteyden sulkeminen. Yhteydenoton hyväksymiseen ohjelmoija ylikirjoittaa *CSocket*-luokalta perityn metodin *OnAccept()*, jota kutsutaan automaattisesti toisen osapuolen ottaessa yhteyttä socketin kuuntelemaan porttiin. Saapuneen tiedon käsittelyyn ohjelmoija ylikirjoittaa metodin *OnReceive()*, jota kutsutaan automaattisesti tiedon saapuessa. Yhteyden sulkemisen käsittelyyn ohjelmoija ylikirjoittaa metodin *OnClose()*, jota kutsutaan automaattisesti yhteyden katketessa jostain syystä.

Kuva 34 esittää, miten kaksi sovellusta kommunikoi keskenään MFC-luokkakirjaston socket-luokkia käyttäen. Kommunikoinnin vaiheet ovat seuraavat:

1. Palvelin luo tiettyä porttia kuuntelevan socketin.

2. Asiakas ottaa yhteyden palvelimen kuuntelemaan porttiin.

3. Sovelluskehys kutsuu kuuntelevan socketin *OnAccept()*-metodia.

4. *OnAccept()*-metodin toteutuksessa hyväksytään yhteys asiakkaaseen. Accept()-metodille annetaan parametrina toinen socket, jonka kautta tiedon välitys tapahtuu. Näin palvelinsocket voi jatkaa yhteydenottojen kuuntelemista.

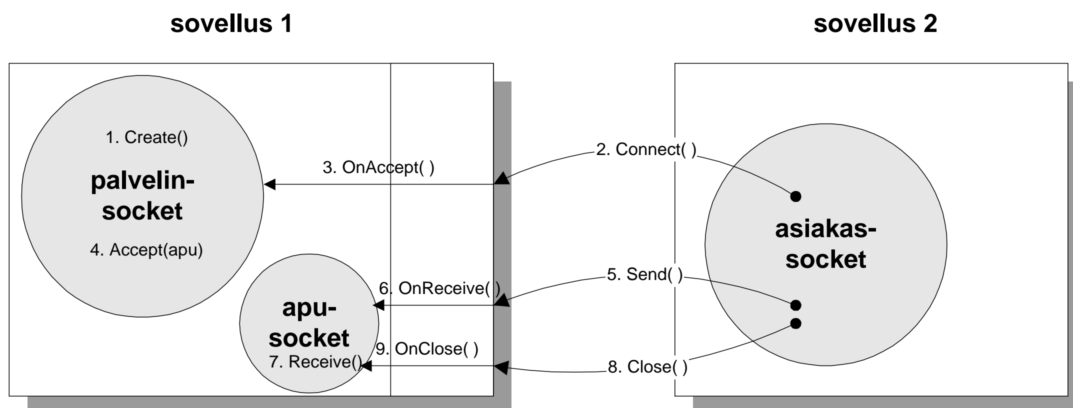
5. Asiakas lähettää tietoa palvelimelle.

6. Sovelluskehys kutsuu *OnReceive()*-metodia.

7. *OnReceive()*-metodin toteutuksessa luetaan tieto.

8. Asiakas sulkee yhteyden.

9. Sovelluskehys kutsuu *OnClose()*-metodia, jossa suoritetaan tarvittavat toimenpiteet.

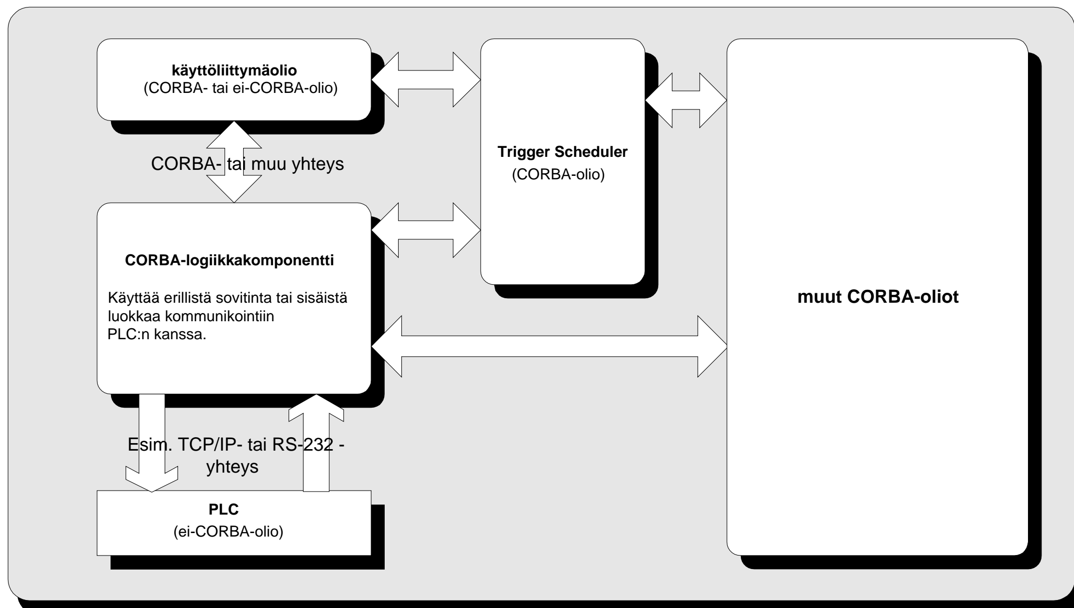


Kuva 34. Socket-yhteys MFC-luokkakirjaston CSocket-luokilla.

5.10 Laitteistoliitynnät

Liitynnät CORBA-olioiden ja muiden kuin CORBA-olioiden välillä tulee toteuttaa muita kommunikointimekanismeja kuin CORBAa käyttäen. Esimerkiksi käyttöjärjestelmän tarjoama viestinvälitys, TCP/IP-socketit tai RS-232, ovat yleisesti käytettyjä mekanismeja. Eri mekanismien vaatimat liitynnät kannattaa toteuttaa omina luokkinaan, jotka on helppo vaihtaa lähdekooditasolla kommunikointimekanismin muuttuessa. Esimerkiksi jokaista PLC:tä ohjaava CORBA-olio sisältää erityisen PLC-liityntäluokan instanssin, jonka metodeissa käytetään kullekin logiikalle parhaiten soveltuvaa siirtoprotokollaa. Kuva 35 esittää kohdelaitteen mahdollista liityntätapaa muuhun järjestelmään. Toinen vaihtoehto on käyttää erityisiä sovittimia, jotka toimivat viestinvälittäjinä CORBA-komponenttien ja muiden komponenttien välillä. Tällöin liityntäluokan instanssi toteutetaan sovittimessa, jolloin muut järjestelmän komponentit

voivat kommunikoida pelkästään CORBA-väylää pitkin. Sovittimia käsitellään tarkemmin kohdassa 5.11.



Kuva 35. Laiteliityntä CORBA-ympäristöön.

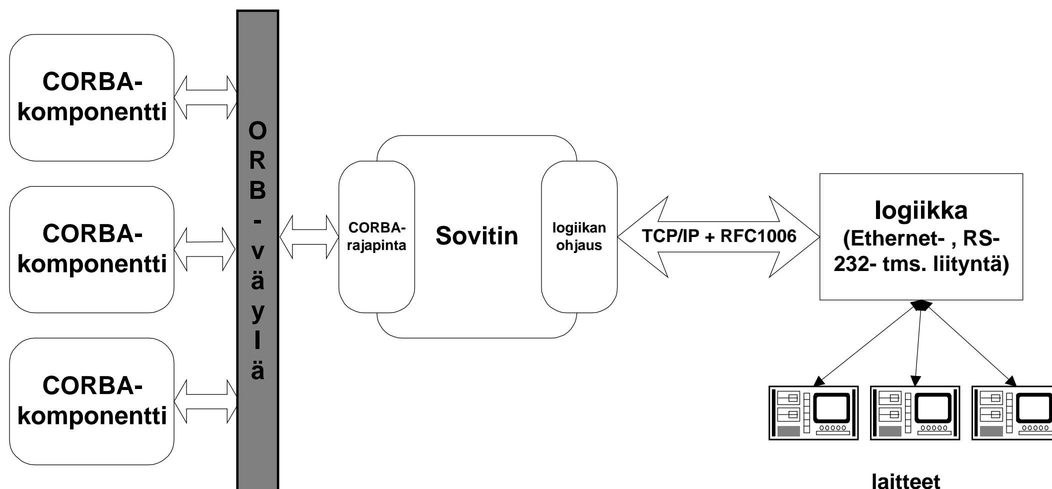
Mikäli logiikka on liitetty järjestelmään TCP/IP-yhteydellä, sovellukset voivat sovitin mukaan lukien sijaita fyysisesti missä koneessa tahansa. Sovitin ottaa yhteyden logiikan IP-osoitteeseen ja sen kuuntelemaan porttiin, ja yhteys tiedon välittämiseen on luotu. Mitään paikkariippuvaa koodia ei tarvitse toteuttaa CORBA-komponentteihin viestien välittämiseksi laitetasolle, sillä IP-osoite ja porttinumero voidaan antaa ohjelmalle esimerkiksi funktiokutsun parametreina.

Jos logiikan liityntä on toteutettu RS-232-protokollaa käyttäen, logiikalle viestejä lähettävän sovittimen täytyy sijaita logiikan kanssa samassa koneessa. Paikkariippumattomuuden taso on hieman alhaisempi kuin TCP/IP-yhteyden tapauksessa, mutta sijainti on edelleen läpinäkyvä muille kuin sovittimelle.

Erilaiset kenttäväyläratkaisut järjestelmän osien väliseen kommunikointiin ovat yleisiä FM-järjestelmissä. PC voidaan liittää kenttäväylään joko omalla kenttäväyläkortilla tai sarjaportin ja kenttäväylän väliin liitettävällä sovitinkappaleella. Paikkariippuvuuden kannalta kenttäväyläratkaisu on TCP/IP-liitynnän ja RS-232-liitynnän välimuoto: sovittimen tulee sijaita jossakin kenttäväyläliitynnän tarjoavassa koneessa. Jos logiikka toimii kenttäväylän yhtenä solmuna, sovitin ei ole sidottu mihinkään tiettyyn kenttäväylän solmuun.

5.11 Sovittimet

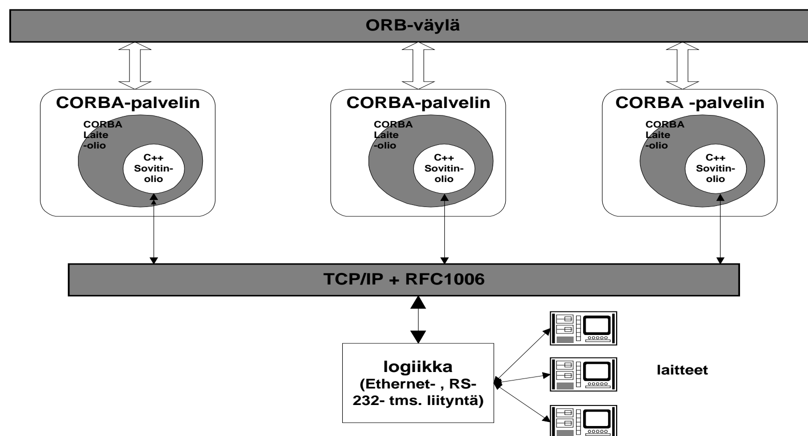
Windows NT -tason ohjelmiston liittäminen järjestelmää ohjaaviin logiikoihin on toteutettu esimerkkisovelluksessa erillisten sovittimien avulla. Sovitin on prosessi, joka toteuttaa CORBA-rajapinnan ja muuntaa CORBA-viestit logiikan ymmärtämään muotoon ja päinvastoin. Esimerkkisovelluksessa tämä tarkoittaa CORBA-viestien ja RFC 1006 -protokollan mukaisten TCP/IP-viestien välistä muuntamista. Kuva 36 esittää sovittimen roolia järjestelmässä.



Kuva 36. Sovittimen rooli järjestelmässä.

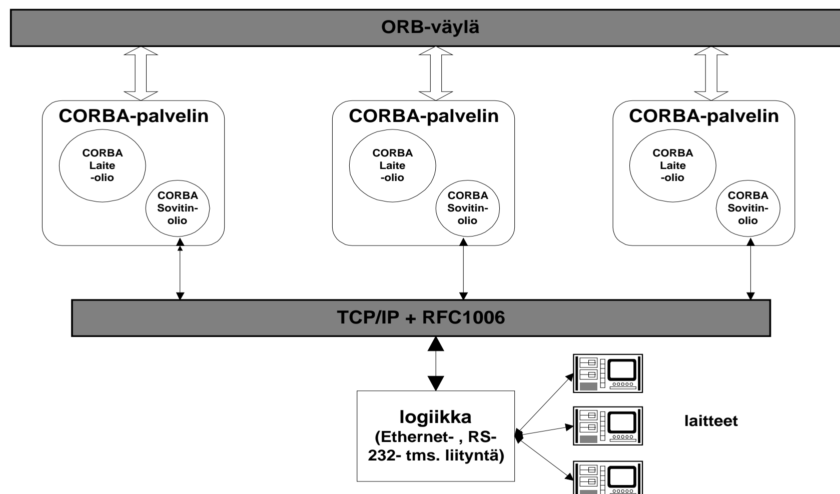
Esimerkkijärjestelmän pienestä koosta johtuen sovitin toteutettiin yhtenä komponenttina, jonka kautta kaikki laitetasolle lähetettävät ja sieltä vastaanotettavat viestit välitettiin. Näin laitteita ohjaavaan logiikkaan tarvittiin vain yksi TCP/IP-yhteys. Sovitin toteutettiin omana prosessinaan toiminnan testaamisen helpottamiseksi.

Sovittimen voi toteuttaa myös integroituna C++-luokkana CORBA-komponentteihin, kuten kuten Laite- tai Hissikone-luokkaan. Se edellyttää käytettävän protokollan toteuttavan sovitinluokan ja -koodin mukaan ottamista jokaiseen järjestelmää ohjaavaan sovellukseen. Siten käytettävän protokollan muuttaminen vaatii muutoksia suurempaan osaan ohjelmistoa. Etuna on yhden CORBA-prosessin poistuminen järjestelmästä ja siten toiminnan nopeutuminen ja resurssitarpeiden vähentyminen, mutta sovittimien ja logiikkakortin välille tarvittavien socket-yhteyksien määrä kasvaa. Kuva 37 esittää tätä järjestelyä.



Kuva 37. Sovittimen integrointi sovellusolion sisälle.

Kompromissiratkaisuna on menettely, jossa sovittimelle määritellään IDL-rajapinta, mutta esimerkiksi Laite-olion tarjoavaan palvelimeen luodaan instanssi myös määritellystä sovitinluokasta. Kun Laite-olio käyttää sovitinta samassa muistiavaruudessa, kutsut voivat olla tavallisia C++-virtuaalimetodikutsuja CORBA-kutsujen sijaan. Ohjelmiston kehityksessä voidaan edelleen käyttää IDL-rajapintamäärityksiä ja yhtenäistä CORBA-kutsumekanismia kommunikoivien olioiden välillä. Haittana on sovittimien ja logiikkakortin välille tarvittavien socket-yhteyksien määrän kasvaminen sekä logiikkakortin konfiguroinnin monimutkaistuminen. Kuva 38 esittää tätä järjestelyä.



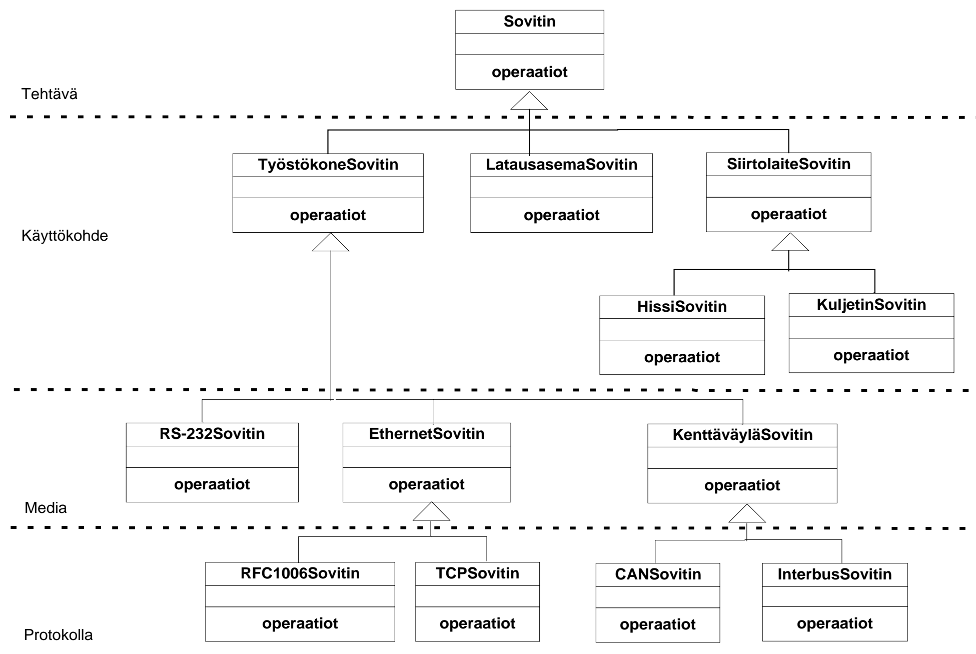
Kuva 38. Palvelinkohtainen sovitin.

Sovittimet voidaan luokitella niiden kohdelaitteen ja käytettävän protokollan mukaan. Esimerkiksi työstökoneilla voi olla omat sovittimensa, aliluokitettuna käytettävän protokollan ja siirtotien mukaan. Toinen vaihtoehto on laittaa monen kohdelaitteen

(esim. työstökoneen ja hissin) ohjaamiseen tarvittavat toiminnot yhteen sovittimeen, kuten esimerkkisovelluksessa on tehty. Tällöin yksi sovitinkomponentti välittää kaikki järjestelmän viestit Windows NT -tasolta logiikoille ja sieltä laitetasolle. Kuva 39 esittää yhtä mahdollista sovitinluokitusta.

Sovittimia aliluokitettaessa niiden CORBA-rajapinnat pysyvät samana, jos kohdelaitteiden loogiset operaatiot säilyvät samoina. C++-toteutusten aliluokissa ylikirjoitetaan kantaluokan loogiset operaatiot ja näin annetaan operaatioille uusi merkitys. Esimerkiksi kantaluokassa voidaan määritellä looginen operaatio "*siirraAlusta()*", joka sarjaportin kautta ohjattavassa järjestelmässä toteutetaan sovittimessa RS-232-kirjastokutsuilla ja TCP/IP-yhteydellä ohjattavassa järjestelmässä omana socket-luokkanaan.

IDL-tasolla operaatioiden ylikuormitus ei onnistu, mutta perintä ja uusien operaatioiden määrittäminen johdetuille rajapinnoille on mahdollista. C++-metodeiksi muunnettujen rajapintaoperaatioiden ylikuormittaminen tapahtuu aina C++-tasolla. Käytännössä usein vain perintäpuun lehtisolmut ovat toimivia komponentteja, muut ovat abstrakteja apuluokkia. Kuva 39 esittää tilannetta, jossa perintäpuun luokalla TyöstökoneSovitin ei ole käyttöä, ellei sille ole tehty toteutuskoodia, kuten RFC1006Sovitin-luokalle.



Kuva 39. Sovittimen mahdollinen luokkarakenne.

Sovitinkomponentit ovat monistettavissa helposti. Jos järjestelmän kuormitus vaatii, samasta sovitinluokasta voidaan luoda useampi instanssi, joiden looginen nimi erottaa ne toisistaan. Tämän jälkeen muilta järjestelmän komponenteilta tulevat viestit ohjataan

joko satunnaisesti tai jonkin algoritmin mukaan jollekin järjestelmän sovittimista. Useampi sovitin voi välittää viestejä samalle logiikalle, tai sovittimet voivat ohjata ja seurata eri logiikkaliityntöjä

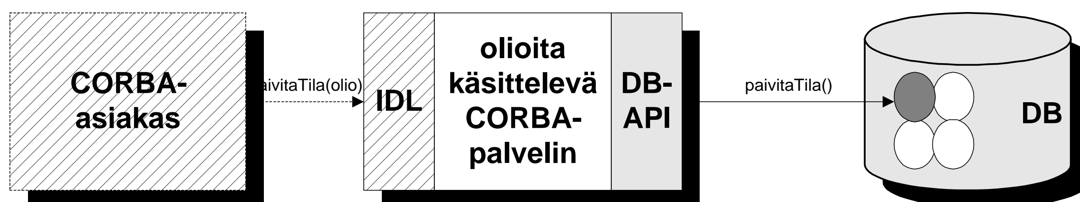
Sovitinkomponentit ovat paikkariippumattomia käytettäessä logiikkaliityntänä TCP/IP-korttia. Jos logiikkaa ohjataan sarjaportin tai PC:n kenttäväyläkortin kautta, sovitinkomponentin tulee sijaita kohdelogiikkaa ohjaavassa PC:ssä. Sovittimen sijainti on silti edelleen läpinäkyvä kaikille sovittimelle viestejä lähettäville CORBA-komponenteille: pelkkä tieto sovittimen loogisesta nimestä riittää asiakaskomponenteille.

5.12 Sovelluksen ja tietokannan integrointi

Sovellusolioiden tila tulee saada tietokantaan virhetilanteiden ja uudelleenkäynnistysten varalta. Se, kuinka olion tila talletetaan kantaan ja luetaan kannasta, riippuu olion käyttötarpeesta. Seuraavassa pohditaan tietokannan käyttötapaa muutamassa esimerkkitilanteessa.

1) Ei CORBA-olio

Jos olio ei ole CORBA-olio, sitä voidaan käsitellä suoraviivaisesti tietokantaoliona tietokannan tarjoamien API-kutsujen kautta. Esimerkiksi FM-järjestelmän työkalujen käsittelyyn voidaan tehdä sovellus, joka käyttää työkaluolioita tällä tavalla. Jos CORBA-sovellusten pitää pystyä käsittelemään tietokannan olioiden tilaa CORBA-väylän kautta, olioita käsittelevästä prosessista voidaan tehdä CORBA-rajapinnan tarjoava palvelin. Kuva 40 esittää tätä mekanismia.



Kuva 40. Sovellusolioiden käsittely muuna kuin CORBAolioina.

2) CORBA-olio

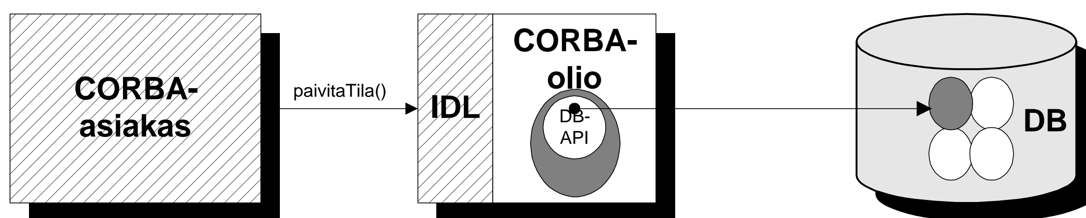
Jos olio on CORBA-olio, seuraavat kaksi tapaa ovat mahdollisia:

2.1) CORBA-olio peilataan tietokantaan sisäisen tietokantaolion avulla.

CORBA-olion luokkamäärittelyssä annetaan luokalle sisäinen tietojäsen, joka on esimerkiksi Objectivityn tietokantaa käytettäessä tyyppiä *ooHandle(luokanNimi)*. CORBA-oliota luotaessa sen luokan muodostimessa luetaan annetun tunnisteen perusteella tietokannasta olio, jos sellainen on olemassa. Tietokantaolio on siten CORBA-olion viitattavissa tietojäsenensä kautta.

Kun jokin CORBA-asiakas haluaa lukea olion tilan, se välittää kutsun haluamalleen CORBA-kohdeoliolle. Kohdeolio puolestaan lukee sisäisen tietokantaolion viittauksen avulla tilan tietokannasta ja palauttaa sen CORBA-väylässä asiakkaalle. Toteutus ei merkittävästi hidasta CORBA-kutsun palvelemista, koska tietokantaan ei kohdistu päivitysoperaatiota.

Jos CORBA-asiakas haluaa päivittää kohdeolion tilan, kohdeolio joutuu käyttämään tietokantaa päivitysmoodissa, jolloin operaation kestoaika kasvaa huomattavasti. Kuva 41 esittää tätä mekanismia.



Kuva 41. CORBA-olion tilan päivitys tietokantaan sisäisen muuttujan avulla.

Ongelmana tässä menettelytavassa on järjestelmän luokkamääritelmien lisääntyminen: CORBA-olion luokan sisäiselle tietojäsenelle pitää määritellä luokka, jonka tietokanta ymmärtää ja jossa on samat jäsenet kuin sen edustamassa CORBA-luokassa. Tämä tietokantaluokka voidaan luoda melko suoraviivaisesti CORBA-luokan määritelmästä, mutta lisäongelmia aiheuttaa CORBA-spesifisten tietotyyppien muuntaminen tietokannan ymmärtämään muotoon. Olioviittauksen tallettamista helpottaa metodi *object_to_string()*, joka muuntaa olioviittauksen merkkijonomuotoon, mutta esimerkiksi sekvenssien ja Any-tyypin muuntamiseen ohjelmoijan pitää kehittää oma mekanisminsa.

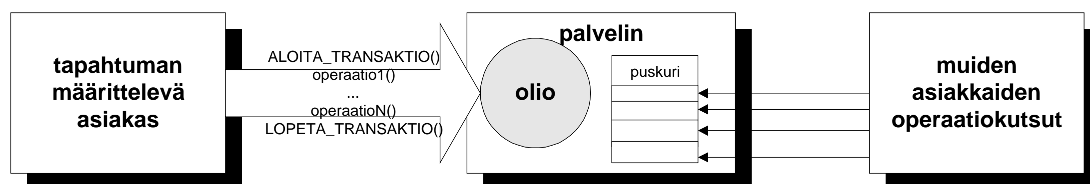
2.2) CORBA-olio peilataan tietokantaan lähettämällä CORBA-oliosta tapahtuma, esimerkiksi yksisuuntainen CORBA-kutsu, ja antamalla jonkin toisen komponentin tehdä tietokantapäivitys. Tällöin päivitetyn olion toiminta ja sen vaste asiakkaalle nopeutuvat, mutta ylimääräinen osapuoli aiheuttaa lisää viestiliikennettä. Päivitetty CORBA-olio ei voi tietää päivityksen onnistumisesta, ellei tietokannan päivittäjä vastaa lähetettyyn viestiin kertomalla operaation mahdollisesta epäonnistumisesta. Tämä edellyttää taphtumien kirjanpitoa CORBA-oliossa, jotta tietokantapalvelimelta tulleet

kuittaukset osataan käsitellä oikein. Tämä mekanismi ei sovellu tehtäväkriittisiin järjestelmiin, koska se on epävarma ja aiheuttaa rinnakkaisuuden hallintaongelmia.

5.12.1 Tapahtumien hallinta

Operaatioiden jakamattomuus on monissa sovelluksissa välttämätön ominaisuus. Asiakas voi haluta, että sen määrittelemät operaatiot toteutetaan kokonaisuutena: jos yksikin epäonnistuu, kaikkien operaatioiden vaikutukset tulee peruuttaa. Tavallisesti tämä hoidetaan tietokannan tapahtumien (transaction) avulla, mutta asiakkaan käsitellessä CORBA-oliota tietokantaolion sijaan asia monimutkaistuu. Kun asiakas haluaa kohdistaa CORBA-olioon luku- ja kirjoitusoperaation peräkkäin siten, ettei kukaan muu osapuoli pääse väliin muuttamaan tapahtuman kannalta olennaisia tietoja, jonkinlainen lukitusmekanismi myös CORBA-tasolla on välttämätön. Tämä voidaan toteuttaa ostamalla jokin valmis palvelu, kuten esimerkiksi Orbixin OTS, tai toteuttamalla lukitusmekanismi itse.

Mekanismin toteuttaminen itse voi olla työlästä riippuen tarvittavista ominaisuuksista. Asiakkaan määrittelemät tapahtuman rajat kertovat palvelimelle, milloin asiakas on saanut palvelun kaikkiin tapahtumansa operaatiokutsuihin. Tämän perusteella palvelin voi estää muiden asiakkaiden operaatiokutsujen limittymisen ensimmäisen asiakkaan tapahtumaan. Palvelimen tulee puskuroida kaikki estetyt kutsut, jotta ne voidaan suorittaa, kun ensimmäinen asiakas on vapauttanut lukot. Ellei muita operaatiokutsuja puskuroida, esimerkiksi operaation palautusarvo joudutaan käyttämään operaation onnistumisen tai epäonnistumisen ilmoittamiseen. Näin estettyjen asiakkaiden on yritettävä kutsuja uudelleen, kunnes ne voidaan suorittaa. Kuva 42 esittää tilannetta, jossa yksi asiakas on aloittanut useamman operaation sisältävän tapahtuman ja jossa muilta asiakkailta tulevat operaatiokutsut puskuroidaan odottamaan kohdeolion tai palvelimen vapautumista.

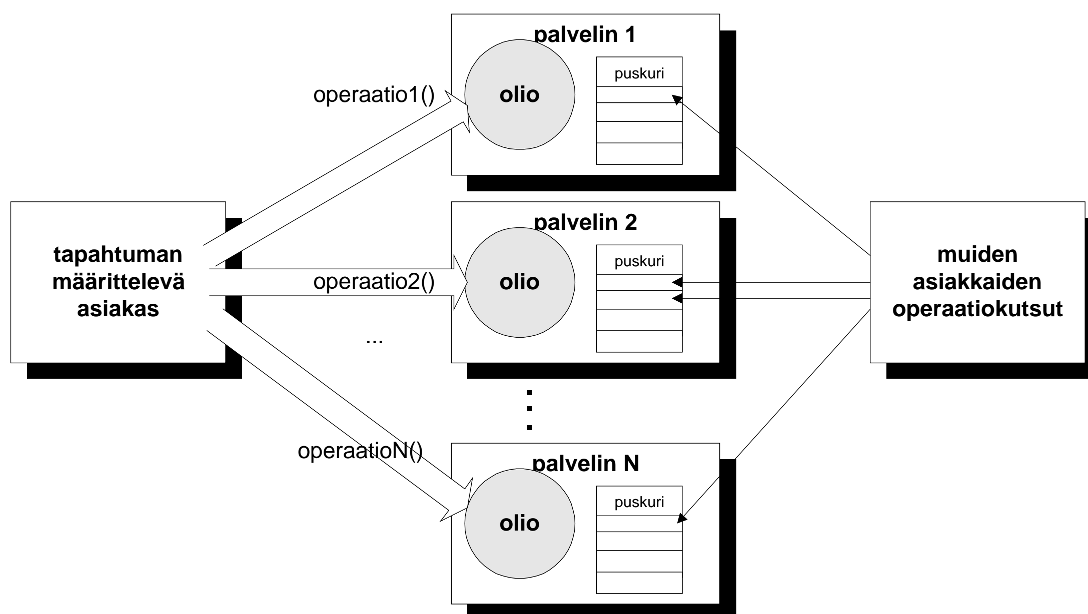


Kuva 42. Palveltava tapahtuma ja sen takia estetyt operaatiokutsut.

Estettyjen kutsujen puskuroimisen lisäksi on toteutettava mekanismi, joka palvelimen vapautuessa aloittaa puskuroitujen viestien käsittelyn. Jos estettyjen kutsujen lähettämät asiakkaat jäävät odottamaan kutsunsa suoritusta, kullakin asiakkaalla voi olla vain yksi operaatiokutsu kerrallaan puskuroituna. Mikäli asiakkaat ovat lähettäneet viestinsä

asynkronisesti ilman palvelun toteutumisen odottamista, jokaisella asiakkaalla voi olla monta kutsua puskuroituna eikä niiden saapumisjärjestys palvelimelle välttämättä ole sama kuin asiakkaan lähetysjärjestys.

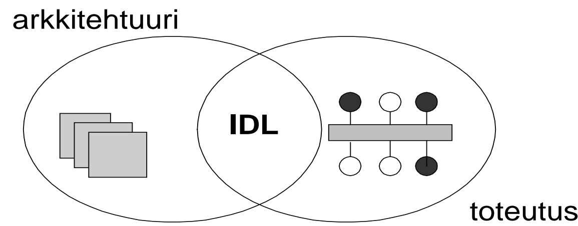
Edellä kuvatussa tilanteessa tapahtuman ohjaus on toteutettavissa palvelimessa tai kohdeoliossa lukuun ottamatta asiakkaan määrittelemiä tapahtuman rajoja. Jos asiakkaan määrittelemä tapahtuma sisältää kutsuja usealle eri palvelinprosessissa sijaitsevalle kohdeoliolle, tapahtuma on huomattavasti vaikeampi hallita. Tällöin tarvitaan hajautettujen tapahtumien hallintaa, jossa tapahtuman ohjaajan on oltava selvillä kaikista aktiivisista tapahtumista. Kuva 43 esittää hajautettua tapahtumaa.



Kuva 43. Hajautettu tapahtuma.

5.13 CORBA-järjestelmien toteutusperiaatteita

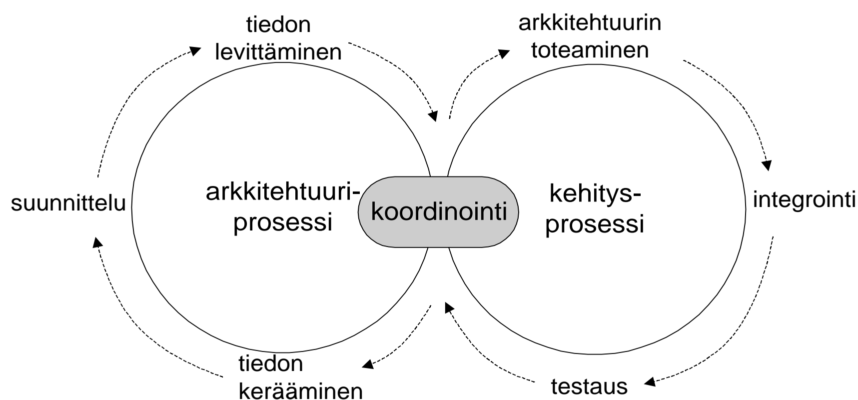
Mowbrayn mukaan CORBA-järjestelmien kehittämisen tulisi muodostua kahdesta rinnakkaisesta prosessista, arkkitehtuurin määrittämisprosessista ja järjestelmän kehitysprosessista. IDL-kuvauskielellä määritellyt rajapinnat ovat osa sekä järjestelmän arkkitehtuuria että kehitystyön tuloksena saatavaa toteutusta. Kuva 44 esittää ILD-kuvauskielen roolia CORBA-järjestelmässä. [7]



Kuva 44. IDL:n rooli CORBA-järjestelmässä.

Kuva 45 esittää arkkitehtuuri- ja kehitysprosessin vaiheita järjestelmän kehittämisessä. Arkkitehtuuriprosessi alkaa tiedon keräämisellä, minkä tarkoituksena on sovellusalueen ongelman ymmärtäminen ja siten riskien vähentäminen. Seuraavassa vaiheessa suunnittelu tuottaa arkkitehtuurin kannalta tärkeät abstraktiot, joista ohjelmiston rajapintakuvaukset johdetaan. Suunnittelun tulokset ja tavoitteet jaetaan muille projektin jäsenille kokousten ja keskustelujen muodossa. [7]

Kehitysprosessi keskittyy alijärjestelmien luomiseen ja niiden integroimiseen. Se alkaa arkkitehtuurin omaksumisella, jotta toteutus noudattaisi riittävästi suunnittelua. Arkkitehtuurin omaksumisen ja alijärjestelmien toteuttamisen jälkeen alijärjestelmien integrointi suunnitellaan ja toteutetaan. Testausvaiheessa tarkistetaan sekä integroinnin että suunnittelun onnistuminen. [7]



Kuva 45. Arkkitehtuuri- ja kehitysprosessit.

Kehityskierroksen jälkeen tarvitaan prosessien välistä koordinointia. Koordinoinnin avulla arkkitehdit ymmärtävät paremmin suunnittelun seuraukset ja voivat hienosäätää arkkitehtuuria. Mahdollisten muutosten jälkeen prosesseja toistetaan, kunnes riittävän kypsä arkkitehtuurimalli on saavutettu. [7]

Suunniteltaessa ja toteutettaessa CORBA-sovelluksia jollakin ORB-tuotteella seuraavat asiat ovat huomioimisen arvoisia [19]:

- Mitkä oliot toteutetaan CORBA-olioina ja mitkä muina olioina, esimerkiksi muistinvaraisina C++-olioina tai tietokantaolioina? CORBA-oliot tarvitsevat enemmän keskusmuistia ja niiden käsittely on raskaampaa kuin tavallisten C++-olioiden. Usein vain suuremmat ohjelmistokomponentit kannattaa toteuttaa CORBA-olioina.

- Montako oliota voidaan toteuttaa palvelinta kohden? Palvelinsovelluksen keskusmuistin käyttö nousee vain vähän sen ylläpitämien olioiden lukumäärän kasvaessa. Vasteaikojen salliessa kannattaa toteuttaa monta oliota samaan palvelimeen, ellei olioiden tule olla toisistaan riippumattomia toimintavarmuuden kasvattamiseksi.

- Olioiden välisten assosiaatioiden hallinta? Olioviittausten yhtenäisyyden ylläpito CORBA-olioissa on vaivalloista verrattuna esimerkiksi tietokannan osittain automatisoituun mekanismiin.

- CORBA-olioiden tietokantaan talletus? Tietokannan ja sovelluksen integrointi hidastaa järjestelmän toimintaa. Tietokantaan peilatun CORBA-olion tilan lukeminen ei aiheuta CORBA-kutsuun merkittävää viivettä, mutta tilan päivittäminen CORBA-olion kautta tietokantaan on selvästi raskaampi operaatio.

- Suurissa järjestelmissä kannattaa usein tehdä CORBA-palvelin suurelle joukolle tavallisia C++- tai tietokantaolioita. Olioita käytetään palvelimen rajapinnan kautta määrittämällä olion tunniste.

- Suurissa järjestelmissä tarvitaan roskien keruuta. Oliot, jotka eivät ole aktiivisesti käytössä, tulee poistaa tuhlaamasta keskusmuistia. Tämän toteuttamiseen on useita tapoja. Asiakas voi vapauttaa käyttämänsä olion, kun ei enää tarvitse sitä. Vaihtoehtoisesti palvelin voi vapauttaa olion käyttämän keskusmuistin tietyn ajan jälkeen automaattisesti, jos oliota ei ole käytetty. Palvelin voi myös vapauttaa olion varaamat resurssit huomatessaan yhteyden asiakkaaseen katkenneen.

- Suuret järjestelmät vaativat ylläpito- ja diagnostiikkatyökaluja - ellei niitä ole valmiiksi myytävänä, ne pitää toteuttaa itse.

- Tapahtumien (transactions) hallinta on tärkeä ominaisuus: joskus kaikki tapahtumaan kuuluvat operaatiot tulee suorittaa onnistuneesti. Mikäli yksikin operaatio epäonnistuu, kaikkien muidenkin tapahtuman operaatioiden vaikutukset tulee palauttaa. Kannattaa selvittää, miten kaupalliset tapahtumapalvelut ovat integroitavissa muihin järjestelmän tuotteisiin (ORB, muut palvelut, tietokanta tms.), ellei tapahtumapalvelua päätetä toteuttaa itse.

- Suurten tietomäärien välittäminen on usein järkevää tehdä esimerkiksi tietokannan tai jaetun tiedostojärjestelmän avulla.

Lukkiutumien välttäminen vaatii huolellista suunnittelua - jos olio kutsuu toista oliota ja kutsuttu olio puolestaan tarvitsee kutsuneen olion palvelua, ollaan lukkiutumatilanteessa. Lukkiutumia voidaan yrittää välttää esimerkiksi yksisuuntaisilla kutsuilla tai säikeistämällä sovellus.

CORBAn valitseminen hajautusteknologiaksi ja arkkitehtuurimalliksi ei tarkoita sitä, että kaikki olemassa olevat ohjelmistot pitäisi toteuttaa uudelleen ORB-tuotteen avulla. Viisaampaa on aloittaa CORBAn soveltaminen CORBA-pakkaaja-luokkien avulla, jolloin jo olemassa oleva koodi “kääritään” CORBA-rajapinnan taakse [6, 20]. Rajapinnan avulla tarjotaan hajautettu pääsy vanhan koodin toteuttamaan toiminnallisuuteen. Tämä menettelytapa mahdollistaa CORBAn käyttöönoton asteittain, jolloin ORB-toteutuksen mahdollisen virheellisyyden ja tietämyksen puutteen merkitys saadaan pienemmäksi.

5.14 Jatkokehitys

Toteutettu esimerkkisovellus on prototyyppi, joka ei ole toiminnoiltaan täydellinen eikä esimerkiksi suorituskyvyn osalta loppuun asti optimoitu. Tässä kohdassa esitetään ajatuksia siitä, mitä ohjelmiston ominaisuuksia tulisi toteuttaa lisää tai paremmin.

5.14.1 Suorituskyvyn optimointi

CORBA-kutsujen aiheuttamaa viivettä voidaan vähentää vähentämällä kutsuttavien operaatioiden määrää. Lukuoperaatioiden palautusarvojen tietomäärän lisääntyminen ei aiheuta merkittävää viivettä kutsuihin (kohta 4.1.10). Siksi yksittäisellä operaatiolla kannattaa siirtää mahdollisimman paljon tietoa, ellei ohjelman selkeys siitä kärsi.

Yhteyden muodostaminen kohdeolioon *_bind()*-metodia käyttäen on huomattavasti hitaampaa kuin operaatiokutsun suorittaminen. Yhteydet kannattaa muodostaa mahdollisuuksien mukaan sovelluksen alustusrutiineissa, esimerkiksi luokkien muodostimissa.

Tietokannan käsittely kannattaa pitää mahdollisimman pitkälle muistinvaraisena, sillä tietokanta on tyypillisesti järjestelmän hitain komponentti. Järjestelmän olioiden tilan tietokantaan peilaaminen tulisi saada ohjelmoijalle mahdollisimman läpinäkyväksi.

Ideaalitilanteessa tietokantaan peilattava CORBA-olio yksinkertaisesti perii pysyvyysominaisuudet toiselta luokalta.

Joustavuudestaan huolimatta ECA-konsepti vähintään kaksinkertaistaa välitettävien viestien lukumäärän. Aikakriittiset kutsut tulisi toteuttaa suoraan kommunikoivien osapuolten välillä, tai lähetettäviin tapahtumiin tulisi saada tilatietoa mukaan. Tilatietoa sisältävän geneerisen tapahtumaviestin määrittely on kuitenkin vaikeaa, koska komponenteilla on vaihteleva määrä eri tietotyypeillä toteutettavia tila-attribuutteja. IDL-tietotyyppi *Any* helpottaa geneeristen viestien toteuttamista.

5.14.2 Tietokannan ja tapahtumien hallinta

Tietokannan integrointi järjestelmään on välttämätöntä. Tietoa ei saa kadota virhetilanteissa. Myös tapahtumien hallinta on tärkeää tiedon yhtenäisyyden takaamiseksi. Tietokannan avulla nämä kaksi olennaista palvelua voidaan toteuttaa yhtenä palveluna, mutta se ei ole helppoa. Esimerkkisovelluksen yhteydessä kokeiltiin näitä molempia mekanismeja tarjoavaa kaupallista toteutusta. Objectivity Inc. aikoi toimittaa Windows NT -käyttöjärjestelmään Orbix-Objectivity-integrointipaketin beta-version, mutta toimitti aikataulusta myöhässä Solaris-käyttöjärjestelmään toteutetun version lähdekoodit. Ohjelmiston siirtäminen Windows NT -käyttöjärjestelmään ei onnistunut kohtuullisella vaivalla.

5.14.3 Ohjelmiston komponentointi

Järjestelmän muunnelmien ja luonteen perinpohjainen tunteminen antaa edellytykset komponentoida ohjelmiston osat. Hyvin toteutettuna sekä CORBA-olioiden että MFC-perustaisen käyttöliittymän omien OCX-ohjainten toiminnallisuus saadaan uudelleenkäytettävään muotoon. Komponentointi vaatii sovellusalueen perinpohjaisen tuntemuksen lisäksi vahvaa kokemusta käytettävistä työkaluista ja ohjelmointikielestä.

CORBA-komponentoinnin onnistumisen tärkein edellytys on stabiilin IDL-rajapintahierarkian suunnittelu. Jotta perintämekanismin avulla saavutettaisiin merkittävää hyötyä uudelleenkäytön kannalta, tulee ohjelmistokomponenttien yhteiset ja poikkeavat ominaisuudet määrittää huolellisesti. Määrityksen perusteella voidaan toteuttaa sekä IDL-rajapintahierarkia että toteutuskielen komponentointi, kuten esimerkiksi funktiokirjasto tai C++-luokkamalli.

Suunnittelussa tulee ennakoida mahdollisimman pitkälle myös tulevia muunnelmia. Hienojakoinen rajapintahierarkia helpottaa uusien muunnelmien tuomista järjestelmään,

mutta alentaa järjestelmän ymmärrettävyyttä. Kokemus on tässä asiassa paras opettaja, mutta CORBAn yhtä tärkeimmistä tavoitteista tulee noudattaa: rajapintamääritelmä ja toteutus tulee säilyttää erillään väistämättömän muunnostyön helpottamiseksi.

6. TULOSTEN ARVIOINTI

Tässä luvussa arvioidaan esimerkkisovelluksen toteuttamisessa kertyneen kokemuksen pohjalta CORBAn ja Orbixin soveltumista FM-järjestelmien ohjausohjelmiston toteuttamiseen. Arviointi on tehty kohdissa 2.3. ja 2.4 esitettyjen ominaisuuksien ja vaatimusten pohjalta.

6.1 CORBAn vaatima koulutustarve

C++-ohjelmoinnin osaaminen on välttämätön edellytys CORBA-järjestelmien toteuttamiselle, jos käytetyn ORB-tuotteen IDL-kääntäjä muuntaa rajapintakuvaukset C++-koodiksi. Vaikka käytettäisiinkin jotain muuta ohjelmointikieltä, IDL-tason suunnittelussa tarvitaan perustiedot oliosuuntautuneesta suunnittelusta. Itse IDL-kieli on erittäin helppo oppia.

Kokeneen C++-ohjelmoijan on vaivatonta siirtyä käyttämään C++:lla toteutettua CORBA-tuotetta, esimerkiksi Orbixia. Perustiedot tietoverkkojen toiminnasta (TCP/IP) on hyvä hallita, mutta ei välttämätöntä. Jos saatavissa on hyvät manuaalit ja esimerkit, opiskelutarve yksinkertaisten sovellusten tekemiseen on muutamia kuukausia. ORB-toteutuksen edistyneempien piirteiden oppimiseen ja tehokkaalla tavalla soveltamiseen tarvitaan ainakin puolen vuoden opettelu.

6.2 CORBAn soveltuvuus joustaviin valmistusjärjestelmiin

ORB-tuotteen käyttäminen hajautettujen järjestelmien ohjelmointiin antaa mahdollisuudet keskittyä paremmin varsinaiseen ongelmaan piilottamalla matalan tason ohjelmoinnin. Koska rajapintojen määrittelyssä käytetään operaatioita ja niille tyypitettyjä parametreja, käännösaikana saadaan selville virheitä, jotka ilman vahvaa tyypintarkastusta havaitaan vasta ohjelman suorituksen aikana. Sovellusten ohjelmointi C++-perustaisella ORB-tuotteella noudattaa hyvin pitkälle tavallista C++-ohjelmointia, ja IDL-rajapintakuvausissa käytettävissä olevat tietotyypit tarjoavat riittävästi ilmaisuvoimaa. Suurin CORBAn tarjoama hyöty saavutetaankin ehkä sovelluskehityksen helpottumisen myötä.

CORBAn tarjoama arkkitehtuuri voidaan mieltää muös tuotekehitystä ohjaavaksi valmiiksi suunnittelumalliksi, joka luo pohjan yhtenäiselle sovelluskehitykselle. Sen periaatteiden hyödyntäminen on mahdollista, vaikka ei käytettäisikään ORB-tuotetta ohjelmiston toteuttamiseen.

Ohjelman kehitystyön selkiytymisen ja systematisoitumisen ohella ORB-tuotteet tarjoavat varsin suorituskykyisen hajautusalustan. Millisekuntien luokkaa olevat vasteajat operaatiokutsuissa riittävät pehmeisiin reaaliaikasovelluksiin. Yhteyden ottaminen kohdeolioon kestää kauemmin, mutta mikäli yhteydenotot voidaan suorittaa järjestelmän alustusrutiinien yhteydessä, tämän viiveen merkitys pienenee.

Tuki poikkeustilanteiden käsittelylle on jo IDL-tasolla. Rajapintakuvauksen yhteydessä ohjelmoija voi määritellä kunkin operaation yhteydessä asiakkaalle palautettavan poikkeuksen virhetilanteen tapauksessa. Asiakas havaitsee mahdollisen poikkeustilanteen operaatiokutsun suorittamisen yhteydessä *try-catch*-makroilla. Yksinkertainen mutta vahva virheenkäsittelymekanismi on tärkeä apuväline hajautettujen järjestelmien toteuttamisessa millä tahansa sovellusalueella.

6.2.1 Paikkariippumattomuus

Orbix tarjoaa asiakasohjelmille palvelinten paikkaläpinäkyvyyden konfigurointitiedoston avulla *_bind()*-metodia käytettäessä. Mikäli asiakas ei määrittele palvelimen isäntäkonetta, Orbix käyttää oletuksena oletuspaikannintaan, joka lukee konfigurointitiedostosta palvelimen isäntäkoneen nimen. Lisäksi *_bind()*-metodi on rajapintakohtainen ja edellyttää palvelimen ja olion nimen määrittämistä[7]. Palvelimen nimen määrittäminen ottaa kantaa olion toteutukseen eikä sen vuoksi eristä täydellisesti olion rajapintaa ja toteutusta. Jos tämä muodostuu ongelmaksi, oletuspaikantimen voi korvata omalla paikannusmekanismilla. Orbixin nimipalvelu mahdollistaa olioviittauksen hankkimisen toteutusriippumattomammalla tavalla: olioviittaus saadaan määrittelemällä pelkästään olion nimi merkkijonomuodossa ja tekemällä tyypinmuunnos palautetusta yleisestä olioviittauksesta kohdeolion tyyppiin [8].

Chorus/COOL ORB tarjoaa tuotteen mukaan upotetun nimipalvelun, jonka avulla asiakasohjelma voi hankkia olioviittauksen pelkän olion loogisen nimen perusteella. Tyypinmuunnos yleisestä oliovittaustyypistä kohdeolion viittaustyypiksi tarvitaan tässäkin, mutta asiakkaan ei tarvitse tietää olion toteuttavasta palvelimesta mitään. Chorus/COOL ORB tarvitsee myös konfigurointitiedostoja eri koneiden yhdistämiseen, mutta niissä ei oteta kantaa koneilta löytyviin palvelimiin.

---

[7] Palvelimen, olion tai molempien nimi voidaan jättää määrittelemättä, jolloin Orbix käyttää oletuksia kohdan 4.1.6 mukaisesti.

6.2.2 Resurssien käyttö

ORB-ohjelmiston resurssien tarve voi olla tekijä, joka rajaa mahdollisia käyttökohteita. Windows NT -käyttöjärjestelmän Task Manager -ohjelman mukaan yksinkertainen Orbix-sovellus tarvitsee vajaan megatavun keskusmuistia, ja Orbix-daemon saman verran. Suurissa järjestelmissä käyttöjärjestelmän asettamat rajoitukset socket-yhteyksien ylläpidossa käytettäville tiedostokuvaajille [15] ja TCP/IP-protokollan rajoitus porttien lukumäärälle [16] saattavat aiheuttaa ongelmia. Järkevällä suunnittelulla nämä potentiaaliset ongelmat ovat kuitenkin hyvin pitkälle vältettävissä.

Laajennettavuuden kannalta merkittävässä osassa ovat myös ORB-toteutuksen sisäiset tietorakenteet. Esimerkiksi Orbixin käyttämä oliotaulu palvelimen muistissa ei ole dynaaminen, vaikka sen kokoa voikin API-funktiolla muuttaa. Jos ORB-toteutus käyttää lineaarista hakua kohdeolion ja suoritettavan metodin paikantamiseen, vasteajat kasvavat olioiden määrän ja rajapinnan määrittelemien operaatioiden lukumäärän kasvaessa [21].

6.2.3 Tuotevariointi

CORBAn tarjoama tuki sovellusalueen ohjelmistojen variointiin on lähinnä IDL-kuvauskielen tarjoamassa rajapintojen perintämekanismissa. Samantyyppisille olioille voidaan määritellä yhteinen kantarajapinta, jonka ominaisuudet johdetut rajapinnat perivät ja joihin ne voivat lisätä uusia ominaisuuksia. Näiden muunnelmien toteutuserot koodataan Orbixin tapauksessa C++-tasolla ylikirjoittamalla yleisen kantaluokan operaatiot, tarvittaessa erikseen jokaisessa johdetussa luokassa. Järjestelmän yksittäisten luokkien yhdistäminen suuremmiksi toiminnallisiksi kokonaisuuksiksi luo edellytykset järjestelmien rakentamiseen komponenttiperustaisesti.

6.2.4 Hajautusalustan jatkokehitystarpeet ja tulevaisuuden näkymät

ORB-toteutusten laaja kirjo aiheuttaa arvostelua CORBA-arkkitehtuuria kohtaan. Vaikka ei pitäisi yhdistää yksittäisiä ORB-toteutuksia ja itse standardia, pelkästä standardista on varsin vähän hyötyä. Siksi CORBA on kullakin ajanhetkellä loppukäyttäjälle vain niin hyvä kuin on sen paras ORB-toteutus, ellei ORB:tä toteuteta itse.

CORBA 3.0 -spesifikaatio on työn alla. Sen olennaisin osa on POA:n määrittäminen, jolloin eri ORB-toteutusten palvelinpuolen siirrettävyys parantuu. API-kutsujen tulee

olla yhtenäiset ORB-toteutusten välillä, jotta sovelluksen siirto toiseen ORB-järjestelmään vaatisi käännöstyön lisäksi mahdollisimman vähän muutoksia lähdekoodiin.

Osittain vielä virheellisesti ja puutteellisesti toimivat ORB-toteutukset haittaavat CORBAn menestymistä hajautuksen tukiteknologioiden markkinoilla. Ne selittävät osaltaan tarpeellisten CORBA-palvelujen hidasta markkinoille tuloa, sillä ORB-ydin on perustuote, minkä ostamisesta järjestelmän toteuttaja aloittaa investoinnit. Mikäli ORB-ydin osoittautuu huonosti toteutetuksi, asiakas joko vaihtaa ORB-toimittajaa tai valitsee jonkun muun hajautuksen tukiteknologian järjestelmänsä toteuttamiseen.

Yleisimmin tarvitut CORBA-palvelut on saatava toimivalle tasolle, jotta ORB-toteutusten käyttöönottokynnys saadaan riittävän alas. Uuden teknologian ja sen opettelun vaatimat investoinnit ovat sitä luokkaa, että suuritöinen sovellusriippumattomien peruspalvelujen toteuttaminen on monelle ohjelmistonkehittäjälle liian suuri lisätaakka. Hajautusalustan ja ohjelmistoväylän tarkoitus on tarjota sovellusten kehittäjälle mahdollisuus keskittyä vain ja ainoastaan sovellusalueeseen eikä tuoda mukanaan uusia ongelmia.

Microsoftin Distributed Component Object Model (DCOM) on CORBAn vahvin kilpailija. Sen etuna on Microsoftin markkina-asema ja tuotteen saaminen kääntäjän mukana. Vuonna 1997 esimerkiksi Orbix 2.2 kehityslisenssi Windows NT -käyttöjärjestelmään maksoi 2 500 dollaria ja ajonaikaiset lisenssit 100 dollaria kappale. Vuonna 1998 markkinoitua Orbixin OTS-kehityslisenssiä tarjottiin hintaan 6 500 dollaria ja päivitysversiota hintaan 3 000 dollaria. Yleensä ORB-tuotteet vaativat lisäksi tietyn version kääntäjästä.

Myös ORB-toteutusten suorituskyky on sellainen tekijä, joka rajaa mahdollisia sovelluskohteita. Koville reaaliaikajärjestelmille millisekuntien vasteajat ovat edelleenkin liian suuria, ja epädeterministisyyttä kutsujen kestoajoissa on karsittava. OMG onkin asettanut erityisen asiantuntijaryhmän reaaliaika-CORBAn kehittämiseen, jotta reaaliaikavaatimusten kannalta rajoittavat tekijät saadaan minimoitua. Douglas C. Schmidtin kotisivulta *http://siesta.cs.wustl.edu/~schmidt/* löytyy ajankohtaista tietoa reaaliaika-CORBAsta.

7. YHTEENVETO

Tässä työssä tarkasteltiin CORBA-arkkitehtuuria ja sen mukaisia ORB-toteutuksia sekä sovellettiin IONA Technologiesin Orbix-tuotetta FM-järjestelmää ohjaavan perusohjelmiston toteuttamiseen. Tehdyn työn perusteella analysoitiin CORBAn ja sen toteutusten mukanaan tuomia etuja ja haittoja sekä soveltuvuutta hajautuksen tukiteknologiaksi.

Ohjelmistojen hallittavuus ja matalan tason ohjelmointi ovat ongelmia, joita CORBAn avulla pyritään helpottamaan. IDL-rajapintakuvausten avulla ohjelmisto voidaan jakaa modulaarisiin komponentteihin, jotka piilottavat toteutuksensa niitä käyttäviltä asiakasohjelmilta. Oliokielellä toteutetulla ORB-tuotteella komponenttien välinen kommunikointi rakentuu tavallista olio-ohjelmointia muistuttavalla syntaksilla välittämättä alla olevien käyttöjärjestelmien tai siirtoprotokollien monimutkaisuudesta.

Heterogeenisten ympäristöjen yhteenliittäminen on tavoite, joka on jatkuvasti työn alla. CORBA-pakkaaja-luokkien käyttö mahdollistaa jo olemassa olevien ohjelmistojen integroimisen CORBA-järjestelmään, ja IIOP mahdollistaa eri ORB-toteutusten loogisen yhteensulauttamisen TCP/IP-ympäristössä.

ORB-toteutusten mahdolliset puutteet toiminnallisuudessa, toimintavarmuudessa ja dokumentoinnissa haittaavat CORBA-järjestelmien kehittämistä. Molemmat työn yhteydessä testatut ORB-toteutukset jättivät kuitenkin positiivisen vaikutelman: niiden avulla hajautettujen järjestelmien toteuttaminen on yksinkertaista verrattuna esimerkiksi socket-yhteyden avulla toteutettuihin sovelluksiin. Korkea abstraktiotaso, olio-suuntatuneisuus, sovellusten siirrettävyys ja yhteistoiminta ovat asioita, joista ollaan valmiita maksamaan ja joita pyritään ratkaisemaan, vaikkei CORBA siinä täydellisesti onnistuisikaan.

Suorituskyky, korkeamman tason palvelut, kuten tietokantaintegrointi ja tapahtumien hallinta sekä ongelmatilanteiden varalta tarjottavat tukipalvelut, ovat asioita, joissa tällä hetkellä on eniten kehittämisen varaa.

LÄHTEET

[1] Vannas, V. 1993. FM-järjestelmien turvallisuus ja käyttöönotto. Tampereen Teknillinen Korkeakoulu. 101 s.

[2] Mortimer, J. 1982. The FMS Report. IFS Publications Ltd. 179 s.

[3] Ljung, S. 1986. Robot Cells And System for Small Part Assembly. Proceedings of the 5th International Conference on Flexible Manufacturing Systems. IFS Publications Ltd. S. 512.

[4] Architecture Technology Corporation 1988. Flexible Manufacturing Systems Report. 2nd edition. S. 3-27 - 3-33.

[5] OMG 1995. Common Object Request Broker: Architecture and Specifications. Version 2.0. OMG.

[6] Mowbray, T., Zahavi, R. 1995. The Essential CORBA. John Wiley & Sons, Inc. 316 s.

[7] Mowbray, T., Ruh, W. 1997. Inside CORBA. Addison-Wesley. 376 s.

[8] Baker, S. 1997. CORBA Distributed Objects Using Orbix. Addison-Wesley. 518 s.

[9] OMG 1997. Common Object Request Broker: Architecture and Specifications. Version 2.1. OMG.

[10] Orfali, R., Harkey, D., Edwards, J. 1997. Instant CORBA. John Wiley & Sons, Inc. 313 s.

[11] IONA Technologies 1997. Orbix 2 Programming Guide. 383 s.

[12] IONA Technologies 1997. Orbix 2 Reference Guide. 441 s.

[13] http://www.chorus.com. Helmikuu 1998.

[14] Dayal, U., Buchmann, A., McCarthy, D. 1988. Rules Are Objects Too: A Knowledge Model for an Active, Object-Oriented Database System. In: Dittrich, K. (ed.). LNCS 334, Advances in Object-Oriented Database Systems. Springer-Verlag. S. 129 - 143.

[15] Comer, D., Stevens, D. 1993. Internetworking with TCP/IP. Prentice Hall. S. 43 - 55.

[16] Black, U. 1995. TCP/IP & Related Protocols. McGraw-Hill, Inc. S. 164.

[17] Schmid, H. 1996. Design Patterns for Constructing the Hot Spots of a Manufacturing Framework. Journal of Object Oriented Programming. Vol. 9, No. 3.

[18] Fayad, M., Schmidt, D. 1997. Object-oriented Application Frameworks. Communications of the ACM. Vol. 40, No. 10.

[19] Niemelä, E., Holappa, M. 1998. Experiences with the Use of CORBA. Ehdotettu julkaistavaksi 24 th Euromicro Conference 98 -julkaisussa. 7 s.

[20] Mowbray, T., Malveau, R. 1997. CORBA Design Patterns. John Wiley & Sons, Inc. S. 121 - 125.

[21] Gokhale, A., Schmidt, D. 1997. Measuring and Optimizing CORBA Latency and Scalability Over High-speed Networks. Proceedings of the International Conference on Distributed Computing Systems. Baltimore, MD, May 1997.