Joona Vuolle

OLIOPOHJAISTEN VÄLIOHJELMISTOJEN RAKENNE JA TEKNINEN TOIMINTA

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

9.12.2003

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Joona Vuolle

Yhteystiedot: Sähköposti: joona.vuolle@processvision.fi

Työn nimi: Oliopohjaisten väliohjelmistojen rakenne ja tekninen toiminta

Title in English: Structure and Technical Operation of Object Request Broker Middleware

Työ: pro gradu -tutkielma

Sivumäärä: 120+11

Linja: Ohjelmistotekniikka.

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: Hajautetut järjestelmät, väliohjelmisto, asiakas-palvelinmalli, etäkutsu, etäreferenssi, sarjallistaminen, DCOM, CORBA.

Keywords: Distributed systems, middleware, client-server architecture, remote invocation, remote reference, marshalling, DCOM, CORBA.

Tiivistelmä: Tutkielmassa tarkastellaan oliopohjaisten hajautettujen väliohjelmistojen yleistä rakennetta ja toimintaa. Pääpaino tarkastelussa on väliohjelmistojen hajautukseen ja prosessien väliseen kommunikaation liittyvissä yksityiskohdissa. Tutkimuskohteina ovat yksityiskohtaisemmin DCOM- ja CORBA-väliohjelmistot, joiden teknisiä toteutustapoja vertaillaan toisiinsa teoriaosuuden teknisen tarkastelun ja hajautettuun ympäristöön toteutetun sovelluksen avulla.

Abstract: The master thesis discusses the general structure and function of the object request broker middleware. The discussion concentrates on the technical details of distribution and communication between the distributed processes. The DCOM and CORBA middlewares are studied more closely and their technical implementations are compared in theory and in practice.

Esipuhe

Tämän työn kirjoittaminen muuttui yllättävän pitkäksi projektiksi ja tahdonkin kiittää kaikkia, jotka auttoivat minua sen teossa. Kiitos lehtori Jukka-Pekka Santaselle, joka ohjasi ja jaksoi kommentoida työtäni sen edistyessä. Kiitos myös Heikki Liimataiselle sekä Process Vision Oy:lle, jonka antama tuki oli tärkeää kirjoittamisen aikana.

Kiitokset myös ystävilleni ja erityisesti jo valmistuneille, jotka esimerkillään motivoivat minua saattamaan työn valmiiksi.

Viimeiseksi suuri kiitos vanhemmilleni Paulille ja Tuulalle sekä siskolleni Tellalle, jotka kaikki kannustivat ja rohkaisivat minua koko kirjoitusprosessin ajan.

Jyväskylässä 8.12.2003
Joona Vuolle

Termiluettelo

ATL	(Active Template Library) on Microsoft Visual C++ kääntäjän mukana tuleva luokkamallikirjasto, joka helpottaa COM- ja DCOM-luokkien kehittämistä.
Base64	on koodaustapa, jossa 8-bittinen data muunnetaan 7-bittiseksi dataksi. Menetelmää käytetään tarvittaessa välittämään binääridataa kirjoitusmerkkeinä.
CDR	(Common Data Representation) on muun muassa CORBAn käyttämä siirtomuoto parametrien välittämiseksi verkon yli.
CORBA	(Common Object Request Broker Architecture) on standardoitu arkkitehtuuri hajautettujen järjestelmien kehittämiseksi.
DARPA	(Defense Advanded Research Project Agency) on Yhdysvaltain puolustusministeriön tutkimusosasto, joka tunnetaan TCP/IP-protokollan kehittämisestä.
DCE	(Distributed Computing Environment) on asiakas-palvelin-mallisten hajautettujen järjestelmien toteutusstandardi.
DCOM	(Distributed Component Object Model) on Microsoftin kehittämä oliopohjainen hajautusjärjestelmä.
DLL	(Dynamic Link Library) on Windows-käyttöjärjestelmässä käytetty dynaaminen ajonaikana ladattava sovelluskoodikirjasto.

DNS	(Domain Name Service) on nimipalvelu, jonka avulla Internetin selkokieliset osoitteet muutetaan IP-protokollan ymmärtämiksi numeropohjaisiksi osoitteiksi.
DTD	(Document Type Definition) on lähinnä XML-tiedostojen yhteydessä käytetty dokumentin rakenteen kuvaava kieli.
EBCDIC	(Extended Binary-Coded Decimal Interchange Code) on IBM:n standardi merkkien esittämiseksi binäärilukuina.
Enumeraatio	on järjestyvä joukko symboleita, joihin liittyy vakio numeroarvo, esim. `{NAINEN=1, MIES=2}`.
EXE	(EXEcutable) on Windows-käyttöjärjestelmien sovellustyyppi, joka voidaan suorittaa.
HTTP	(HyperText Transfer Protocol) on laajasti käytetty protokolla tekstin ja kuvien siirtoon Internetissä.
IDL	(Interface Definition Language) on kuvauskieli, jolla esitellään luokan toteuttama rajapinta.
IEEE	(Institute of Electrical and Electronics Engineers) on teknisten alojen ammattilaisten yhteenliittymä, joka kehittää mm. teknisiä standardeja.
IETF	(Internet Engineering Task Force) on yhteenliittymä, joka hallinnoi ja julkaisee ehdotuksia sekä standardeja Internet-tekniikoista.

IID-tunnus	tarkoittaa globaalisti uniikkia tunnusta, joka yksilöi olion tietyn rajapinnan.
IOR	(Interoperable Object Reference) on CORBA-arkkitehtuurin käyttämä yhtenäinen olioreferenssien esitystapa.
IPID	on ORPC:n käyttämä tunniste, joka yksilöi tietyn palvelinprosessin tietyn olion rajapinnan.
Kriittinen alue	(engl. c*ritical section)* on alue sovelluskoodissa, joka suojataan niin, että vain yksi säie kerrallaan pystyy sitä suorittamaan.
MFC	(Microsoft Foundation Classes) on Microsoft Visual C++ -kääntäjän mukana tuleva luokkakirjasto, joka helpottaa Windows-sovellusten kehittämistä.
NDR	(Network Data Representation) on muun muassa DCE/RPC-protokollan ja DCOMin käyttämä siirtomuoto parametrien välittämiseksi verkon yli.
NFS	(Network File System) on RPC:tä käyttävä protokolla tiedostojen jakamiseksi verkossa.
OAD	(Object Activation Daemon) on väliohjelmistopalvelin, joka huolehtii CORBA-palvelinprosessien konekohtaisesta aktivoinnista (vrt. SCM).
OID	on ORPC:n käyttämä tunniste, joka yksilöi tietyn palvelinprosessin ylläpitämän olion.
Oktetti	on kahdeksasta bitistä koostuva yksittäinen tavu.

ONC	(Open Network Computing) on Sun Microsystemsin standardi hajautettujen järjestelmien toteutukseen.
ORPC	(Object Remote Procedure Call) on vahvasti RPC:hen perustuva DCOMin käyttämä protokolla olioiden hajautukseen.
OXID	on ORPC:n käyttämä tunniste, joka yksilöi RPC:n merkkijonosidonnan (engl. *RPC string binding*) ja mahdollistaa kytkennän IPID:n määräämään rajapintaan.
PDU	(Protocol Data Unit) on tietue, jota RPC käyttää kommunikoinnissaan.
Ranka	(engl. *stub, skeleton*) on olion rajapintaan liittyvä etäkutsun sarjallistava komponentti palvelinprosessissa.
RMI	(Remote Method Innovacation) merkitsee olion metodikutsun välitystä verkon yli eri koneiden välillä. Sitä käytetään usein JavaRMI:n yhteydessä.
RPC	(Remote Procedure Call) on aliohjelmakutsu, joka tapahtuu verkon yli eri koneilla sijaitsevien prosessien välillä.
Sarjallistaminen	(engl. *marshalling*) on toimenpide, jossa etäkutsulle välitettävät parametrit pakataan verkon yli välitettävään muotoon.
SCM	(Service Control Manager) on väliohjelmistopalvelin, joka hallitsee DCOM-palvelinprosesseja sekä koneiden välisiä yhteyksiä (vrt. OAD).

Skriptikieli	on yleisnimitys heikosti tyypitetyille ohjelmointikielille (esim. Tcl ja Perl), joita käytetään usein muiden sovelluksien yhteenkytkemiseen.
SMTP	(Simple Mail Transfer Protocol) on protokolla sähköpostiviestien välitykseen Internetissä.
Sokettirajapinta	on useiden käyttöjärjestelmien tukema ohjelmointirajapinta TCP/IP-protokollan käyttämiseksi sovelluksista käsin.
TFTP	(Trivial File Transfer Protocol) on vain lukua ja kirjoitusta tukeva yksinkertainen protokolla koneiden välisiin tiedostonsiirtoihin.
Tynkä	(engl. *proxy, stub*) on olion rajapintaan liittyvä etäkutsun sarjallistava komponentti.
UTC	(Universal Time Coordinated) on kansainvälinen koordinoitu maailmanaika.
XDR	(eXternal Data Representation) on ONC/RPC-protokollan käyttämä siirtosyntaksi parametrien välittämiseksi verkon yli.

Sisältö

1 Johdanto

Ohjelmistojen arkkitehtuurit ovat muuttunut viimeisen kolmen vuosikymmenen aikana huomattavasti. Alkuaikojen yksinkertaisista, yhden prosessin käsittävistä ohjelmista, on siirrytty monimutkaisiin, useita erillisiä prosesseja käsittäviin ohjelmistoihin, jotka on usein hajautettu eri laitteistojen kesken. Tällaisten ohjelmistojen toteutus on hyvin vaativaa, joten on tullut tarpeelliseksi kehittää ns. väliohjelmistoja (engl. *middleware*). Ne piilottavat verkkotasolla tapahtuvan prosessien välisen kommunikoinnin ja yksinkertaistavat näin varsinaisen ohjelmiston kehitystä.

Väliohjelmistoja hajautuksen toteuttamiseen on tänä päivänä tarjolla useita. Markkinoita hallitsevat kuitenkin muutamat paikkansa vakiinnuttaneet laajalti käytössä olevat ratkaisut. Valinta näidenkin välillä voi kuitenkin olla hankalaa. Oikean väliohjelmiston käyttö oikeaan tarkoitukseen on kriittistä järjestelmäkehityksen onnistumisen kannalta.

Tutkielmassa tarkastellaan, mihin tekniikoihin hallitsevat hajautusväliohjelmistot perustuvat, millä tavalla ne on teknisesti toteutettu sekä minkälaisia yhteneväisyyksiä toteutuksien väliltä löytyy. Tarkastelun pääkohteina ovat DCOM- ja CORBA-arkkitehtuurit. Tavoitteena on selvityksen pohjalta analysoida ja verrata kyseisten väliohjelmistojen sopivuutta yrityksen nykyiseen sovelluskehitykseen.

Tutkielma alkaa työn taustan ja tavoitteiden tarkentamisella luvussa 2. Seuraavaksi tutustutaan luvussa 3 yleisellä tasolla hajautettuihin järjestelmiin ja niille asetettuihin vaatimuksiin, kerrotaan erilaisista arkkitehtuureista, määritellään hajautettuun oliomalliin liittyviä käsitteitä sekä esitellään erityyppisiä väliohjelmistoja. Luvussa 4 kuvataan lyhyesti väliohjelmistojen käyttämiä verkko-, siirto- ja sovelluskerrosten protokollia. Tämän jälkeen luvussa 5 esitellään ensin olemassa olevia väliohjelmistototeutuksia ja selvitetään, mistä osista väliohjelmistot rakentuvat.

Luvuissa 6 ja 7 kuvataan ensin yleisesti ja sitten yksityiskohtaisemmin DCOMin ja CORBAn osalta hajautukseen hallintaan ja etäkutsujen tekoon liittyviä toimenpiteitä. Tutkielman käytännön osuudessa luvuissa 8 ja 9 kuvataan Generis Alarmer -ohjelmiston suunnittelu, toteutus ja testaus. Luvussa 10 on esitetty väliohjelmiston suorituskyvyn

mittaamiseen tarkoitettujen testisovellusten toteutus sekä testaustulokset. Generis Alarmerin toteutuksen analyysi, testitulosten tarkastelu sekä väliohjelmistojen vertailu tapahtuu luvussa 11.

2 Tutkielman taustaa ja tavoitteet

Ohjelmistojen hajautuksen historia on kirjava ja erilaisia toteutusratkaisuja on kehitetty lukuisia. Osa toteutuksista on hiipunut nopeasti pois tai jäänyt muutaman ohjelman käyttöön. Näistä osa on ollut puhtaasti kokeiluluonteisia tutkijapiireissä suunniteltuja ratkaisuja. Jotkut toteutukset taas ovat osoittaneet paremmuutensa ja alkaneet levitä ohjelmistokehittäjien keskuudessa. Luvussa esitellään lyhyesti hajautuksen historiaa sekä kuvataan tutkielman taustaa ja tavoitteet.

2.1 Hajautettujen järjestelmien historiaa

Ensimmäiset hajautetut järjestelmät oli suunniteltu lähinnä tiedostojen jakamiseen lähiverkoissa. Vuonna 1985 julkaistu Sun Microsystemsin kehittämä NFS (*Network File System*) oli ensimmäinen todelliseksi tuotteeksi tarkoitettu ratkaisu tähän tarkoitukseen. Sen menestys pohjautui helppoon siirrettävyyteen eri laitteistojen ja käyttöjärjestelmien välillä sekä julkisiin rajapintoihin, jotka nykyään tunnetaan Internet-standardina RFC 1813 [Cal95]. Muita samoihin aikoihin kehitettyjä jaettuja tiedostojärjestelmiä olivat mm. AFS (*Andrew File System*), Apollo DOMAIN sekä RFS (*Remote File Sharing*), joista varsinkin ensimmäinen on yhä laajassa käytössä [Bor89].

Hajautetut tiedostojärjestelmät sekä 80-luvun puolenvälin jälkeen yleistyneet hajautetut tietokannat mahdollistivat tiedon jakamisen, mutta tiedon laajempaan käsittelyyn hajautetusti etsittiin samaan aikaan uusia ratkaisuja. Ensimmäinen tärkeä oliopohjainen hajautettu järjestelmäratkaisu oli Cronus. Sen ominaisuuksia olivat mm. täydellinen ympäristö hajautettujen ohjelmien kehitykseen, toiminta heterogeenisissä ympäristöissä sekä yhteensopivuus eri käyttöjärjestelmien (kuten Unixin ja VAXin) kanssa. Muita samantyyppisiä ratkaisuja olivat mm. Eden (1981) sekä Clouds (1986), jotka kummatkin oli suunnattu lähinnä Unix-käyttöjärjestelmille [Bor89].

Merkittävä askel kohti nykyisiä hajautettuja väliohjelmistoja otettiin, kun Open Group -ryhmä julkaisi DCE-ympäristönsä (*Distributed Computing Environment*) vuonna 1991. DCE käsittää joukon palveluita, kuten nimipalvelu, hakemistopalvelu, aikapalvelu ja säiepalvelu, jotka yhdessä muodostavat alustan hajautuksen toteutukseen. Prosessien

väliseen kommunikointiin DCE käyttää RPC-protokollaa (*Remote Procedure Call)*, joka mahdollistaa DCE:n käytön useissa erilaisissa käyttöjärjestelmissä ja ympäristöissä. DCE ei kuitenkaan tukenut samaan aikaan yleistyvää oliopohjaista sovelluskehitystä. Tähän puutteeseen tuli korjaus vuonna 1989 perustetun OMG-ryhmän (*Object Management Group*) toimesta.

OMG on yli 700 ohjelmistotuottajan ja käyttäjän yhteenliittymä, joka perustettiin kehittämään oliopohjaisia hajautustekniikoita. Tavoitteena ryhmällä on luoda yhteinen arkkitehtuurikehys hajautettujen olioiden kommunikointiin heterogeenisissä laitteistoissa ja käyttöjärjestelmissä. Vuonna 1990 OMG esitteli ensimmäisen kuvauksen kehittämästään OMA-arkkitehtuurista (*Object Management Architecture).* Vuonna 1991 julkaistu CORBA 1.1 -määrittely (*Common Object Request Broker Architecture*) kuvasi ne rajapinnat ja palvelut, jotka OMA-arkkitehtuurin kriittisimmän osan eli ORB-palvelun (*Object Request Broker)* pitää toteuttaa. Tämän jälkeen ohjelmistokehittäjien oli mahdollista lähteä toteuttamaan ensimmäisiä CORBA-implementaatioita [Yan96].

Merkittävä ohjelmistovalmistaja Microsoft ei kuitenkaan lähtenyt kehittämään omaa CORBA-toteutustaan (vaikka olikin osa OMG-ryhmää), vaan julkaisi 1996 DCOM-mallin (*Distributed Component Object Model)* ratkaisuna oliopohjaiselle hajautukselle. Se mahdollisti sovelluskomponenttien käytön eri prosessien ja koneiden välillä yhtenevällä tavalla. DCOM oli alun perin suunniteltu käytettäväksi heterogeenisissä ympäristöissä eri käyttöjärjestelmien kesken, mutta tällä hetkellä se on käytössä lähinnä vain Windows-käyttöjärjestelmissä [COM98].

Java-ohjelmointikielen kehittänyt Sun Microsystems erkani myös CORBA-leiristä kehittämällä oman oliopohjaisen hajautusratkaisunsa, joka tunnetaan nimellä JavaRMI (*Remote Method Invocation*). Sen suurimpia etuja aiemmin mainittuihin verrattuna ovat helppokäyttöisyys ja mahdollisuus siirtää luokkien toteutuksia prosessien välillä ajonaikana [SUN].

2.2 Teknologian valinnan vaikeus

Kuten luvusta 2.1 käy ilmi, valinnan mahdollisuuksia hajautuksen toteuttamiseksi on useita. Kuinka siis valita tarkoitukseen sopivin ratkaisu? Tämä kysymys oli pääasiallisena lähtökohtana tutkielman tavoitteille. Ensin kuitenkin kerron valintatilanteesta, jonka edessä työnantajayritykseni oli ja joka antoi sysäyksen tutkielman synnylle.

Tutkielman tilaajana toimi Process Vision Oy. Yritys on toiminut 90-luvun alkupuolelta lähtien ja työntekijöitä on tällä hetkellä noin 60. Yritys aloitti kehittämällä ohjelmistoja energiatuotantoalan prosessien simulointiin, mutta on viime vuosina siirtynyt entistä enemmän energiakauppojen hallinta- ja välitysohjelmistojen kehittämiseen. Yrityksen hajautettujen ohjelmistojen toteutus oli pienimuotoista vuosikymmenen vaihteeseen saakka, mutta ohjelmien luonteen muuttuessa ovat niiden suorituskykyvaatimukset kasvaneet ja on tullut tarpeelliseksi hajauttaa raskaimmin kuormitettuja osia järjestelmistä.

Aiemmat yrityksen hajautusratkaisut (tietokannan laajan käytön lisäksi) ovat perustuneet lähinnä TCP/IP-protokollaa käyttäviin sanomiin, joiden lähetys- ja vastaanottorutiinit on toteutettu sokettirajapinnalla. Kommunikointi on ollut melko yksinkertaista ja yksittäisiin viesteihin perustuvaa. Uudet tarpeet ovat monimutkaisempia ja ohjelmistojen toteutustapoja on haluttu kehittää lähemmäs kolmikerrosarkkitehtuurin tarjoamia ratkaisuja.

Koska yrityksen nykyisistä tuotteista suurin osa on kehitetty käyttäen olio-ohjelmointia, vaikutti oliopohjaisten väliohjelmistojen käyttö parhaimmalta vaihtoehdolta hajautusratkaisua valittaessa. Yrityksessä on toteutettu muutamia projekteja DCOM-väliohjelmistoa käyttäen, mutta sen soveltuvuus haluttiin uudelleen arvioida. Teknologian valintatilanne tuli konkreettisesti eteen, kun yritys alkoi kehittämään hajautettua hälytysten käsittelyohjelmistoa nimeltään Generis Alarmer.

2.3 Tutkielman tavoitteet

Tutkielman tärkeimpänä tavoitteena oli selvittää, kuinka jo käytössä oleva DCOM-väliohjelmisto soveltuu yrityksen ohjelmistokehitykseen sekä tutkia muiden väliohjelmistojen sopivuus kyseiseen tarkoitukseen. Yhtenä vaihtoehtona haluttiin erityisesti tarkastella CORBA-arkkitehtuurin käyttöä. Selvitystyö väliohjelmistojen kesken tapahtui tarkasteltavien väliohjelmistojen teknisten ominaisuuksien, suorituskyvyn ja käytännön ohjelmointiteknisten toteutusratkaisujen pohjalta.

Tutkielman sivutavoite oli lisätä yrityksen hajautettuihin väliohjelmistoihin liittyvää tietotaitoa. Tietoa kaivattiin erityisesti väliohjelmistojen teknisistä ominaisuuksista sekä suorituskykyyn ja luotettavuuteen liittyvistä seikoista. Lisäksi tavoitteena oli saada konkreettista vertailutietoa tarkasteltavaksi valittujen väliohjelmistojen eroista.

2.4 Tutkimusasetelma

Pääosin tutkielmassa keskityttiin kahteen laajasti käytettyyn hajautustekniikkaan: Microsoftin DCOM-arkkitehtuuriin sekä OMG:n CORBA-arkkitehtuuriin. Kyseiset väliohjelmistot valittiin tarkasteltaviksi seuraavista syistä. Ensinnäkin niiden rajapintojen ja palveluiden määrittelyt ovat vakiintuneet ja täten vertailtavissa toisiinsa. Toiseksi kummastakin on tarjolla paljon yksityiskohtaista materiaalia, jonka avulla on mahdollista selvittää ohjelmiston teknisiä toteutustapoja syvällisemmin. Lisäksi DCOM-väliohjelmistoa oli jo hyödynnetty muutamissa aiemmissa yrityksen järjestelmissä ja sen soveltuvuus haluttiin arvioida uudelleen.

Tutkielmaan sisältyi myös Generis Alarmer -järjestelmän toteuttaminen ja niiden johtopäätösten esittely, joiden takia valittuun hajautustekniikkaan päädyttiin. Ohjelmiston toteutuksen tarkoituksena oli lisäksi havainnollistaa käytännössä joitakin teoriaosuudessa esiteltyjä asioita toteutettuina valitulla väliohjelmistolla ja yrityksen käytössä olevilla työkaluilla. Käytännön toteutus oli myös tarpeen, jotta analyysin teko ohjelmistolle asetettujen vaatimusten täyttymisestä ja tekniikan valinnan oikeellisuudesta oli mahdollista. Lisäksi tutkielmaan kuului Generis Alarmeria simuloivan järjestelmän

toteuttaminen vaihtoehtoisella väliohjelmistolla, jotta eri väliohjelmistototeutuksia voisi vertailla keskenään.

Tutkielmalle asetettuun tavoitteeseen pyrittiin väliohjelmistojen teknisiä ominaisuuksia käsittelevän teoriaosuuden, DCOM- ja CORBA-teknologioita hyödyntävien sovellusten toteutuksien vertailun sekä sovelluksille suoritettujen testien avulla. **Teoriaosuudessa** ensisijainen tavoite oli löytää yhteneväisyyksiä, joita väliohjelmistojen toteutustavoilla on keskenään. Yhteneväisyyksien avulla tekniikoiden vertailu päätavoitetta ajatellen olisi helpompaa, ja samalla yleiskuva oliopohjaisen väliohjelmiston arkkitehtuurista selkiytyisi. **Käytännön osuudessa** taas pyrittiin saamaan tietoa siitä, miten väliohjelmistoja sovelletaan ja käytetään todellisessa kehitystyössä. Valmiille toteutuksille tehtyjen **testien** tarkoituksena oli puolestaan varmistaa ohjelmistolle asetettujen vaatimusten täyttyminen sekä saada kerättyä konkreettista vertailutietoa väliohjelmistojen suoritus- ja virheenkäsittelykyvyistä. Lisäksi testeillä kerättiin aineistoa väliohjelmistoja käyttävien sovellusten asentamiseen liittyvistä seikoista.

3 Hajautetut järjestelmät ja oliomalli

Hajautettu järjestelmä on Coulouriksen määritelmän [Cou01, sivu 1] mukaan systeemi, jossa eri palvelimille sijoitetut komponentit kommunikoivat ja koordinoivat toimiaan ainoastaan lähettämällä viestejä toisilleen. Määrittely on laaja kattaen mm. FTP- ja Telnet-protokollat, Intranetin, Internetin ja ns. läsnälaskennan (engl. *ubiquitous computing*). Läsnälaskenta (tai läsnä-äly) on tulevaisuuden teknologiaa, jolla tarkoitetaan kaikkialla olevien ja meitä ympäröivien pienitehoistenkin tietokonelaitteiden piilotettua laskennallista hyödyntämistä.

Vaikka näiden esimerkkien hajautustarpeet vaihtelevat suuresti, pätee niille kaikille samat (vaikkakin eri mittasuhteissa) olevat vaatimukset. Tässä luvussa esitellään näitä vaatimuksia, selvitetään, millaisia arkkitehtuureja hajautetuille ympäristöille on olemassa, sekä kerrotaan, kuinka väliohjelmistot sijoittuvat tähän kokonaisuuteen. Luvun päälähde on [Cou01, sivut 16–27].

3.1 Olennaisimpia vaatimuksia

Hajautetulle järjestelmälle asetettavat vaatimukset eivät pääpiirteissään poikkea yleisistä ohjelmistoille asetetuista vaatimuksista, mutta hajautus synnyttää osaan vaatimuksista lisäpiirteitä. Esimerkiksi turvallisuuteen liittyvät vaatimukset korostuvat ja laajenevat paikallisesti suoritettaviin ohjelmiin verrattuna. Luvuissa 3.1.1–3.1.5 on lyhyesti kerrottu tärkeimmistä hajautetuille järjestelmille asetetuista vaatimuksista.

3.1.1 Skaalautuvuus

Hyvän hajautetun järjestelmän täytyy kyetä skaalautumaan eli mukautumaan resurssien ja käyttäjämäärien lisääntyessä. Tunnettu ääriesimerkki hajautetun ympäristön kasvusta on Internettiin kytkettyjen tietokoneiden määrä, joka ilmenee taulukosta 1.

Ajankohta	Tietokoneita	WWW-palvelimia
1979, joulukuu	188	0
1989, heinäkuu	130 000	0
1999, heinäkuu	56 218 000	5 560 866

Taulukko 1. Tietokoneiden määrä Internetissä [Cou01, sivu 19].

Skaalautuvan järjestelmän algoritmit on ensinnäkin suunniteltava sellaisiksi, **etteivät niiden aika- ja muistivaatimukset ole eksponentiaalisia** järjestelmän resurssien tai käyttäjien määrän nähden, vaan lähellä lineaarista. Käyttäjämäärän tuplaantuminen ei siis saa kolminkertaistaa järjestelmän muistintarvetta.

Toiseksi skaalautuvan järjestelmän omat **sisäiset resurssit ei tule olla rajoitettuja** johonkin tiettyyn määrään yksikköjä. Esimerkkinä voisi kuvitella järjestelmää, jossa eri koneiden prosessit tunnistavat toisensa 8-bittisen tunnuksen avulla. Jossain vaiheessa järjestelmän kasvaessa prosessien määrä ylittää 8-bittisen luvun maksimiarvon (255) ja järjestelmä joko lakkaa toimimasta tai laajentaminen tulee mahdottomaksi. Tulevaa kasvua on tietenkin vaikea ennustaa ja valmistautuminen ylisuureen kasvuun voi toisaalta laskea järjestelmän tehokkuutta.

Laajennettavan järjestelmän **tehon kasvu suhteessa kuluihin** fyysisten laitteiden osalta täytyy myös olla hyvä. Ideaalitilanteessa toisen palvelinkoneen hankkiminen kaksinkertaistaisi järjestelmän tehokkuuden. Tähän on kuitenkin hyvin vaikea päästä, sillä usein palvelimet eivät ole täysin riippumattomia toisistaan ja ne joutuvat käyttämään jotakin yhteistä resurssia (kuten levypalvelinta, verkkolinjaa tai tietokantaa) jaetusti. Tällöin resurssin yhteiskäytön synkronointi ja koordinointi alkaa itsessään kuluttaa järjestelmän tehoja ja muodostua pullonkaulaksi järjestelmälle.

3.1.2 Avoimuus

Monet aiemmista hajautetuista järjestelmistä olivat hyvin monimutkaisia ja käyttivät lukuisia eri protokollia ja rajapintoja viestintäänsä. Jotta ohjelmistojen kehittäjät voivat toteuttaa näihin järjestelmiin uusia yhteensopivia ohjelmistoja, täytyy järjestelmien rajapintojen olla tarkasti kuvattu ja dokumentoitu.

Dokumenttien julkistaminen joko epävirallisesti tai standardoinnin kautta, on lähtökohta järjestelmän avoimuudelle. Järjestelmä voi olla avoin vasta silloin, kun sen palvelut ja rajapinnat on kuvattu niin tarkalla tasolla, että ulkopuoliset kehittäjät pystyvät lisäämään uusia sovelluksia tai laitteita järjestelmään. Hyvin toteutettu järjestelmän avoimuus sallii myös sovellusten toimintaympäristöjen poiketa toisistaan.

Yksi esimerkki avoimesta järjestelmästä on CORBA-arkkitehtuuri, jonka kaikki palvelut ja rajapinnat on tarkasti kuvattu vapaasti saatavilla olevissa teknisissä dokumenteissa. Näiden dokumenttien avulla on kehitetty useita spesifikaatioita noudattavia CORBA-toteutuksia eri käyttöjärjestelmille ja ympäristöille. CORBA-standardista kerrotaan tarkemmin luvussa 5.1.2.

3.1.3 Tuntumattomuus

Hajautetun järjestelmän tuntumattomuus[1] (engl. *distribution transparency*) tarkoittaa yksittäisten järjestelmän osien salaamista käyttäjältä tai sovellusohjelmoijalta niin, että hänelle järjestelmä näyttäytyy yhtenä kokonaisuutena, eikä joukkona erillisiä riippumattomia komponentteja. Tuntumattomuuden toteuttaminen vaatii täten tiettyjen järjestelmäosien piilottamista siten, että hajautetut kutsut vaikuttavat käyttäjälle paikallisilta kutsuilta.

Tuntumattomuutta on useita eri tyyppejä ja yhden luokittelutavan tarjoaa **ODP-viitemalli** (*Open Distributed Processing)*. Se esittelee kahdeksan eri tuntumattomuuden lajia, joista tärkeimpiä hajautetulta järjestelmältä vaadittavia ovat saanti- ja paikkatuntumattomuus

[1] Termistä näkee joskus käytettävän myös synonyymia ”läpinäkyvyys”.

(engl. *access* ja *location transparency)*. Ensimmäinen mahdollistaa resurssien käytön identtisellä tavalla riippumatta siitä, ovatko ne paikallisia vai ei. Jälkimmäinen puolestaan mahdollistaa resurssien käytön ilman tietoa niiden sijainnista. Muita tuntumattomuuden tyyppejä ovat mm. vikatuntumattomuus, poikkeustuntumattomuus ja toisinnustuntumattomuus [ISO96, sivut 45–48].

3.1.4 Vikasietoisuus

Laadukkainkin järjestelmä saattaa joko itse aiheuttaa virhetilanteen tai joutua kärsimään siitä riippumattomista ulkopuolisista virhetilanteista. Tällöin ohjelma joko lakkaa toimimasta, suorittaa tehtävänsä vain osittaisesti tai virheellisesti. Erityisesti hajautetuille ympäristöille on tunnusomaista, että vain osa prosesseista ajautuu virhetilanteeseen, kun loput jatkavat toimintaansa normaalisti. Tällaisissa tilanteissa virheiden käsittely on entistä hankalampaa, mutta sitäkin tärkeämpää. Vikasietoisen järjestelmän tulisi kyetä käsittelemään kaikki viestiliikenteeseen liittyvät virheet yhtenäisellä tavalla ja jatkamaan toimintaansa virhetilanteen korjaannuttua.

Hajautetussa ympäristössä tapahtuvien virheiden käsittelyssä on hyvä erottaa toisistaan prosesseissa tapahtuvat virheet etäviestien välityksessä tapahtuvista verkkovirheistä. Nämä virheet voidaan vielä edelleen luokitella laiminlyönteihin, satunnaisvirheisiin ja aikavirheisiin. **Laiminlyöntivirheiksi** (engl. *omission failure*) kutsutaan tilanteita, joissa prosessi tai verkko jättää suorittamatta sille annetun tehtävän. **Satunnaisvirheitä** (engl. *arbitrary failure)* taas voi syntyä esim. prosessin asettaessa odottamattomia arvoja siirrettäviin viesteihin, verkon kadottaessa tai toistaessa viestejä taikka viestien korruptoituessa siirron aikana. Odotettujen virheiden havaitseminen sekä niistä toipuminen on yleensä helpompaa kuin satunnaisten virheiden kohdalla. Hyvin suunnitellulla virheenkäsittelyllä satunnaiset virheet voidaan kuitenkin muuntaa (engl. *failure masking)* odotetuiksi virheiksi ja näin yksinkertaistaa toimintaa virhetilanteissa. Kolmatta virheenlajia eli **aikavirheitä** voi tapahtua asynkronisten kutsujen yhteydessä, kun kutsu ei onnistu tietyn aikavälin sisällä tai prosessi ei kykene suoriutumaan riittävästä määrästä operaatioita annetussa ajassa.

3.1.5 Turvallisuus

Hajautetut ympäristöt asettavat ohjelmistoille uudenlaisia turvallisuusvaatimuksia. Koska järjestelmän viestit liikkuvat usein julkisessa verkossa, on niihin mahdollista päästä käsiksi melko vaivattomasti niin sanottuja ”sniffer”-ohjelmia käyttämällä. Ulkopuolinen luvaton tunkeutuja voi yrittää esim. **kopioida** viestejä talteen (engl. *eavesdropping)*, **muuttaa** matkalla viestien sisältöä (engl. *message tampering)*, **uudelleenlähettää** aiemmin hankittuja viestejä (engl. *replaying)* tai yrittää **tukkia palvelinta** viestien massalähetyksillä (engl. *denial of service).*

Useisiin edellä mainittuihin uhkiin voidaan varustautua vaatimalla hajautetulta järjestelmältä ensinnäkin kykyä **tunnistaa** (engl. *identification)* luotettavasti palvelua tai rajapintaa käyttävä käyttäjä. Toiseksi järjestelmään täytyy pystyä määrittämään käyttäjäkohtaisia oikeuksia. Tällöin järjestelmä voi **varmistaa** (engl. *authorization),* mitkä toiminnot ja operaatiot ovat sallittuja kyseiselle käyttäjälle. Hyvän järjestelmän täytyy myös **kirjata suorittamansa toiminnot** (engl. *auditing)* lokiin, jota tarkkailemalla voidaan huomata mahdolliset hyökkäysyritykset tai vieraat käyttäjät. Jos välitettävät viestit sisältävät arkaluonteista aineistoa, täytyy ne pystyä **salaamaan** (engl. *encrypt)* lähetyksen ajaksi. Järjestelmän pitäisi myös kyetä olemaan varma viestin lähettäjästä digitaalista allekirjoitusta (engl. *digital signature*) käyttämällä.

Hajautetun järjestelmän toteutus täytyy lisäksi olla erityisen hyvin testattu mahdollisten hyökkäysten varalta, sillä joissakin tapauksia hyökkääjät ovat hyödyntäneet järjestelmään tunkeutumisessa siihen jääneitä ohjelmointivirheitä, kuten puskurien ylivuotoa tai tarpeettomasti avoinna olevia tietoliikenneportteja. Hajautetun järjestelmän suunnittelijan täytyykin otaksua, että mahdolliset hyökkääjät tuntevat järjestelmän suojauskeinot yhtä hyvin tai jopa paremmin kuin itse kehittäjät. Parhaat suojauskeinot saavutetaankin yleensä käyttämällä julkisesti tarjolla olevia salaus- ja allekirjoitusalgoritmeja, joita on yritetty tuloksettomasti murtaa ja jotka on laajasti todettu toimiviksi.

3.2 Hajautettujen ympäristöjen arkkitehtuureja

Järjestelmän arkkitehtuurilla kuvataan sen rakennetta erillisten osakomponenttien näkökulmasta. Arkkitehtuuri käsittää komponenttien sijoittumisen järjestelmässä suhteessa muihin komponentteihin, komponentin tehtävät ja vastuut sekä vuorovaikutuksen ja kommunikoinnin muiden komponenttien kanssa. Tällaisen kuvauksen tuloksena syntyy arkkitehtuurimalli.

Hajautetuissa ympäristöissä arkkitehtuurin komponentit ovat yleensä prosesseja. Tietyssä arkkitehtuurimallissa esiintyvillä prosesseilla on erilaisia rooleja riippuen siitä, millä tavalla prosessi kommunikoi mallin muiden prosessien kanssa. Yleinen tapa luokitella prosessit on seuraavanlainen [Cou01, sivut 34–35]:

Asiakasprosessit välittävät kutsuja palvelinprosesseille ja käsittelevät paluuarvoina saatuja tuloksia.

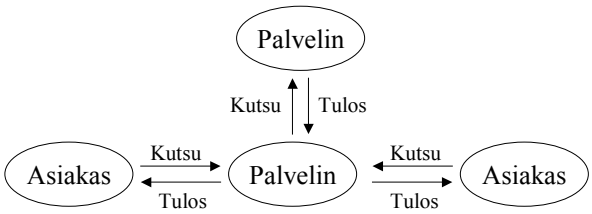
Palvelinprosessit ottavat vastaan kutsuja asiakasprosesseilta ja palauttavat mahdollisen tuloksen.

Vertaisprosessit ovat yhtä lailla asiakasprosesseja kuin palvelinprosesseja.

Luvuissa 3.2.1–3.2.4 esitellään lyhyesti joitakin ohjelmistotekniikassa vakiintuneita arkkitehtuurimalleja. Pääasiallisena lähteenä on [Cou01, luku 2.2].

3.2.1 Asiakas-palvelinmalli

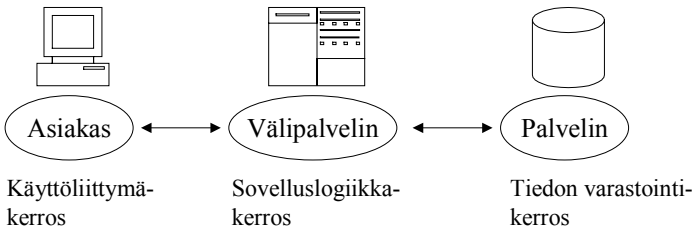
Yleisin ja perinteisin arkkitehtuurimalli hajautetuissa ympäristöissä on asiakas-palvelinmalli, jossa yksi tai useampi palvelinprosessi ottaa kutsuja asiakasprosesseilta ja palauttaa kutsun tuloksen. Malli esitetään kuvassa 1.



Kuva 1. Asiakas-palvelinmalli.

Huomattavaa mallissa on, että palvelin voi myös toimia asiakkaan roolissa ja olla yhteydessä muihin palvelinprosesseihin.

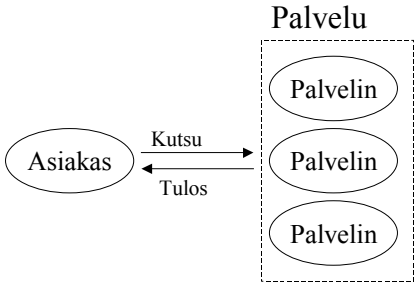
Erikoistapauksia asiakas-palvelinmalleista ovat **monitasoiset järjestelmät**, joita käytetään hyvän skaalautuvuuden saavuttamiseen palvelintasolla. Kaksitasoinen järjestelmä (engl. *two-tier architecture*) vastaa perinteistä mallia, jossa tiedon varastointi on eriytetty omalle palvelimelleen. Tämän mallin on kuitenkin syrjäyttämässä kolmitasoinen järjestelmä (engl. *three-tier architecture*), jossa on eriytetty eri palvelinprosesseille tiedon esittäminen, tiedon käsittely sekä tiedon varastointi (katso kuva 2). Tällainen jako estää asiakaspalvelinta pääsemästä suoraan käsiksi tietovarastoihin ja lisää näin järjestelmän turvallisuutta. Myös yksittäisten järjestelmän osien kehittäminen ja korvaaminen on helpompaa, kunhan prosessien ulospäin näkyvät rajapinnat pysyvät muuttumattomina.



Kuva 2. Kolmitasoinen järjestelmä.

3.2.2 Monipalvelinmalli

Useat palvelimet voivat yhdessä muodostaa palvelun, jota asiakasprosessit käyttävät samaan tapaan kuin yhtä palvelinprosessia. Mallia havainnollistetaan kuvassa 3.

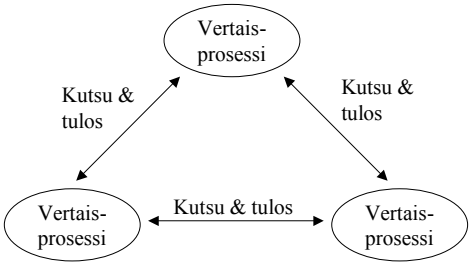


Kuva 3. Monipalvelinmalli.

Monipalvelinmallia käytetään usein silloin, kun asiakasprosesseja on hyvin suuri määrä eikä yksi palvelin pystyisi tarjoamaan riittävää tehokkuutta. Mallilla voidaan myös varmistaa palvelun toimivuus mahdollisen virhetilanteen ja palvelimen alasajon sattuessa. Internetin DNS-palvelu on esimerkki monipalvelinmallista. Sen useat eri puolilla maailmaa sijaitsevat palvelimet muodostavat yhdessä palvelun, joka pystyy palvelemaan tehokkaasti tuhansia asiakkaita, eikä kärsi yhden tai useammankaan palvelimen alasajosta.

3.2.3 Vertaisverkkomalli

Jos järjestelmän kaikki prosessit voivat toimia niin asiakkaina kuin palveliminakin, puhutaan vertaisverkkomallista (engl. *peer-to-peer*), joka on esitetty kuvassa 4.



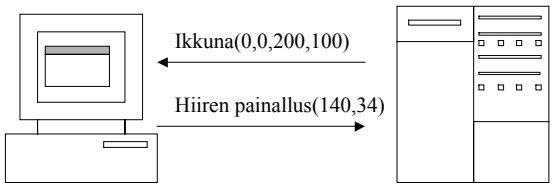
Kuva 4. Vertaisverkkomalli.

Vertaisverkkomalli sopii tilanteisiin, joissa järjestelmän jokaisen prosessin on saatava mahdollisimman nopeasti tieto muissa prosesseissa tapahtuneista muutoksista. Esimerkkinä voisi ajatella usean käyttäjän reaaliaikaista keskusteluohjelmaa, jossa kunkin käyttäjän syöttämä teksti välittyy välittömästi kaikkien muiden käyttäjien nähtäväksi. Tällaisen nopean reagoinnin saavuttamiseksi vertaisprosessit käyttävät usein kommunikointiin ns. ”rekisteröityjä tapahtumia” (engl. *events)*, joiden avulla tieto muutoksista leviää prosessien kesken. Rekisteröidyt tapahtumat ovat asynkronisesti lähetettyjä viestejä, joita vertaisprosessit lähettävät toisille vertaisprosesseille jäämättä odottamaan vastausta.

3.2.4 Mallien variaatioita

Luvuissa 3.2.1–3.2.3 esiteltyjen arkkitehtuurimallien lisäksi on olemassa joukko malleja, jotka ovat laajennuksia edellä esiteltyihin. Lähinnä nämä muistuttavat perinteistä asiakas-palvelinmallia. **Kevyen asiakkaan malliksi** (engl. *thin-client*) kutsutaan ratkaisua, jossa asiakaskoneella on käytettävissä graafinen käyttöliittymä, mutta varsinaisten ohjelmien suoritus tapahtuu palvelinkoneella.

Palvelinkone välittää tällöin ohjelmien tulosteet asiakaskoneelle, joka esittää ne graafisessa muodossa käyttäjälle. Käyttäjän hiirellä tai näppäimistöllä antamat mahdolliset syötteet asiakaskone välittää edelleen takaisin palvelinkoneelle. Tällaisessa ratkaisussa asiakaskoneen ei tarvitse olla kovin suorituskykyinen, mutta palvelinkoneen täytyy sitä vastoin pystyä suorittamaan useita prosesseja rinnakkain asiakkaita palvellessaan. Esimerkkeinä kevyen asiakkaan järjestelmistä voidaan mainita X11-ikkunointijärjestelmä, työaseman etäkäytön mahdollistava VNC-sovellus (*Virtual Network Computer)* sekä useat WWW-selaimella käytettävät sovellukset.



Kuva 5. Kevyen asiakkaan malli

Toinen mielenkiintoinen arkkitehtuurimalli liittyy ns. **siirrettävän koodin** (engl. *mobile code)* mahdollisuuksiin. Siirrettävällä koodilla tarkoitetaan ohjelmaa tai ohjelmakomponenttia, joka voidaan siirtää palvelimelta asiakkaalle ja suorittaa tämän jälkeen paikallisena asiakkaan koneella. Siirto voi tapahtua lähes huomaamattomasti, eikä käyttäjän tarvitse huolehtia ajettavan ohjelman asennuksesta tai käyttöönotosta. Etuna tällaisessa tekniikassa on hyvä suorituskyky asiakkaan koneella, koska verkkoliikenteen nopeus ei vaikuta ohjelman suoritukseen.

Java-kielen tarjoamat sovelmat (engl. *applet*) sekä ActiveX-komponentit ovat esimerkkejä siirrettävästä koodista. Niitä on mahdollista sijoittaa WWW-sivuille, jolloin selain osaa automaattisesti siirtää komponentin käyttäjän koneelle ja aloittaa suorittamaan sitä. Myös yleistymässä olevat Grid-laskentaverkot hyödyntävät siirrettävää koodia toiminnassaan.

3.3 Hajautetun oliomallin käsitteitä

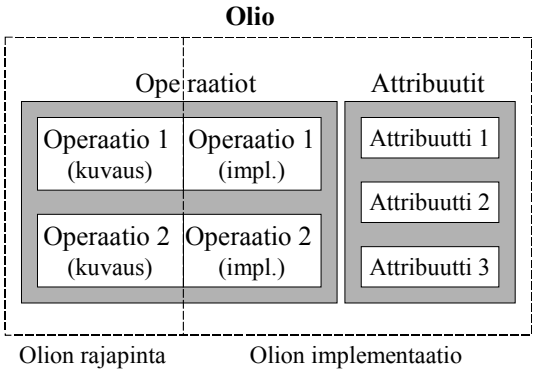
Ohjelmistosuunnittelussa tapahtui 1980-luvun loppupuolella suuri muutos, kun perinteisen moduuli-ajattelulle pohjautuvan suunnittelun rinnalla otettiin käyttöön oliosuuntautunut ohjelmistosuunnittelu. Tämä vaati luopumista proseduraalisesta ajattelutavasta, jossa erotettiin tieto ja sitä käsittelevät algoritmit. Oliopohjaisessa lähestymistavassa kumpikin sidottiin yhteen samaksi entiteetiksi, jota kutsutaan olioksi.

Tällä hetkellä suuri osa ohjelmistoyrityksistä käyttää tai on siirtymässä käyttämään oliopohjaisia ohjelmointi- ja suunnittelutapoja ohjelmistoprojekteissaan. Menetelmän tarjoamat edut, kuten uudelleenkäytettävyys, hallittavuus ja skaalautuvuus, ovat alkaneet tulla entistä tärkeämmäksi ohjelmien yhä monimutkaistuessa.

Luvussa tutustutaan aluksi lyhyesti perinteiseen oliomalliin. Tämän jälkeen tarkastellaan sen käsitteitä hajautetun oliomallin näkökulmasta. Luvun loppupuolella selvitetään, millaisilla tekniikoilla olioita luodaan, kuinka niihin viitataan ja millä tavalla ne tuhotaan hajautettuun oliomalliin perustuvissa ympäristöissä. Luvun pääasiallinen lähde on [Cou01, luvut 5.1–5.2]

3.3.1 Oliomallin käsitteitä

Ennen perinteisen oliomallin kuvausta on hyvä selventää muutamia käsitteitä, jotka esiintyvät oliomallissa ja myös myöhemmässä osassa tutkielmaa. **Oliolla** tarkoitetaan yleisesti mallia jostakin kokonaisuudesta (engl. *entity)*. Oliota kuvaa sekä sen tila (engl. *state)* että ulkoinen toimintapatapa (engl. *behavior)*. Tilan määrää olion **attribuuttien** arvot tietyllä hetkellä, kun taas toimintapa kertoo, mitä toimintoja eli **operaatioita** olio voi suorittaa kyseisellä hetkellä. **Rajapinnaksi** (engl. *interface)* sanotaan olion ulospäin näkyvää kuvausta olion operaatioista. **Luokka** (engl. *class)* on olion rajapinnan ja toteutuksen kuvaus, jonka pohjalta olio voidaan luoda. **Instanssi** on puolestaan luokan pohjalta luotu olio eli luokan ilmentymä. Kuvassa 6 on selvennetty näitä olioon liittyviä käsitteitä.



Kuva 6. Olion abstrakti malli.

Usein ohjelmointikielissä olion operaatioita kutsutaan metodeiksi ja olion attribuutteja jäsenmuuttujiksi. Näitä termejä tullaan käyttämään myös tutkielman loppuosassa [Cou01, luku 5.2.1].

3.3.2 Perinteinen oliomalli

Perinteisen oliomallin mukaan toteutettu ohjelma sisältää aina yhden tai useamman olion. Oliot voivat omistaa vaihtelevan määrän jäsenmuuttujia. Jäsenmuuttujien arvoja voidaan muuttaa ja lukea olion metodien avulla. Ohjelmointikielestä riippuen voidaan sallia myös arvojen sijoittaminen suoraan jäsenmuuttujiin, mutta tämä rikkoo osittain tiedon piilottamisen periaatetta. Tiedon piilottamisella tarkoitetaan sitä, että osa olion metodeista ja ominaisuuksista on yksityisiä ja vain olion omassa sisäisessä käytössä. Osa taas julkisia ja käytettävissä olion ulkopuolelta. Tiukkaa tiedon piilottamista noudatettaessa vain olion metodit ovat julkisia ja kaikki jäsenmuuttujat yksityisiä.

Olion metodien kuvaukset muodostavat olion rajapinnan, jonka avulla oliota käytetään ulkoapäin. Rajapinta kuvaa siis olion metodit, mutta ei niiden toteutusta. Metodin kuvaus sisältää vähintään metodin nimen sekä tyyppitietoa metodille välitettävistä parametreista ja paluuarvosta. Rajapintojen avulla olion sisäistä toteutusta on mahdollista muuttaa ilman, että ulospäin näkyvä rajapinta muuttuu. Oliot keskustelevat keskenään toistensa rajapintojen välityksellä.

Tärkeä osa perinteistä oliomallia on myös mahdollisuus kehittää uusia olioita perimällä olion luokka jo olemassa olevasta luokasta. Uusi luokka pääsee näin käsiksi perityn luokan operaatioihin, ja voi joko julkaista ne rajapinnassaan sellaisenaan tai kätkeä niitä valikoidusti hyödyntäen operaatioita ainoastaan omassa sisäisessä toteutuksessaan. Luokka voi myös periytyä useammasta kuin yhdestä luokasta, jolloin puhutaan moniperinnästä.

3.3.3 Hajautettu oliomalli

Hajautettu oliomalli perustuu luvussa 3.3.2 esitellyn perinteisen oliomallin käsitteisiin. Kun perinteistä oliomallia laajennetaan hajautettuihin järjestelmiin, joudutaan ottamaan käyttöön joitakin uusia käsitteitä tai uudelleen määrittelemään jo olemassa olevia. Tärkeimmät näistä käsitteistä esitellään seuraavissa kappaleissa lähteen [Cou01, luku 5.2.2] mukaisesti.

Olioreferenssi (engl. *object reference)* on tapa viitata olioon epäsuorasti. Referenssi ei ole suora muistiosoitin, vaan olion tunniste. Sen avulla olion metodeja voidaan kutsua niin paikallisessa kuin hajautetussakin järjestelmässä. Referenssejä on mahdollista kopioida ja välittää ilman, että viittauksen kohteena olevaa oliota kopioidaan tai siirretään. Hajautetuissa ympäristöissä referenssien tärkeys korostuu entisestään. Jos olioreferenssi viittaa toisessa prosessissa sijaitsevaan olioon, käytetään usein termiä ”etäolioreferenssi”.

Etärajapinta (engl. *remote interface)* kuvaa ne olion metodit, joita on mahdollista suorittaa hajautetusti. Usein etärajapinta vastaa olion paikallista rajapintaa, mutta joissakin tapauksissa voi olla tarpeen erottaa oma osajoukko etärajapinnaksi olion kokonaisrajapinnasta. Tärkeä merkitys etärajapintojen määrittelyllä on myös generoitaessa (usein automaattisesti omalla työkalulla) rutiineja kutsujen sarjallistamista varten. Sarjallistaminen on toimenpide, jossa kutsu sekä sen parametrit pakataan verkon yli siirrettävissä olevaan muotoon.

Tapahtuma (engl. *event*) on laajahko käsite, jolla tarkoitetaan olioon kohdistuvaa kutsua, sen vaikutusta olion tilaan sekä mahdollista paluuarvon palautusta. Tapahtuma voi synnyttää uusia tapahtumia, jos kutsutussa metodissa kutsutaan vuorostaan jonkin toisen olion metodeja. Hajautetuissa ympäristöissä tapahtumat voivat levitä eri prosesseihin ja koneisiin olioiden sijainnista riippuen.

Poikkeukset (engl. *exception)* antavat mahdollisuuden käsitellä ohjelman suorituksen aikana syntyviä virheitä yhtenäisellä tavalla. Mahdollisissa virhetilanteissa ei palauteta kutsuvalle aliohjelmalle virheen ilmaisevaa paluuarvoa, vaan sen sijaan aiheutetaan (engl. *throw)* virhettä vastaava poikkeus. Poikkeus otetaan kiinni (engl. *catch)* tämän jälkeen sille määrätyssä paikassa kootusti, eikä ohjelman suoritus palaa virheen jälkeiseen kohtaan. Tällainen virheenkäsittelytapa yksinkertaistaa ohjelman rakennetta sekä vähentää mahdollisuuksia virheiden huomioimatta jättämiseen. Hajautetuissa järjestelmissä poikkeukset voivat välittyä palvelinprosessilta asiakasprosesseille, jonka vastuulla on niiden käsittely.

3.4 Väliohjelmistot

Asiakas-palvelinarkkitehtuurien yleistyessä 1980-luvulla alkoivat myös tekniset vaikeudet tulla näkyviin. Heterogeenisissä ympäristöissä oli vaikea sovittaa yhteen eri sovellusten tietovarastoja ja tiedon käsittelylogiikkaa. Erilaisia ohjelmistorajapintoja syntyi lukuisia ja niiden kehittäminen oli yrityksille sekä vaikeaa että aikaa vievää. Järjestelmät olivat usein ns. suljettuja järjestelmiä, joiden yhteensopivuus muiden järjestelmien kanssa oli heikkoa. Vuosikymmenen loppupuolella vaatimukset avoimiin järjestelmiin ja standardeihin alkoivat kasvaa integraatiotarpeiden myötä. Jotain uutta kaivattiin ratkaisuksi.

3.4.1 Väliohjelmiston tehtävät

Ratkaisuksi tilanteeseen muodostui 1990-luvun alussa hajautetut väliohjelmistot (engl. *distributed middleware)*. Käsitteellä tarkoitetaan ohjelmistoa, joka mahdollistaa hajautetun tiedonkäsittelyn erilaisia liittymäpalveluja tarjoamalla. Liitettävyys onkin väliohjelmistojen myötä noussut jopa avoimuutta tärkeämmäksi vaatimukseksi, kun järjestelmiä yritetään yhteensovittaa keskenään.

Väliohjelmiston tehtäviin kuuluu muunnos erilaisten rajapintojen välillä sekä prosessien välinen kommunikaatio datan ja käskyjen siirtämiseksi. Lisäksi väliohjelmisto hallitsee prosessien välisiä yhteyksiä. Käytännössä väliohjelmisto piilottaa sovelluskehittäjältä matalan tason kommunikaatiorutiinit yhtenäisen rajapinnan alle.

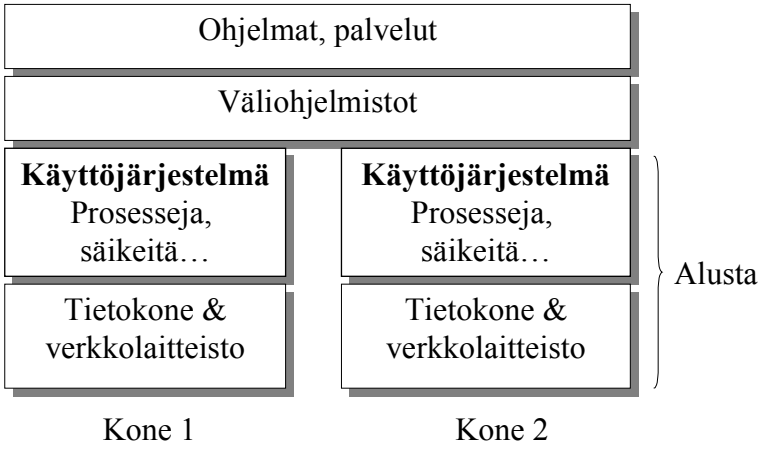
Hajautetuilla väliohjelmistoilla on myös oma arkkitehtuurinsa, eikä sitä pidä sekoittaa luvussa 3.2 esiteltyihin järjestelmäarkkitehtuureihin. Hajautetun väliohjelmiston arkkitehtuuri (katso luku 5.1) kuvaa hajautuksen toteutukseen osallistuvat komponentit, niiden vastuut sekä millä tavalla ne kommunikoivat toistensa kanssa. Komponentti voi olla esimerkiksi alijärjestelmä, ajonaikainen palvelu, luokkakirjasto, tietovarasto tai jokin muu järjestelmän erillinen osa.

Väliohjelmiston käyttö yksinkertaistaa ja samalla tehostaa useimmissa tapauksissa sovelluskehittäjien työtä tarjoamalla yhden yhteisen rajapinnan sovelluksille. Näin on varsinkin, jos tarve on saada useampia sovelluksia toimimaan keskenään. Kahden sovelluksen kesken väliohjelmiston käyttö ei kuitenkaan aina ole perusteltua, vaan se

saattaa jopa hankaloittaa toteutusta. Lisäksi väliohjelmistot tarjoavat usein valmiita työkaluja hajautuksen konfigurointiin sekä monitorointiin, jolloin näitä sovelluksia ei tarvitse erikseen toteuttaa.

3.4.2 Väliohjelmiston ja käyttöjärjestelmän roolit

Väliohjelmisto ei voi toimia ilman käyttöjärjestelmän tukea. Käyttöjärjestelmä puolestaan sijaitsee laitetason päällä muodostaen yhdessä kokonaisuuden, jota kutsutaan alustaksi (engl. *platform)*. Kuva 7 havainnollistaa näitä järjestelmäkerroksia.



Kuva 7. Järjestelmäkerrokset [Cou01, sivu 210].

Käyttöjärjestelmän tärkein tehtävä on abstrahoida tarjoamansa erityyppiset resurssit siten, että niiden käyttö tapahtuu loogisen ja yhtenäisen rajapinnan kautta. Tämä tapahtuu piilottamalla eri resurssien ja laitteiden käytön monimutkaisuus yhdenmukaisten ohjelmointirajapintojen alle. Käyttöjärjestelmän täytyy myös huolehtia siitä, että eri prosessit voivat käyttää samaa resurssia yhtäaikaisesti ilman erityistä synkronointia prosessien puolella. Resurssien käyttö tulee myös olla suojattua. Prosessi ei saa pystyä muuttamaan tai tuhoamaan resursseja tavalla, johon sille ei ole myönnetty oikeuksia.

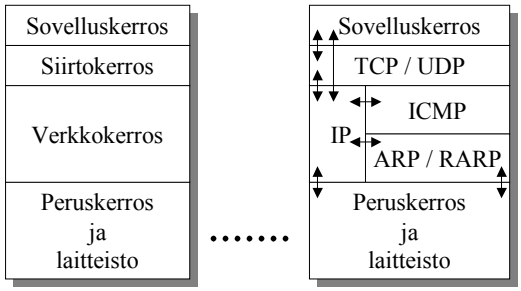
Yleisimmät tällä hetkellä käytössä olevista käyttöjärjestelmistä ovat rakenteeltaan niin kutsuttuja **verkkokäyttöjärjestelmiä** (engl. *network operating system*), joissa verkkotuki on sisäänrakennettuna käyttöjärjestelmään. Käyttöjärjestelmän prosessien ja resurssien hallinta on kuitenkin konekohtaista. Todellisissa **hajautetuissa käyttöjärjestelmissä**

(engl. *distributed OS*) verkkoon liitettyjen koneiden prosessit ja resurssit ovat täydellisesti jaossa ja koko usean koneen järjestelmä näyttäytyy yhtenä järjestelmänä, jossa ei tehdä eroa paikallisten ja ulkoisten resurssien välillä. Tällaisia käyttöjärjestelmiä ei kuitenkaan vielä ole laajassa käytössä, sillä niiden yleistymistä hidastaa mm. käyttäjien mieltymys oman koneensa täydelliseen kontrollointiin sekä turvallisuusnäkökohdat.

4 Väliohjelmistojen protokollia

Useimmat oliopohjaiset hajautusmenetelmät turvautuvat jo aiemmin kehitettyihin ja toimiviksi todettuihin tekniikoihin siirtäessään informaatiota verkon yli. Siirto tapahtuu tällöin käyttämällä jotakin tunnettua verkkoprotokollaa, joka määrittelee lähetettävien viestien rakenteen ja formaatin sekä säännöt viestien käsittelyyn.[2]

Yhdessä protokollat muodostavat ns. protokollapinoja, joissa alemman tason protokolla tarjoaa aina rajapinnan ylemmän tason protokollalle. Tunnetuin tällaisten tasojen määrittely on OSI-malli *(Open Systems Interconnect)*, jossa erilaisia tasoja on seitsemän. Internet-ympäristöissä on kuitenkin vakiinnuttanut paikkansa OSI-mallia yksinkertaisempi TCP/IP-protokollamalli, joka rakentuu neljästä eri tasosta. Tasot on kuvattu kuvassa 8.



Kuva 8. TCP/IP-protokollapino [Mur98, sivu 13].

Tutkielman aiheena olevat hajautustekniikat sijoittuvat lähinnä sovelluskerrostasolle, mutta käyttävät alempien tasojen protokollia hajautuksen toteutukseen. Tästä syystä hajautusmenetelmissä käytetyt protokollat esitellään lyhyesti luvuissa 4.1 ja 4.2. Luvun pääasiallisia lähteitä ovat [Mur98] sekä RFC-dokumentit [Pos80], [Pos81a] ja [Pos81b].

[2] Lähde [Cou01, sivu 76] määrittelee protokollan seuraavasti: ”Protocol is a well-known set of rules and formats to be used for communication between processes in order to perform a given task.”

4.1 Verkko- ja siirtokerroksen protokollia

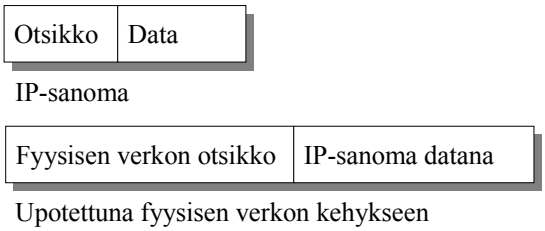
OSI-tason verkkokerroksella sijaitsevat protokollat huolehtivat tiedon matkalla tapahtuvista reitityksistä (engl. *routing*) ja kytkennöistä (engl. *switcihing*) muodostaen näin näennäisen jatkuvan reitin (engl. *virtual circuit*) siirtoa varten. Siirtokerroksen protokollat puolestaan hoitavat tiedon läpinäkyvän siirron alku- ja loppupään välillä sekä varmistavat tiedon oikeellisuuden ja vuon tarkkailun (engl. *flow control*) siirron aikana. Luvuissa 4.1.1, 4.1.2 ja 4.1.3 esitellään kolme tärkeintä näiden kahden kerroksen protokollaa.

4.1.1 IP (*Internet Protocol*)

Verkkokerroksen perustana toimii IP-protokolla. Sen tarkoituksena on kätkeä fyysinen verkkorakenne ja muodostaa sen päälle virtuaalinen verkkokerros. Protokolla on toiminnaltaan melko yksinkertainen ja sen ominaisuuksiin kuuluu, että se on **reititettävä** (engl. *routable)* ja **epäluotettava** (engl. *unreliable*). Jälkimmäisestä seuraa, että protokollan lähettämät paketit voivat kadota, monistua tai vaihtaa keskinäistä järjestystään matkalla kohteeseen protokollan huomaamatta. Protokollan voidaan myös sanoa olevan **yhteydetön** (engl. *connectionless*), sillä kohteeseen ei muodosteta erillistä yhteyttä ennen datan lähetystä.

Pakettien lähettäjät ja vastaanottajat yksilöidään **IP-osoitteen** avulla. Osoite on 32-bittinen numero, joka usein ilmoitetaan tutummassa muodossaan neljänä 8-bittisenä lukuna (esim. 194.236.2.32). Osoitteina voidaan käyttää myös selkokielisiä kirjaimista koostuvia nimiä, mutta nämä joudutaan muuntamaan ennen lähetystä nimipalvelun avulla numeromuotoon. Verkon laajentuessa on uusista 32-bittisistä osoitteista alkanut tulla pulaa, mutta yleistymässä olevan IPv6-protokollan 128-bittisten osoitteiden pitäisi ratkaista tämä ongelma.

Protokollan pienin siirtoyksikkö on **sanoma** (engl. *datagram*), joka rakentuu kahdesta osasta, kuten kuvassa 9 on esitetty. Otsikko-osa sisältää mm. sanoman lähettäjän ja vastaanottajan IP-osoitteen, sanoman pituuden sekä tarkistussumman. Dataosa voi sisältää mielivaltaista ylempien protokollatasojen tallentamaa informaatiota.



Kuva 9. IP-viestin formaatti [Mur98, sivu 48].

Sanomalle on määritelty enimmäispituudeksi 65 535 tavua. Tätä pidemmät sanomat lähetetään useassa osassa, joista jokaista käsitellään kuin yksittäistä sanomaa. Sanomat voidaan kuitenkin perillä yhdistää otsikoissa kulkeneen tiedon perusteella jälleen alkuperäiseksi sanomaksi. Jos yhdenkin osan perilletulo epäonnistuu, katsotaan koko sanoman perillemenon epäonnistuneen.

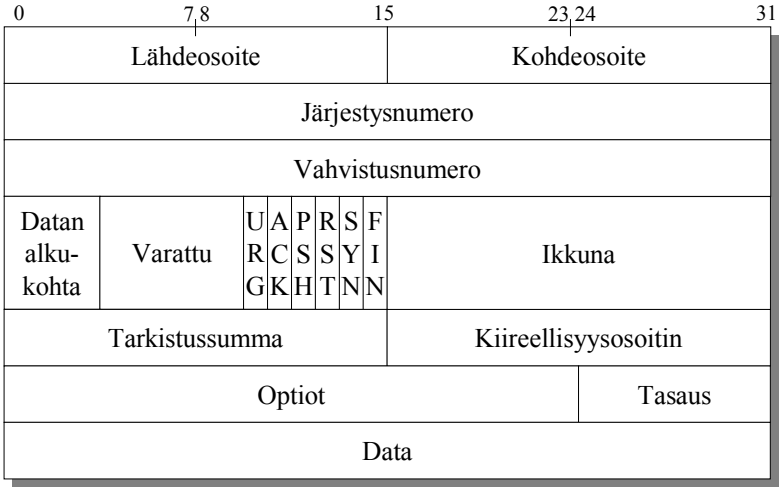
IP-protokolla on suhteellisesta yksinkertaisuudestaan huolimatta osoittautunut erittäin toimivaksi. Protokollan reititettävyys sallii vaihtoehtoisten reittien käytön viestien välityksessä. Tämä on erityisen tärkeä ominaisuus kriisitilanteissa (maanjäristykset, tulipalot, laajat sähkökatkot jne.), joissa kymmeniäkin palvelimia ja reitittimiä saattaa mennä epäkuntoon [Pos81a].

4.1.2 TCP (*Transmission Control Protocol*)

Vuoden 1978 tienoilla ensimmäistä kertaa kokeiluluonteisesti käyttöönotettu TCP oli merkittävä askel tietoverkkojen yhtenäistämiselle. Yhdysvaltain puolustusministeriön tutkimusosaston (DARPA) hankkeena alkanut ja nykyisen Internetin synnyn mahdollistanut protokolla rakennettiin luvussa 4.1.1 esitellyn IP:n päälle mahdollistamaan luotettavampi kommunikointi kuin mitä IP-protokolla pystyi tarjoamaan [Mur98].

TCP on **yhteydellinen** (engl. *connection-oriented)* siirtokerroksen protokolla, jonka merkittävimmät ominaisuudet ovat sen **luotettavuus (**engl. *reliable*) sekä riippumattomuus alempien tasojen protokollista. Protokollan peruskäsite on **paketti**. Yksi paketti on pienin tietomäärä, jonka protokolla kerralla välittää. Paketin mukana kulkee varsinaisen datan lisäksi tietoa mm. lähettäjästä, vastaanottajasta sekä paketin järjestysnumerosta.

Protokolla ei ota kantaa siihen, mitä kautta paketti kuljetetaan määränpäähän. Paketit voivat tästä johtuen saapua vastaanottajalle eri järjestyksessä kuin ne lähetettiin. Tällöin paketit voidaan järjestysnumeron avulla käsitellä vastaanottajan päässä oikeassa järjestyksessä. TCP-paketin otsikkotiedot on kokonaisuudessaan esitetty kuvassa 10.



Kuva 10. TCP-paketin otsikkotiedot.

Protokollan luotettavuus saavutetaan sillä, että paketin vastaanottajan on aina lähetettävä saapuneesta paketista kuittaus (ACK) lähettäjälle. Jos lähettäjä ei saa kuittausta tietyn ajan sisällä, lähettää se paketin uudelleen. Paketin mahdollinen rämettyminen (engl. *corruption)* siirron aikana voidaan havaita paketin mukana kulkevan tarkistussumman avulla. Jos näin käy, vastaanottaja yksinkertaisesti jättää paketin kuittaamatta ja odottaa paketin uudelleenlähetystä.

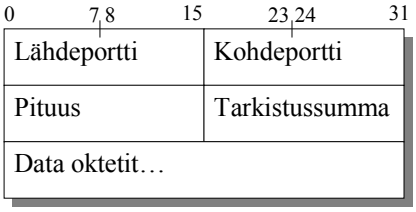
Eräs TCP:n tärkeistä ominaisuuksista on sen kyky muodostaa useita erillisiä yhteyksiä (engl. *multiplexing)* kahden koneen välille ns. **porttien** avulla. Portti on tarkenne koneen IP-osoitteeseen ja mahdollistaa eri sovellusten välisten yhteyksien identifioinnin. Sovelluksen halutessa käyttää tiettyä kohdekoneen palvelua, ottaa se yhteyden kohdekoneen palvelua vastaavaan porttiin. Porttinumerot ovat 16-bittisiä tunnuksia, joista osa on kiinteästi varattu vakiintuneille palveluille. Esimerkiksi sähköpostin lähetykseen käytetty SMTP-protokolla toimii aina portin 25 kautta.

TCP sisältää myös lähetysvuon **kontrollointitekniikan**, jonka avulla lähettäjä pystyy säätelemään pakettien lähetystiheyttä. Tekniikka toimii siten, että vastaanottaja kertoo jokaisen kuittauksensa yhteydessä pakettien määrän, joita se pystyy vielä vastaanottamaan viimeiseksi tulleen paketin jälkeen. Lähettäjä voi tämän tiedon avulla säätää omaa lähetettävien pakettien määrää. Menetelmä tunnetaan ikkuna-mekanismina (engl. *windowing)* ja tarkemman kuvauksen tästä tekniikasta voi löytää lähteestä [Pos81b].

4.1.3 UDP (*User Datagram Protocol*)

Yksinkertaisemman rajapinnan IP-protokollan käyttöön verrattuna TCP-protokollaan tarjoaa UDP. Protokolla on yhteydetön, eikä se sisällä minkäänlaisia lisäyksiä virheentarkistukseen, lähetyksen kontrollointiin tai virheistä toipumiseen IP:hen verrattuna. Se kuitenkin mahdollistaa TCP:n tavoin usean erillisen yhteyden muodostamisen kahden koneen välille portteja käyttämällä.

UDP:n lähettämät sanomat ovat rakenteellisesti yksinkertaisia, kuten kuvasta 11 selviää.



Kuva 11. UDP-sanoman rakenne [Pos80].

UDP-protokollan edut TCP-protokollaan verrattuna ovat sen keveys ja pienet resurssikustannukset. Näistä seuraa kuitenkin, että virheentarkistuksen ja muiden TCP:hen sisältyvien ominaisuuksien toteutus jää UDP:tä käyttävän sovelluksen vastuulle. UDP:tä toiminnassaan hyödyntäviä protokollia ovat mm. TFTP (*Trivial File Transfer Protocol),* DNS (*Domain Name System)*, RPC (*Remote Procedure Call)* ja SMTP (*Simple Network Management Protocol)* [Pos80].

4.2 Sovelluskerroksen protokollia

Sovelluskerroksella sijaitsevat verkkopalveluita tarvitsevat sovellukset, kuten HTTP, FTP ja Telnet. Myöhemmin esiteltävät väliohjelmistot CORBA ja DCOM lasketaan myös kuuluvaksi tälle tasolle. Sovelluskerroksen protokollien vastuulla on mm. kommunikoivien osapuolten tunnistus, käyttöoikeuksien valvonta ja siirrettävän tiedon syntaksin oikeellisuuden tarkistus. Luvuissa 4.2.1 ja 4.2.2 esitellään kaksi tärkeää perusprotokollaa, jotka ovat yhä käytössä ja tärkeitä hajautettujen väliohjelmistojen näkökulmasta. Luvun pääasiallinen lähde on [Mur98].

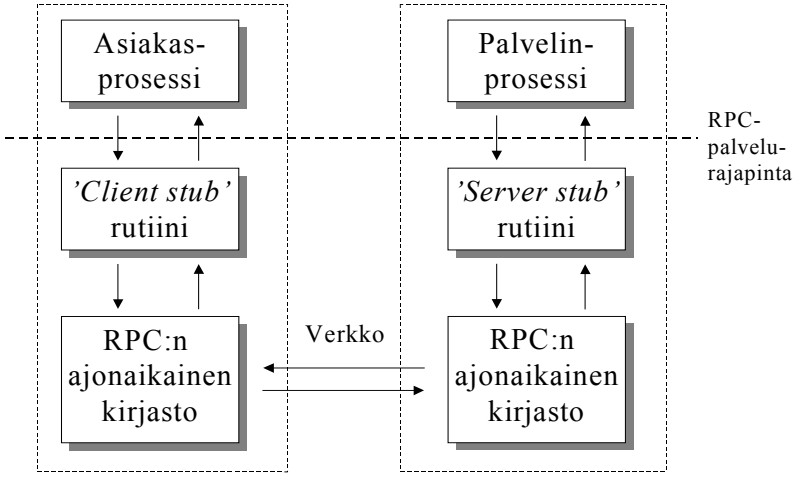
4.2.1 RPC (*Remote Procedure Call*)

RPC-protokolla tarjoaa ohjelmille mahdollisuuden suorittaa aliohjelmia toisen koneen muistiavaruudessa. Päinvastoin kuin TCP, se ei ole tarkoitettu suurien tietomäärien siirtämiseen, vaan yksittäisen kutsujen ja vastauksien välittämiseen eri koneilla toimivien prosessien välillä. Protokollan toimintaperiaate on yksinkertainen. Asiakasprosessi lähettää **kutsuviestin** toisessa koneessa toimivalle palvelinprosessille. Kutsuviesti sisältää aliohjelman nimen sekä mahdolliset parametrit. Palvelinprosessi suorittaa aliohjelman omassa työtilassaan ja palauttaa tuloksen asiakasprosessille vastausviestillä. Malli on esitetty yksinkertaistettuna kuvassa 12.

Kutsuviestit, joita RPC käyttää kommunikointiin, välitetään PDU-tietueita (*Protocol Data Unit)* käyttämällä. PDU-tietue sisältää kolme osaa:

- **Otsikko-osa** sisältää tietoa, joka riippuu käytetystä siirtokerroksen protokollasta. Otsikko-osa täytyy löytyä jokaisesta PDU-tietueesta.
- **Dataosa** sisältää kutsuun liittyvää informaatiota, kuten mahdolliset kutsulle välitettävät parametrit. Dataosa voi puuttua PDU-tietueen tyypistä riippuen.
- **Tunnistusosa** sisältää tietoa, jonka avulla kutsun turvallisuus voidaan todentaa halutessa. Sisältö vaihtelee käytetyn todennusprotokollan mukaan.

PDU-tietueen tyyppi vaihtelee käyttötarkoituksen mukaan. Tyyppi on aina tallennettu tietueen otsikko-osaan ja kolme tärkeintä tyyppiä ovat `Request`-, `Response`- ja `Fault`-tietueet. `Request`-tietuetta käytetään kutsun välittämiseen palvelinkoneelle, `Response`-tietueella välitetään tieto onnistuneesta kutsun suorituksesta takaisin asiakaskoneelle ja `Fault`-tietueella ilmoitetaan kutsussa tapahtuneesta virheestä.



Kuva 12. RPC-malli [Mur98, sivu 253].

Jotta kommunikointi RPC:n avulla toimisi, täytyy palvelinprosessin määritellä **palvelurajapinta** (engl. *service interface*). Se kuvaa aliohjelmat, joita prosessi pystyy suorittamaan. Rajapinnan kuvaus suoritetaan yleisesti Sun Microsystemsin kehittämän XDR-kielen (*eXternal Data Representation)* avulla. Kieli oli alun perin tarkoitettu vain ulkoisten muuttujatyyppien kuvaamiseen, mutta laajentui sittemmin koskemaan myös rajapintoja. Kielen syntaksi on melko alkeellinen, eikä se tue esimerkiksi rajapintojen nimeämistä, vaan niiden tunnistaminen tapahtuu yksilöllisten numerotunnusten avulla.

Jokaisella RPC-palvelulla on oma tunnuksensa, joiden jakoa valvotaan keskitetysti. Yleishyödylliseksi osoittautuville palveluilla myönnetään kiinteä numerotunnus Sun Microsystemsin toimesta. Jokaisella palvelun aliohjelmalla on lisäksi versiotunnus, jota päivitetään aliohjelman parametrilistan muuttuessa. Koska palvelun tunnus ja aliohjelman versionumero välitetään jokaisen kutsun yhteydessä, voivat asiakas- ja palvelinprosessi

näin tarkistaa käyttävänsä samaa versiota rajapinnasta. Esimerkki yksinkertaisesta XDR-määrittelytiedostosta löytyy liitteestä 3.

Kun palvelurajapinta on määritelty XDR-kieltä käyttäen, voidaan sen avulla generoida automaattisesti omaan ohjelmaan sisällytettävät aliohjelmarutiinit (engl. *stub procedures)*. Ne osaavat asiakasprosessin päässä pakata verkon yli välitettävän kutsuviestin ja palvelinprosessin päässä purkaa sen jälleen osiin. Itse viestin välitys verkon yli voi tapahtua joko TCP- tai UDP-protokollaa käyttäen.

4.2.2 NFS (*Network File System*)

Hyvänä esimerkkinä tiedostopohjaisesta hajautuksesta käy NFS-protokolla. Se on Sun Microsystemsin 1980-luvun lopulla kehittämä hajautusjärjestelmä, joka mahdollistaa tiedostojen jakamisen yhtenevällä tavalla erilaisissa laiteympäristöissä ja käyttöjärjestelmissä. Protokolla on toteutukseltaan melko yksinkertainen, ja siksi helposti sulautettavissa eri alustoille. Protokollasta on olemassa ehdotus Internet-standardiksi, jonka tarkan kuvauksen voi löytää lähteestä [Cal95].

Ennen kuin NFS-protokollalla pystytään käyttämään toisen koneen levyjärjestelmää, täytyy etäkoneen levyjärjestelmä ns. ”kiinnittää” (engl. *mount*). Kiinnittämisellä tarkoitetaan yhteyden luomista etäkoneen ja paikallisen levyjärjestelmän välille siten, että etäkoneen tiedostot näyttävät sijaitsevan paikallisissa hakemistoissa. Erityinen NFS-protokollaan kuuluva palvelinohjelma pitää kirjaa kiinnitetyistä levyjärjestelmistä ja mahdollistaa niiden hallinnan.

Hajautukseen NFS käyttää apuna luvussa 4.2.1 esiteltyä RPC-protokollaa. Kaikki tiedostoihin liittyvät operaatiot tapahtuvat 22 RPC-kutsun avulla. Tuettuja operaatioita ovat mm. seuraavat:

- tiedoston attribuuttien haku ja asetus,
- tiedoston ja hakemiston luonti sekä poisto,
- tiedoston luku ja tiedostoon kirjoitus,
- tiedoston olemassaolon tarkistus,

- tiedoston ja hakemiston käyttöoikeuksien tarkistus,
- hakemiston sisällön listaus,
- tiedostolinkkien eli tiedostoihin viittauksien käsittely,
- tiedoston ja hakemiston nimen muutos sekä
- levyjärjestelmän ominaisuuksien, kuten kokonais- ja vapaatilan kysely.

Protokollan käyttö tapahtuu etäkoneen levyjärjestelmän kiinnityksen jälkeen siten, ettei käyttäjä pysty juurikaan havaitsemaan eroa paikallisen ja etäkoneen tiedostojen välillä tiedostoja käsitellessään. Kun käyttäjä kohdistaa etäkoneen tiedostoon jonkin tiedosto-operaation, muuntaa käyttöjärjestelmä tämän NFS-protokollan komennoksi ja suorittaa sen etäkoneen levyjärjestelmässä. NFS-protokollan tärkein ominaisuus onkin piilottaa käyttäjältä paikalliskoneen ja etäkoneen levyjärjestelmien eroavaisuudet. Tämä toteuttaa tyylikkäällä tavalla hajautetuilta järjestelmiltä vaaditun paikkatuntumattomuuden (engl. *location transparency,* luku 3.1.3).

Koska NFS on tilaton protokolla (engl. *stateless protocol*), se ei pidä yllä tietoa tiedostoista komentojen suoritusten välillä. Siksi NFS-protokollan ympärille on täytynyt kehittää oma tiedostojen lukitusprotokolla NLM (*Network Lock Manager*). Protokolla mahdollistaa tiedostojen lukitsemisen yksittäisen prosessin käyttöön siten, ettei mikään muu prosessi pysty käsittelemään tiedostoa samanaikaisesti.

NFS-protokollan pohjalta on myös kehitetty tehokkuutta parantava CacheFS-protokolla (*Cache File System*), joka tallentaa etätiedostoja väliaikaisesti paikalliselle koneelle parantaen näin NFS-protokollan suorituskykyä. Yksi viimeisimmistä laajennuksista NFS-protokollaan on WebNFS-protokolla, jonka avulla NFS:n käyttö on mahdollista Internetin yli TCP-protokollaa ja URL-osoitteita hyödyntäen. WebNFS:ää kuvataan tarkemmin määrittelyissä [Cal96a] ja [Cal96b].

5 Hajautettu väliohjelmisto

Oliopohjainen väliohjelmisto joutuu toteuttamaan useita eri tehtäviä. Se pitää usein yllä varastoa olioiden rajapinnoista ja toteutuksista, hallitsee koneiden välisiä yhteyksiä, välittää olioiden kutsuja prosessien välillä sekä käsittelee erilaiset virhetilanteet mahdollisimman robustilla tavalla. Väliohjelmiston toteutuksesta riippuu, millä tavalla tehtävät on jaettu väliohjelmiston eri osien kesken.

Luvussa esitellään aluksi joitakin olemassa olevia väliohjelmistotototeutuksia. Seuraavaksi hahmotellaan tyypillinen hajautettua väliohjelmistoa käyttävä järjestelmä sekä kuvataan sen eri osat. Tämän jälkeen esitellään kaksi hajautettuihin väliohjelmistoihin liittyvää yksityiskohtaa: olioita tukevat siirtoprotokollat sekä tyyppitiedon varastointi. Luvun pääasialliset lähteet ovat [COM95, luku 2], [OMG, luku 2] sekä [SUN].

5.1 Väliohjelmistotototeutuksia

Ennen kuin tarkastelemme DCOM- ja CORBA-väliohjelmistojen toteutuksia yksityiskohtaisemmin, tutustutaan tässä luvussa kyseisiin teknologioihin yleisellä tasolla. Lisäksi luvussa esitellään JavaRMI-väliohjelmisto sekä joitakin muita vähemmän tunnettuja hajautettuja väliohjelmistoja.

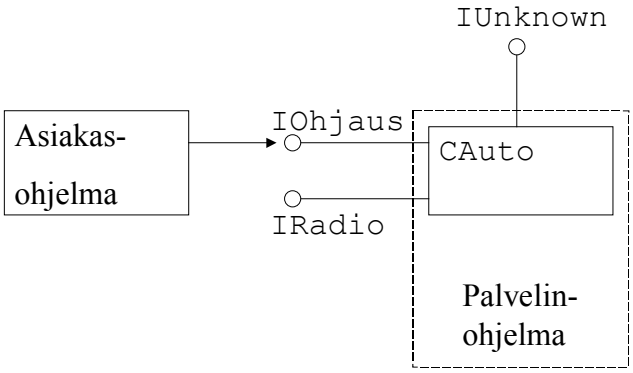
5.1.1 DCOM

Microsoftin DCOM on hajautettu väliohjelmisto, joka mahdollistaa olioiden käytön eri palvelinten välillä. DCOM on laajennus vanhempaan COM-teknologiaan, joka määrittelee binääristandardin olioiden metodien kutsumiseksi. Standardia noudattavaa oliota kutsutaan COM-komponentiksi.

COM-komponentti koostuu **luokasta** sekä yhdestä tai useammasta luokan toteuttamasta **rajapinnasta**. Jokaisella luokalla on yksilöllinen 128-bittinen luokkatunniste, josta käytetään lyhennettä CLSID (*CLasS IDentifier)*. Luokan luominen tapahtuu tämän tunnisteen avulla. **Olioksi** kutsutaan luokasta luotua ilmentymää, mutta käytännöllisistä syistä termillä ”olio” viitataan usein sekä olioluokkaan että sen mahdollisiin ilmentymiin.

COMissa olio toteuttaa aina yhden tai useamman rajapinnan. Kaikilla luokan toteuttamilla rajapinnoilla on oma rajapintatunnisteensa, jonka lyhenne on IID (*Interface IDentifier*). Asiakasohjelmat käyttävät olioita ainoastaan kutsumalla rajapinnassa määriteltyjä metodeja. Olioiden rajapintoihin asiakasohjelmat voivat päästä käsiksi kaikille olioille yhteisen **`IUnknown-rajapinnan`** kautta, joka jokaisen luokan on vähintään toteutettava. Se sisältää metodit olion referenssilaskureiden käsittelyyn sekä olion toteuttamien muiden rajapintojen kyselyyn.

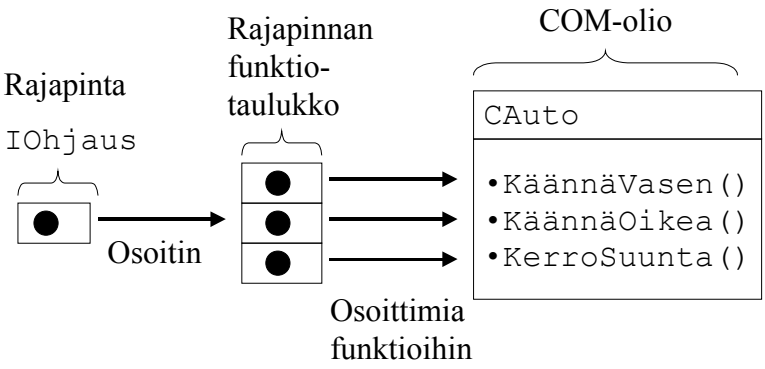
DCOMissa luokkia sekä rajapintoja havainnollistetaan usein omalla merkintätavalla, josta on esimerkki kuvassa 13.



Kuva 13. DCOM-luokka `CAuto` sekä sen rajapinnat.

Kuvassa 13 on esitetty DCOM-luokka `CAuto` sekä sen tarjoamat rajapinnat `IUnknown, IOhjaus ja IRadio`. Kuvaan on myös piirretty asiakasohjelma, jolla on referenssi `CAuto`-luokkaan `IOhjaus`-rajapinnan kautta. Useampia rajapintoja käyttämällä voidaan ryhmitellä yhteenkuuluvia metodeja kokonaisuuksiksi. Rajapinta `IOhjaus` voisi sisältää esimerkiksi metodit auton kääntämiseen sekä nykyisen suunnan selvittämiseen.

Käytännössä DCOM-luokat ovat joko EXE- tai DLL-binääritiedostoissa sijaitsevia aliohjelmia, joita kutsutaan asiakasohjelman puolelta rajapintoja käyttäen. Rajapinta ei taas ole muuta kuin osoitin taulukkoon, joka sisältää osoittimia luokan toteuttamiin metodeihin. Kuvassa 14 on esitetty `CAuto`-luokan `IOhjaus`-rajapinta tällä tavalla osoittimia käyttäen.



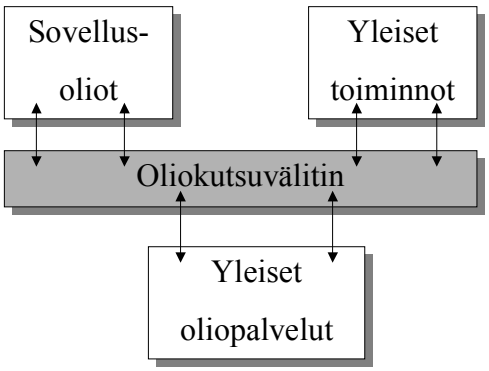
Kuva 14. DCOM-luokka `CAuto` esitettynä rajapinnan osoittimilla.

Koska DCOM-luokasta näkyy ulospäin vain sen binääritason rajapinta, on DCOM-luokkia mahdollista käyttää sekä toteuttaa ohjelmointikielillä, jotka eivät tue oliopohjaisia menetelmiä. Olioita tukevan kielen käyttö kuitenkin yksinkertaistaa ja helpottaa toteutusta. Esimerkiksi C++ -kielen luokkaperinnän mahdollistavat virtuaalitaulukot (engl. *virtual table, vtable*) sopivat sellaisenaan rajapinnan funktiotaulukoiksi. Tällöin DCOM-rajapinnat voidaan kuvata normaaleina C++:n virtuaaliluokkina ja niiden käyttö on hyvin suoraviivaista.

DCOM-luokkien toteutus on mahdollista pelkän kääntäjän avulla ilman muita apuohjelmia. Usein kehityksessä kuitenkin käytetään apuna IDL-kääntäjää (*Interface Definition Language*), joka osaa IDL-tiedoston perusteella luoda valmiin ohjelmarungon sekä sarjallistamiskomponentit (katso luku 5.2.3) luokkaa varten. IDL-tiedosto sisältää kuvaukset rajapinnoista ja metodeista, jotka luokan halutaan toteuttavan. Esimerkki IDL-tiedostosta löytyy liitteestä 2 [COM95, luku 2.1].

5.1.2 CORBA

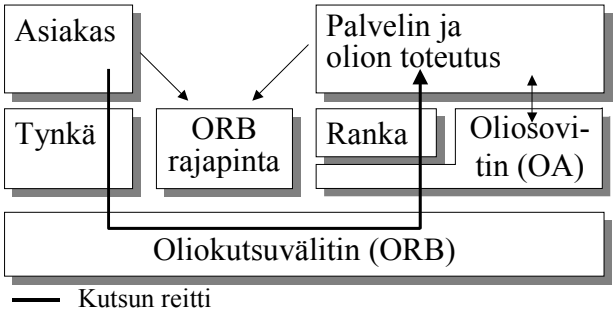
CORBA-arkkitehtuuri on laaja OMG:n standardisoima infrastruktuuri sovellusten väliseen kommunikointiin. Arkkitehtuurin eri osat muodostavat OMG-viitemallin (engl. *OMG Reference Model Architecture*), jonka eri osat on esitetty kuvassa 15.



Kuva 15. OMG-viitemalli.

Yleiset toiminnot (engl. *Common Facilities*) ovat arkkitehtuuriin sisältyvä kokoelma palveluita, jotka ovat suoraan sovellusohjelmien käytettävissä. Kehitteillä olevat palvelut liittyvät mm. liiketoimintaan, tietovirtojen ohjailuun, palomuureihin sekä dokumenttien välitykseen. Yleiset toiminnot on alati kehittyvä palvelukokoelma, joka täydentyy jatkuvasti. **Yleiset oliopalvelut** (engl. *Common Object Services*) käsittävät joukon palveluita, joita usein tarvitaan hajautettuja ympäristöjä toteutettaessa. Palveluita ovat mm. olioiden paikannuspalvelu, lisensointi- ja turvallisuuspalvelu, aikapalvelu, transaktiopalvelu sekä tapahtumapalvelu. **Sovellusolioiksi** (engl. *Application Objects)* kutsutaan olioita, jotka on toteutettu tiettyä sovellusta varten. Niiden rajapinnat eivät ole standardisoituja ja harvoin yhteensopivia muiden sovellusten olioiden kanssa [Orf98, sivut 7–20].

CORBAn ytimen muodostaa **oliokutsuvälitin** eli ORB (*Object Requester Broker*). Se toimii siltana asiakassovellusten ja palvelinsovellusten olioiden välillä. Sen tehtäviin kuuluu olioiden paikantaminen ja aktivoiminen, etäkutsujen välitys ja mahdollisten paluuarvojen palautus. Kuvassa 16 on esitetty ORB-arkkitehtuurin tärkeimmät komponentit.



Kuva 16. ORB-arkkitehtuuri [Orf98, sivu 11].

Oliosovitin (O*bject Adaptor*) auttaa oliokutsuvälitintä kutsujen välityksessä sekä olioiden aktivoinnissa. Olioiden toteutuksien rekisteröinti oliokutsuvälittimelle tapahtuu myös oliosovittimen kautta. **ORB-rajapinta** (*ORB Interface)* sisältää joukon metodeja, jotka ovat yhteisiä kaikille CORBA-toteutuksille. Ne auttavat mm. kutsujen parametrilistojen muodostamisessa sekä olioiden referenssien käsittelyssä. Pareittain toimivat sarjallistamiskomponentit **tynkä** (engl. *stub)* ja **ranka** (engl. *skeleton*) vastaavat olioiden etäkutsujen pakkaamisesta ja purkamisesta verkon yli [OMG, luvut 2.1–2.5].

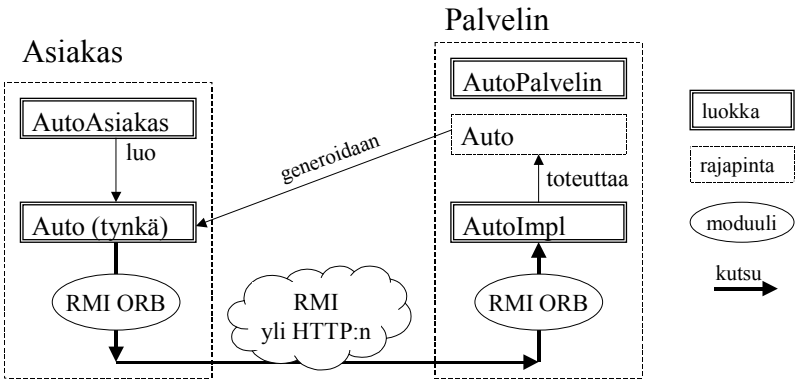
CORBAn oliot voivat toteuttaa enintään yhden rajapinnan, mutta tukevat sitä vastoin moniperintää. Olioiden rajapintojen kuvaamiseen voidaan DCOMin tapaan käyttää IDL-kieltä sekä IDL-kääntäjää helpottamaan toteutusta. Valitettavasti CORBAn ja DCOMin IDL-kielet eivät ole yhteensopivia.

5.1.3 JavaRMI

Java-kielen yleistyessä kehitettiin sille Sun Microsystemsin toimesta myös oma hajautusteknologia JavaRMI (*Java Remote Method Invocation*) mahdollistamaan Java-olioiden käyttö hajautetusti. JavaRMI on ollut osana Javan JDK-kehitysympäristöä (*Java Development Kit*) versiosta 1.1 lähtien. Sen avulla voi toteuttaa hajautettuja Java-luokkia, joita voi käyttää verkon yli samalla tavoin kuin paikallisiakin luokkia.

JavaRMI koostuu noin kahdestakymmenestä luokasta ja rajapinnasta sekä erillisestä apuohjelmasta, joka palvelimen päässä huolehtii luokkien rekisteröinnistä. Toteutuksen rakenne noudattaa hyvin pitkälle yleistä oliopohjaisen väliohjelmiston rakennetta, joka

kuvataan luvussa 5.1. Kuvassa 17 on havainnollistettu esimerkkiluokan `Auto` toteutusta JavaRMI:llä.



Kuva 17. Luokan `Auto` toteutus JavaRMI:tä käyttäen.

Palvelimen puolella on aluksi määritelty rajapintaa `java.rmi.Remote` laajentava `Auto`-rajapinta, jonka metodeja sallitaan kutsuttavan hajautetusti. Metodien toteutus on tehty `AutoImpl`-luokkaan, joka toteuttaa rajapinnan `Auto` ja periytyy luokasta `java.rmi.server.UnicastRemoteObject`. Rajapinnan ja toteutuksen pohjalta on tässä vaiheessa voitu luoda asiakasta varten tynkäluokka `Auto`. Tyngän generointi tapahtuu automaattisesti siihen tarkoitettua apuohjelmaa käyttäen. Lopuksi palvelimen puolella luodaan vielä apuluokka `AutoPalvelin`, joka käynnistyessään rekisteröi `AutoImpl`-luokan JavaRMI-järjestelmälle.

Asiakkaan puolella riittää toteuttaa `AutoAsiakas`-luokka, joka käynnistyessään luo instanssin tynkäluokka `Autosta` ja alkaa käyttämään sitä kuten tavallista Java-luokkaa. `Auto`-luokka sisältää kaikki tarvittavat rutiinit metodien välittämiseksi palvelimelle [SUN, luvut 2.4 ja 2.5].

Eräs JavaRMI:n erikoisempia ominaisuuksia on sen kyky siirtää tynkäluokkia ajonaikaisesti. Tämä **tynkäluokkien dynaaminen siirto** eli DSL (*Dynamic Stub Loading*) antaa asiakassovelluksille mahdollisuuden käyttää hajautetusti luokkia, joiden rajapinnat eivät ole tiedossa käännöksen aikana. Koska operaatio vaatii ajettavan koodin välitystä

palvelimelta asiakkaalle, sisältyy siihen myös suuri turvallisuusriski, joka täytyy käsitellä tilannekohtaisesti [SUN, luku 3.4].

5.1.4 Muita toteutuksia

Parhaiten tunnettujen DCOM- ja CORBA-väliohjelmistojen lisäksi on olemassa joukko vähemmän tunnettuja hajautusratkaisuja. Monet niistä ovat tutkijapiireissä toteutettuja kokeiluluontoisia projekteja, jotka ovat jääneet pienehkön piirin käyttöön. Osa toteutuksista on lähempänä todellista hajautettua käyttöjärjestelmää (katso luku 3.4.2) kuin erillistä väliohjelmistoa. Internetin kasvu on myös luonut joukon hajautusprotokollia, jotka hyödyntävät yleistä WWW-sivujen siirtoon tarkoitettua HTTP-protokollaa (*HyperText Transfer Protocol)*. Tässä luvussa on listattu joitakin tällaisia ohjelmistoja. Tarkempia tietoja voi löytää alan kirjallisuudesta ja alla mainituista lähteistä.

Chorus on ranskalaisessa Institut National de Recherche en Informatique et Automatique -instituutissa (INRIA) 1979 luotu hajautettu käyttöjärjestelmä, jonka kehitystä jatkoi Chorus Systems ja myöhemmin Sun Microsystems. Chorus muistuttaa monessa suhteessa Mach-käyttöjärjestelmää. Se on toteutettu mikroytimen (engl. *micro-kernel*) muodossa, se pystyy emuloimaan binääritasolla UNIX-käyttöjärjestelmää, siinä on joustava virtuaalimuistin tuki ja se tukee moniprosessorikoneita jaettua muistia käyttämällä. Lisäksi Chorus osaa jakaa dynaamisesti tehtäviä eri palvelimien kesken tukien konfiguroitavia palvelinryhmiä kuormituksen säätelemiseksi. Chorus on myös suunniteltu reaaliaikaisia järjestelmiä silmälläpitäen ja se sisältää operaatiot säikeiden suoritusprioriteettien säätelemiseksi [Cou94, sivut 566–579].

Mach on alunperin Carnegia-Mellonin yliopistossa vuosina 1985-1994 kehitetty ja Utahin yliopistossa jatkokehitetty käyttöjärjestelmä, joka mahdollistaa ohjelmien hajautetun suorituksen käyttöjärjestelmän ytimen (engl. *kernel)* tasolla. Mach sisältää kaikki tärkeimmät UNIX-käyttöjärjestelmän palvelut ja tämä emulointi mahdollistaa olemassa olevien UNIX-sovellusten suorituksen Machin päällä. Machin muita ominaisuuksia ovat mm. joustava virtuaalimuistin käyttö, resurssien läpinäkyvä hajautus ja hyvä siirrettävyys eri laitteistojen kesken. Machin kehitys itsenäisenä projektina on loppunut, mutta osia siitä

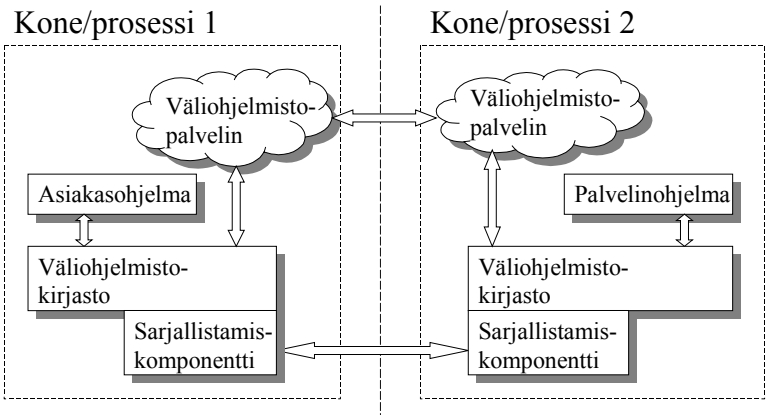
on käytetty muissa toteutuneissa projekteissa, kuten MkLinux-käyttöjärjestelmässä [Cou01, luku 18].

XML-RPC on XML-standardiin pohjautuva järjestelmä, joka mahdollistaa hajautettujen etäkutsujen teon Internetissä HTTP-protokollaa käyttämällä. Kutsun teko tapahtuu muuttamalla etäkutsun tunnus sekä sen parametrit XML-muotoiseksi tiedoksi, joka välitetään HTTP-protokollan POST-komennolla vastaanottajalle. Vastaanottaja purkaa XML-tiedosta etäkutsun tunnuksen sekä parametrit, ja suorittaa varsinaisen kutsun. Mahdolliset paluuarvot palautuvat POST-komennon vastauksen mukana XML-muodossa. XML-RPC on kevyt ja yksinkertainen tapa etäkutsujen tekoon, mutta sen yksinkertaisuus jättää aukkoja tiedon oikeellisuuteen ja turvatarkistuksiin liittyviin seikkoihin. Liitteessä 4 on esimerkki XML-RPC -viestistä [And00, sivut 508–517].

SOAP (*Simple Object Access Protocol)* on XML-RPC:n tapaan etäkutsuja välittävä protokolla, joka välittää kutsut XML-muotoisina viesteinä. SOAP ei kuitenkaan ota kantaa viestien siirtotapaan, joten viestejä voi siirtää HTTP-protokollan lisäksi esim. sähköpostin välityksellä. SOAP-viesteissä käytetyt tietotyypit noudattavat XML Skeemassa (katso luku 7.1.4) määriteltyjä tyyppejä voiden näin muodostaa hyvinkin monimutkaisia tietorakenteita. Liitteestä 4 löytyy esimerkki yksinkertaista SOAP-viestistä [And00, sivut 544–552].

5.2 Järjestelmän osat

Hajautettua väliohjelmistoa hyödyntävä järjestelmä sisältää hieman yksinkertaistettuna kuvan 18 esittämät rakenteet jossakin muodossa toteutettuna.



Kuva 18. Hajautetun järjestelmän osat.

Asiakasohjelma ja **palvelinohjelma** sisältävät järjestelmän logiikan ja toiminnallisuuden kohdealueen osalta. Ne pitävät sisällään järjestelmässä esiteltyjen olioiden implementaatiot, mutta eivät hajautukseen liittyvää ohjelmakoodia. Väliohjelmistoa asiakas- ja palvelinohjelmat käyttävät siihen tarkoitetun **väliohjelmakirjaston** kautta. Se sisältää väliohjelmiston ulospäin näkyvän sovellusrajapinnan, jonka kautta hajautus tapahtuu. Väliohjelmistokirjasto puolestaan keskustelee taustalla toimivan **väliohjelmistopalvelimen** kanssa, joka hallitsee prosessien ja koneiden välisiä yhteyksiä. Välitettäessä tietoa prosessi- tai konerajojen yli, kirjasto kutsuu **sarjallistamiskomponenteissa** sijaitsevia rutiineja, jotka osaavat muuntaa oliot ja kutsut siirrettävään muotoon. Kaikki edellä esitellyt osat kuvataan yksityiskohtaisemmin luvuissa 5.2.1–5.2.5.

5.2.1 Palvelinohjelma

Uuden järjestelmän kehitys alkaa usein palvelinohjelman suunnittelusta. Palvelinohjelman tärkein tehtävä on olioiden metodien implementointi. Palvelinohjelman vastuulla on myös kertoa väliohjelmistolle, mitä rajapintoja ohjelma toteuttaa, sekä tarjota jokin keino

rajapintoja vastaavien olioiden luomiseksi. Tämän informaation välittämistä väliohjelmistolle kutsutaan yleisesti **rajapintojen ja olioiden rekisteröinniksi**. Rekisteröintitieto tallentuu joko väliohjelmiston ajonaikaisiin tietorakenteisiin tai luvuissa 5.4 ja 6.2 esiteltäviin pidempiaikaisiin varastoihin.

Palvelinohjelman täytyy osata ajaa itsensä alas tiettyjen kriteereiden täytyttyä. Kriteerit voivat vaihdella palvelinohjelman tarkoituksesta riippuen, mutta tavallisesti alasajo käynnistyy silloin, kun kaikki asiakasohjelmien referenssit ovat kadonneet, palvelinohjelma on ollut toimettomana määrätyn ajan tai loppukäyttäjä on käskenyt palvelinohjelmaa sammumaan.

Microsoftin DCOM mahdollistaa palvelinohjelman toteutuksen kahdella eri tavalla. Palvelinohjelma voi olla asiakasprosessin ulkopuolella toimiva **ulkoinen palvelin** (engl. *out-of-process server),* jota vastaa tavallinen EXE-tyyppinen ohjelma. Windows NT ja Windows 2000 -käyttöjärjestelmissä tämän tyyppinen palvelin voidaan lisäksi toteuttaa palveluna (engl. *service)*, joka voi toimia jatkuvasti taustalla sisäänkirjoittautuneesta käyttäjästä riippumatta. Ulkoisina palvelimina toteutetaan usein sovelluksia, joita on mahdollista käyttää myös itsenäisesti ilman asiakasohjelmaa. COM-automaatiota tukevat sovellukset (kuten Microsoft Word ja Microsoft Excel) ovat esimerkkejä tällaisista.

Toisena mahdollisuutena on toteuttaa palvelinohjelma **asiakasprosessiin liitettävänä** (engl. *in-process server*) palvelimena. Tällöin palvelinohjelman toteutus sijaitsee DLL-tyyppisessä dynaamisesti ladattavassa kirjastossa, jonka väliohjelmakirjasto liittää asiakasohjelman prosessiin. Asiakasprosessiin liitettävinä palvelimina toteutetaan yleensä pienimuotoisempia komponenttikirjastoja, joita voidaan käyttää ainoastaan asiakasohjelmien kautta [COM95, luku 2.3.6].

CORBA-toteutuksissa palvelinohjelma on yleensä itsenäinen asiakasohjelman ulkopuolella toimiva sovellus. Esimerkiksi ORBacus-väliohjelmistoa käytettäessä ohjelmistopalvelin on oma sovelluksensa, joka käynnistyttyään jää odottamaan asiakasohjelmien yhteydenottoja pääsilmukassaan, kunnes se (oletuksena) manuaalisesti sammutetaan [ORB01, luku 2].

5.2.2 Asiakasohjelma

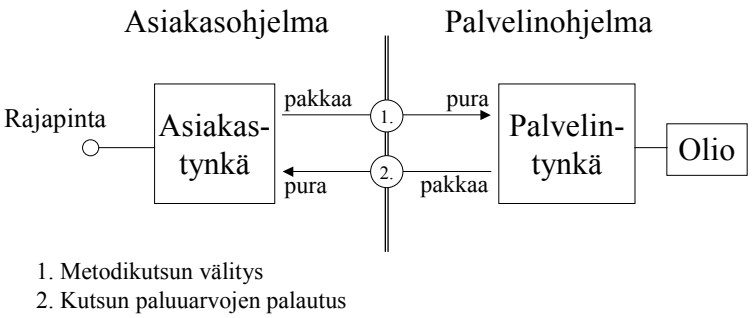
Ilman asiakasohjelmaa on palvelinohjelma tarpeeton. Asiakasohjelma käyttää toteutuksessaan palvelinohjelman tarjoamia olioita väliohjelmakirjaston kautta. Asiakasohjelman täytyy siksi tietää käytettävien olioiden tunnisteet, jotta niiden saanti väliohjelmakirjaston kautta onnistuisi. Varsinaisessa olion käytössä ei väliohjelmakirjastoa enää välttämättä tarvita, vaan oliot kommunikoivat suoraan keskenään. Asiakasohjelman vastuulla on myös huolehtia käyttämiensä olioiden vapautuksesta niiden käytön jälkeen. Olion vapautus on asiakkaan näkökulmasta harvoin todellinen olion tuhoamisoperaatio, vaan enemmänkin ilmoitus olioviittauksen purkamisesta.

Asiakasohjelmien toteutustyyppi on yleensä hyvin vapaa. Ne voivat olla omia ohjelmiaan, kirjastoja tai jopa toimia osana palvelinohjelmaa. Tärkeintä on, että asiakasohjelma pystyy kommunikoimaan väliohjelmakirjaston kanssa.

5.2.3 Sarjallistamiskomponentit

Sarjallistamiskomponenteissa sijaitsevat rutiinit, jotka purkavat ja pakkaavat olioiden metodikutsut ja niiden parametrit verkon yli lähetettäväksi. Sarjallistamiskomponentit voidaan liittää joko osaksi asiakas- tai palvelinohjelmaa taikka vaihtoehtoisesti omiksi erillisiksi moduuleiksi.

Komponentteja eli tynkiä on kahta eri tyyppiä. Asiakasohjelman käyttämä sarjallistamiskomponentti eli **asiakastynkä** (engl. *proxy, stub*) osaa pakata palvelinohjelmalle kohdistuvan metodikutsun ja purkaa sen paluuarvon. Palvelinohjelman **palvelintyngästä** (engl. *stub, skeleton*) käytetään myös nimitystä ranka. Se osaa puolestaan purkaa asiakasohjelmalta saadun metodikutsun ja pakata sen mahdolliset paluuarvot takaisin asiakasohjelmalle lähetettäväksi. Kuvassa 19 on esitetty sarjallistamiskomponenttien keskinäinen toiminta.



Kuva 19. Sarjallistamiskomponentit.

Sarjallistamiskomponentit keskustelevat keskenään käyttäen jotakin ennalta sovittua siirtoprotokollaa. Kaksi tällaista protokollaa esitellään yksityiskohtaisesti luvussa 5.3. Sarjallistamiskomponenttien toteutuksesta kerrotaan yksityiskohtaisemmin luvussa 7.2.

5.2.4 Väliohjelmistopalvelin

Hajautuksen tapahtuessa eri prosessien välillä tarvitaan ulkopuolinen prosessi hallitsemaan asiakasohjelman ja palvelinohjelman välistä yhteydenmuodostusta. Tämä tehtävä kuuluu väliohjelmistopalvelimelle. Se osaa keskustella muiden hajautukseen osallistuvien koneiden kanssa sekä avata, tarkkailla ja sulkea verkkoyhteyksiä niiden välillä. Nämä tehtävät kuvataan yksityiskohtaisemmin luvussa 6.3.

Väliohjelmistopalvelimen tehtäviin kuuluu myös paikallistaa rajapintaa vastaavan olion toteutus ja mahdollisesti käynnistää olion implementaation toteuttava palvelinohjelma. Tiedot tähän se hakee joko ajonaikaisesta tai pysyvästä varastosta, jonne palvelinohjelmat ovat rekisteröityneet. Luvuissa 5.4 ja 6.2 kuvataan tarkemmin näiden varastojen käyttöä ja sisältöä.

Väliohjelmistopalvelin on yleensä toteutettu jonkin tyyppisenä taustaprosessina, joka ei vaadi käyttäjän huomiota. Se voi käynnistyä automaattisesti käyttöjärjestelmän käynnistyksen yhteydessä, kuten Windows-käyttöjärjestelmän COM-väliohjelmistopalvelin SCM (*Service Control Manager*). Se voi myös olla manuaalisesti käynnistettävä, kuten esimerkiksi JavaRMI:n `rmiregistry`-ohjelma.

5.2.5 Väliohjelmistokirjasto

Hajautetun järjestelmän eri osat sitoo yhteen väliohjelmistokirjasto. Se on normaalisti ohjelmaan mukaan linkitettävä kirjasto, joka osaa hallita siirtokomponentteja sekä keskustella väliohjelmistopalvelimen kanssa. Se sisältää lisäksi rutiinit rajapintojen ja toteutusten rekisteröintiin. Väliohjelmakirjasto ei aina ole tarpeen itse metodikutsujen teossa. Kun yhteys asiakasohjelman rajapinnan ja palvelinohjelman toteutuksen välillä on luotu, ei väliohjelmistoa välttämättä tarvita kommunikointiin, vaan keskustelu voi tapahtuu puhtaasti sarjallistamiskomponenttien välillä.

5.3 Hajautusprotokollat

Väliohjelmiston suorittamat hajautetut metodikutsut välitetään eri prosessien välillä käyttäen hajautusprotokollaa. Hajautusprotokollasta käytetään joskus myös nimitystä kommunikointiprotokolla. Se määrittelee, millä tavalla hajautettujen olioiden kutsut tapahtuvat verkon yli sekä millä tavalla olioiden ja rajapintojen referenssejä esitetään, välitetään ja hallitaan hajautuksessa. Luvussa esitellään kaksi yleisessä käytössä olevaa hajautusprotokolla. Luvun 5.3.1 pääasiallinen lähde on [COM95, luku 15] ja luvun 5.3.2 [OMG, luku 15].

5.3.1 ORPC (*Object Remote Procedure Call*)

Olio-RPC on laajennus luvussa 4.2.1 esiteltyyn RPC-protokollaan. ORPC suunniteltiin erityisesti Microsoftin DCOM-väliohjelmistoa varten. Se muokkaa RPC-protokollan määrityksiä siten, että metodien kutsuminen ja olioiden välitys on mahdollista. Protokolla määrittelee myös joukon uusia tietorakenteita, joita käytetään siirrettäessä hajautukseen liittyvää informaatiota verkon yli.

Tärkein ORPC:n laajennus on **oliokutsujen mahdollistaminen** määrittelemällä uudelleen RPC:ssä käytetyn `Request`-tietueen kaksi tunnistekenttää. Käytännössä tämä tarkoittaa uuden merkityksen antamista jo olemassa oleville kentille. Alla on osa jokaiseen `Request`-tietueeseen kuuluvasta otsikko-osasta:

```
typedef struct
{
     ...
     uuid_t    object;  /* Object identifier */
     uuid_t    if_id;   /* Interface identifier */
     ...
} dc_rpc_cl_pkt_hdr_t;
```

Kenttä `object` sisältää ORPC:ssä oliotunnuksen sijaan yksilöllisen rajapintaosoittimen tunnisteen eli **IPID-arvon** (*Interface Pointer IDentifier)*. Se yksilöi tietyllä palvelimella olevan tietyn olion tietyn rajapinnan. Vaikka IPID on myös UUID-tyyppinen arvo, kuten RPC:n oliotunnuskin, se ei kuitenkaan ole globaalisti yksilöllinen UUID-standardin (katso luku 6.1.1) mukaisesti vaan palvelinkohtainen. Sen sijaan kenttä `if_id`, joka ORPC:n määrittelyssä sisältää IPID-arvon osoittaman rajapinnan todellisen IID-tunnuksen, on globaalisti uniikki. Koska IPID jo yksilöi täysin rajapinnan, ei IID-tunnus ole välttämätön, mutta antaa mahdollisuuden lisävarmistuksiin rajapinnan oikeellisuudesta.

Rajapinnan IPID-arvon lisäksi tarvitaan yksi tunniste lisää, jotta ORPC-kutsu voidaan palvelinpäässä assosioida oikean rajapinnan kanssa. Tähän tarkoitukseen ORPC määrittelee uuden UUID-tyyppisen **OXID-tunnuksen** (*Object eXporter IDentifier*). Sen avulla oliokutsu voidaan muuntaa RPC-protokollan ymmärtämään muotoon, jossa kutsujen rajapinnat on yksilöity merkkijonotunnisteilla. Jokaista OXID-tunnusta kohden on sekä asiakaskoneella että palvelinkoneella erityinen OXID-olio, jonka kautta hoidetaan referenssilaskureita sekä uusien rajapintaosoittimen saantia. OXID-tunnus välitetään jokaisen oliokutsun yhteydessä ja se sisältyy implisiittisesti kutsun mukana kulkevaan STDOBJREF-tietueeseen.

ORPC laajentaa myös käyttämäänsä NDR-siirtosyntaksia (katso luku 7.1.2) yhdellä uudella **OBJREF-primitiivityypillä**. Se mahdollistaa rajapintareferenssien siirron ja sarjallistamisen yhtenäisellä tavalla. OBJREF-tietueen sisältö voi vaihdella siirrettävän rajapinnan tyypin mukaan. Erilaisia variaatioita on yhteensä seitsemän, joista neljä olennaisinta on lueteltu taulukossa 2 [COM98, sivut 16–19].

Rajapinnan tyyppi	**Kuvaus**
NULL	Käytetään merkitsemään puuttuvaa (NULL) rajapintaa.
STANDARD	Standardi etäreferenssi. Tällöin OBJREF sisältää mm. IPID- ja OXID-tunnukset sekä rajapinnan omistavan olion yksilöivän OID-tunnuksen. Vastaa STDOBJREF-tietuetta.
CUSTOM	Käytetään silloin, jos olio haluaa määrätä oman referenssinsä esitysmuodon. OBJREF sisältää tällöin luokan CLSID-tunnuksen sekä luokkakohtaista dataa.
HANDLER	Erikoistapaus CUSTOM-variaatiosta. Tätä käytetään silloin, jos olio haluaa kontrolloida oman referenssinsä standardoitua esitysmuotoa. OBJREF sisältää tällöin luokan CLSID-tunnuksen sekä luokkakohtaista standardoitua dataa.

Taulukko 2. OBJREF-tietueen variaatiot.

Rajapintojen referenssien hallitseminen tapahtuu ORPC:ssä jokaiseen `OXID`-olioon liitetyn **`IRemUnknown`-rajapinnan** avulla. Rajapinta sisältää metodit referenssien kasvattamiseen ja vähentämiseen sekä uusien rajapintojen palauttamiseen. Rajapinnan käyttöä osana yhteyden muodostusta kuvataan tarkemmin luvussa 7.2.2.

ORPC-protokolla sisältää myös RPC-protokollaan verrattuna parannetun tavan havaita tilanne, jossa asiakasprosessi tai palvelinprosessi jostain syystä kuolee ilman, että se vapauttaa varaamiaan olioreferenssejä. Sen sijaan, että tarkkailtaisiin yksittäisten olioiden olemassaoloa, ryhmitellään olioita joukoiksi ja tehdään olemassaolotarkistus koko joukolle yhdellä kutsulla. Asiaa on kuvattu tarkemmin luvussa 6.3.2.

5.3.2 GIOP (*General Inter-ORB Protocol*)

CORBA-arkkitehtuuri tarjoaa GIOP-protokollan ratkaisuksi eri CORBA-toteutusten keskinäiseen kommunikointiin. GIOP määrittelee siirrettävien viestien rakenteet, joiden avulla olioiden metodeja voidaan kutsua, uusia olioita voidaan paikallistaa sekä hallita sarjallistamiseen käytettyjä kommunikointikanavia. GIOP on suunniteltu riippumattomaksi käytetyn siirtokerroksen protokollasta, kunhan protokolla täyttää tämän luvun loppupuolella esiteltävät olettamukset.

GIOP käyttää CORBAn IDL-kielen tietotyyppien kuvaamiseen **CDR-siirtosyntaksia** (*Common Data Representation*). Sen ominaisuuksiin kuuluu mm. kyky ymmärtää vaihtelevia tavujärjestyksiä eri koneiden välillä sekä tasata tietotyypit muistissa kiinnitettyihin tavuväleihin tehokkuuden parantamiseksi. Syntaksi on selitetty yksityiskohtaisemmin luvussa 7.1.3.

Varsinaiseen viestintään GIOP 1.0 määrittelee **seitsemän erityyppistä viestiä**. Jokainen viesti alkaa aina samanlaisella otsikkotietueella, joka sisältää seuraavat tiedot:

- GIOP-viestin kiinnitetty tunniste ”GIOP”,
- GIOP-versionumero (joko 1.0, 1.1 tai 1.2),
- kentissä käytettävä tavujärjestys,
- viestin tyyppi ja
- viestin koko.

Otsikkotietoa seuraa itse viesti, jonka sisältö määräytyy viestin tyypin perusteella. GIOP 1.0 -protokollan tukemat eri viestityypit on lueteltu taulukossa 3, jossa suunta-sarakkeessa käytetty lyhenne ’A’ merkitsee asiakasprosessia ja ’P’ palvelinprosessia.

Tyyppi	**Suunta**	**Käyttötarkoitus**
`Request`	A → P	Käytetään olioiden metodien kutsumiseen sekä rajapintojen ja toteutuksien kysymiseen.
`Reply`	A ← P	Palautetaan olion metodikutsun tulos, kysytty rajapinta tai kysytty toteutus asiakkaalle. Viesti voi myös palauttaa poikkeuksia.
`CancelRequest`	A → P	Peruutetaan aiemmin tehty `Request`- tai `LocateRequest`-kutsu.
`LocateRequest`	A → P	Käytetään kysyttäessä, pystyykö palvelin suorittamaan annetun olioreferenssin metodeja, ja jos ei, niin mikä palvelin pystyy. Sama informaatio on myös mahdollista saada `Request`-kutsulla.
`LocateReply`	A ← P	Palautetaan aiemmin tehdyn `LocateRequest`-kutsun tulos asiakkaalle.
`CloseConnection`	A ← P	Kertoo asiakkaalle palvelimen sulkeutuvan, jolloin asiakkaan ei pidä enää odottaa palvelimelle suunnattujen kutsujen onnistuvan.
`MessageError`	A ↔ P	Käytetään ilmoittamaan mahdollisesta virheestä minkä tahansa kuuden muun viestityypin käsittelyn jälkeen.

Taulukko 3. GIOP 1.0 -protokollan viestityypit.

GIOP tekee tiettyjä olettamuksia käytetystä siirtoprotokollasta. Siirtoprotokollan täytyy olla **yhteydellinen** *(*engl. c*onnection-oriented)*, **luotettava** *(*engl. *reliable)* (katso luku 4.1.1) ja sen täytyy pystyä havaitsemaan siirrossa tapahtuneet virheet. Tiedonsiirtoa täytyy myös pystyä käsittelemään yhtenäisenä tavuvirtana, eikä protokolla saa asettaa esteitä kerralla siirrettävän datan määrälle. Esimerkiksi TCP/IP-protokolla (katso luku 4.1.2)

täyttää tällaiset vaatimukset ja sille onkin olemassa oma GIOP-toteutus IIOP *(Internet Inter-ORB Protocol)*, joka mahdollistaa eri oliovälitintoteutusten yhteistoiminnan Internet-tasolla.

5.4 Tyyppitiedon varastointi

Tyyppitiedolla tarkoitetaan informaatiota jonkin olion rajapinnan ominaisuuksista, kuten rajapinnan nimestä, metodeista ja attribuuteista. Tyyppitietoa tarvitaan, kun asiakas käyttää palvelimella sijaitsevia olioita niiden rajapintojen kautta. Usein tyyppitieto on tiedossa jo asiakasohjelman kehittämisen aikana, ja tällöin tyyppitiedon perusteella voidaan toteuttaa hajautettujen kutsujen teossa tarvittavat siirtokomponentit. Siirtokomponenttien toteutus on lähes poikkeuksetta automatisoitua, sillä niiden sisältämät sarjallistamisrutiinit voivat olla hyvinkin monimutkaisia.

Asiakasohjelman käyttäessä jonkin toisen tahon toteuttamia komponentteja, ei sillä ole välttämättä mahdollisuutta päästä käsiksi palvelinohjelman toteuttamiin olioihin ja rajapintoihin lähdekoodin tasolla. Tällöin palvelinohjelman täytyy tallentaa tyyppitieto toteuttamiensa olioiden rajapinnoista johonkin ulkoiseen varastoon, josta asiakasohjelmat voivat sen tarvittaessa hakea. Luvuissa 5.4.1 ja 5.4.2 kuvataan, kuinka DCOM ja CORBA toteuttavat tämän varastoinnin. Vaikka toteutustavat ovat erilaiset, on käytännössä kyse samasta tehtävästä. Lukujen pääasialliset lähteet ovat [Gri98, sivut 158–164] ja [OMG, luku 10].

5.4.1 Tyyppikirjastot

Tyyppikirjastot (engl. *type libraries)* ovat lähinnä COM-teknologian käyttämiä binäärimuotoisia tietovarastoja, jotka sisältävät informaatiota tietyn palvelinohjelman toteuttamista olioista ja niiden rajapinnoista. Lisäksi tyyppikirjastot voivat sisältää enumeraatioita, vakioita tai viittauksia toisiin tyyppikirjastoihin.

Tyyppikirjastot ovat välttämättömiä kolmannen osapuolen tekemien COM-luokkien käyttämiseksi. Microsoft käyttää tyyppikirjastoja laajasti COM-automaatiota tukevien kaupallisten sovelluksiensa (mm. Microsoft Office -tuotteet) rajapintojen julkistamiseen.

Tyyppikirjastojen avulla näitä sovelluksia pystyy käyttämään eri ohjelmointikielistä käsin ja luomaan esimerkiksi dokumentteja automatisoidusti.

COM-arkkitehtuurissa tyyppikirjastot voivat olla tallennettu kokonaan erilleen palvelinohjelmasta omiin binäärimuotoisiin TLB-tiedostoihin (*Type Library Files)* tai ne voivat olla sisällytetty itse palvelinohjelmaan Windows-käyttöjärjestelmän tukemana resurssitietona (engl. *resource data).* Tyyppikirjasto sisältää saman informaation kuin palvelinohjelman rajapintoja kuvaava IDL-tiedosto ja kirjasto onkin mahdollista tuottaa automaattisesti IDL-tiedoston käännöksen yhteydessä.

Tyyppikirjastojen käyttämiseen ja luomiseen COM tarjoaa joukon funktioita ja rajapintoja. Sovellus pystyy lataamaan haluamansa tyyppikirjaston `LoadTypeLib`-funktiota käyttäen. Funktio palauttaa osoittimen `ITypeLib`-rajapintaan, jonka kautta kirjaston sisältämää tyyppitietoa pystyy tarkastelemaan. Jotkin sovelluskehittimet hyödyntävät tätä rajapintaa ja sisältävät ns. ”olioselaimen” (engl. *Object Browser*), jonka avulla kehittäjä voi tarkastella systeemiin asennettujen tyyppikirjastojen sisältöä. Tyyppikirjastojen luonti sovelluksesta käsin onnistuu `CreateTypeInfo`-rajapinnan metodien avulla [MSDN].

5.4.2 Rajapintavarastot

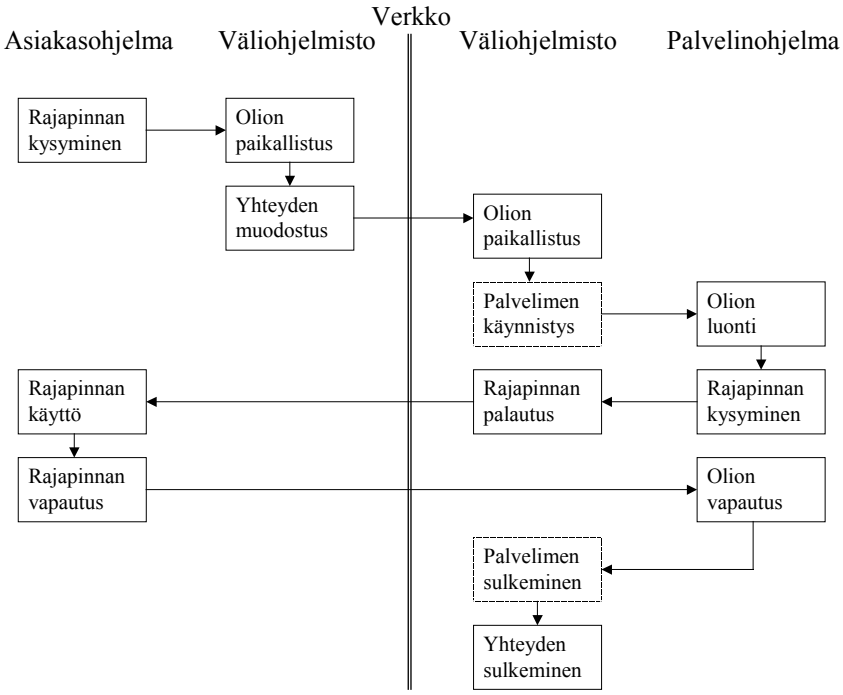
CORBA-arkkitehtuurin määrittelemät rajapintavarastot (*Interface Repository)* sisältävät tietoa palvelimen toteuttamista moduuleista, vakioista, käyttäjän tyypeistä ja rakenteista sekä tärkeimpänä olioiden metodeista ja niiden parametreista. Informaatio on sama kuin mitä OMG IDL -kielellä pystyy esittämään.

CORBA tarvitsee rajapintavarastojen sisältämää tyyppitietoa aina, jos tyyppitieto ei ollut saatavissa sarjallistamiskomponenttien luonnin aikana. Tällöin kutsujen tekoon käytetään **dynaamista kutsurajapintaa** (*Dynamic Invocation Interface*) ja tyyppitieto olioiden metodeista ja parametreista selvitetään vasta ajonaikana. Rajapintavarastoja voidaan hyödyntää myös CASE-sovelluksissa (C*omputer-Aided Software Engineering)* helpottamaan ohjelmistonkehitystä sekä yhdyskäytävissä (engl. *gateway)* mahdollistamaan eri hajautusjärjestelmien yhteistoiminnan.

Koska CORBA ei määrittele varastointimediaa, voidaan varastona käyttää esim. tiedostoa tai tietokantaa CORBA-implementaatiosta riippuen. Kaikkien varastototeutusten täytyy kuitenkin tarjota sama joukko standardin määräämiä rajapintoja varastoon tallennetun hierarkkisen tiedon käsittelemiseksi. **ORBacus** toteuttaa rajapintavaraston erillisenä palvelinsovelluksena (`irservice`), joka voidaan asentaa Windows-käyttöjärjestelmässä taustapalveluksi. Varastoon voidaan lisätä IDL-kuvauksia `irfeed`-apuohjelman avulla ja poistaa `irdel`-ohjelmalla [ORB01, luku 14].

6 Hajautuksen hallinta

Asiakkaan ja palvelimen välinen kommunikointi hajautuksen eri vaiheissa on järkevää jakaa osiin ja tarkastella vaiheita erikseen. Kuvassa 20 on kuvattu rajapinnan käytöstä seuranneet toiminnot, kun rajapinnan toteuttava olio sijaitsee eri palvelimella kuin asiakasprosessi.



Kuva 20. Hajautuksen vaiheet etärajapinnan käytölle.

6.1 Olion tunnistus

Ennen kuin olion metodeja voidaan kutsua, täytyy oliosta luoda uusi ilmentymä. Tällöin tarvitaan jokin tapa nimetä ja tunnistaa luotava olio. Kyseisen tunnisteen muoto voi vaihdella toteutuksesta riippuen, mutta tunnisteen tulisi kuitenkin olla mahdollisimman yksikäsitteinen, jotta olioiden luomisesta vastaava järjestelmä osaa palauttaa halutun oikean olion. Tässä luvussa esitellään kaksi eri tunnisteformaattia UUID ja IOR. Näistä ensimmäistä käytetään lähinnä olioiden tunnistamiseen ja jälkimmäistä lisäksi olioiden paikallistamiseen. Lukujen lähteet ovat [DCE] ja [OMG, luku 13.6].

6.1.1 UUID (*Universally Unique Identifier*)

Eräs yleisesti käytetty tunnistemuoto on UUID-tunniste. Tämän tunnistemuodon suurin etu on se, ettei uusien tunnisteiden luomiseen tarvita minkäänlaista keskitettyä hallinnointia, vaan tunnus voidaan luoda täysin paikallisesti. Jokaisen tunnuksen taataan olevan erittäin suurella todennäköisyydellä yksilöllinen niin ajan kuin paikankin suhteen aina vuoteen 3400 saakka. Microsoft käyttää UUID-tunnisteesta omaa muotoaan, joka tunnetaan nimellä **GUID** (*Globally Unique IDentifier)* [DCE].

UUID-tunnus muodostuu 16 oktetista, jolloin tunnuksen kokonaispituudeksi tulee 128 bittiä. Tunnus rakentuu yleisellä tasolla seuraavista osista:

- **Aikaleima** (oktetit 1–8) on koneen kellonajasta muodostettu lukusarja.
- **Versio** (oktetin 8 loppu) on DCE-versionumero.
- **Muoto** (oktetin 9 alku) sisältää tietoa tunnuksen muodosta.
- **Sekvenssi** (oktetit 9–10) on aikaleiman sisällä juokseva numerosarja.
- **Paikka** (oktetit 11–16) on konekohtainen numerosarja.

Aikaleima muodostetaan koneen UTC-kellonajasta (*Universal Time Coordinated)* ja se mittaa 100 nanosekunnin pituisten yksikköjen määrää ajanhetkestä 15.10.1582 00:00.00 alkaen. Myös paikallista aikaa voidaan käyttää tietyin varauksin, jos UTC-aikaa ei ole saatavilla. **Versio**- ja **muotokentät** sisältävät erinäistä toteutuskohtaista informaatiota. Muun muassa Microsoftin käyttämät GUID-tunnukset voidaan erottaa yleisistä UUID-tunnuksista näiden kenttien sisällön perusteella.

Sekvenssikentän avulla voidaan erottaa samalla ajanhetkellä samassa paikassa luodut tunnukset toisistaan. Tunnuksia luova ohjelma pitää koko ajan tallessa juoksevaa sekvenssinumeroa ja huomatessaan aikaleiman olevan sama kuin edellisellä luodulla tunnuksella, kasvattaa sekvenssiä. Sekvenssi nollataan aina aikaleima-kentän arvon vaihtuessa.

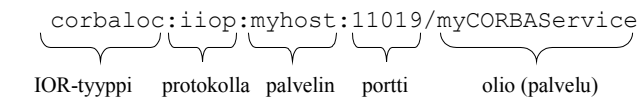
Paikkakenttä erottaa samalla ajanhetkellä, mutta fyysisesti eri paikoissa luodut tunnukset toisistaan. Jos koneessa on IEEE 802 -tunnuksen sisältävä verkkokortti, saadaan arvo yksinkertaisimmin siltä. Muussa tapauksessa tunnus muodostetaan systeemin tilaa kuvaavista muuttujista, joiden arvot vaihtelevat ajallisesti. Näitä muuttujia voivat olla esimerkiksi:

- koneen muistin koko ja vapaan muistin määrä,
- käynnistysaseman koko ja vapaan tilan määrä,
- hiirikursorin paikka,
- parasta aikaa suoritettavien prosessien ja säikeiden määrä,
- prosessien ja säikeiden tunnukset sekä
- ohjelman pino-osoittimen ja rekisterien arvo.

Sopivasti kenttiä yhdistelmällä saadaan arvo, joka on erittäin pienellä todennäköisyydellä (tunnuksia pitäisi luoda tuhansia samalla millisekunnilla, jotta arvo toistuisi) sama kuin jossakin muussa ympäristössä samalla hetkellä luotu tunnus.

6.1.2 IOR (*Interoperable Object Reference*)

GIOP määrittelee olioiden tunnistamiseen uudenlaisen olioreferenssin, joka sisältää olion tunnisteen lisäksi tietoa palvelimesta, jolla olion toteuttava palvelinohjelma sijaitsee. IOR-referenssi koostuu olion tunnisteesta sekä protokollasta riippuvista osista. IOR voi olla esimerkiksi kuvassa 21 esitetyn näköinen merkkijono.



Kuva 21. Esimerkki IOR-tunnisteen rakenteesta.

Sisäisesti GIOP käsittelee IOR-referenssejä binäärimuodossa. Riippumatta CORBA-toteutuksesta, täytyy GIOP:n osata muuntaa merkkijonomuotoinen IOR binäärimuotoiseksi IOR-referenssiksi. Tämä mahdollistaa eri CORBA-implementaatioiden toimivuuden keskenään.

6.2 Olion paikallistaminen ja luonti

Väliohjelmiston saadessa asiakasprosessilta pyynnön olion luomiseksi, täytyy sen paikallistaa, missä olion toteutus sijaitsee. Tämä informaatio olion toteutuksesta pidetään yleensä erillään pääohjelmasta omassa varastossaan, joka voi olla joko pysyvä tai ajonaikainen. DCOMin käyttämä varasto on käyttöjärjestelmän systeemirekisteri, kun taas CORBAssa sijaintitietoa voidaan tallentaa toteutusvarastoon. Seuraavissa luvuissa 6.2.1 6.2.2 kuvaillaan tarkemmin näitä varastoja. Lukujen teksti perustuu lähteisiin [COM95, luku 6.1] ja [ORB01, luku 7].

6.2.1 Systeemirekisteri

Windows-käyttöjärjestelmän alla toimiva DCOM varastoi tiedon olioiden sijainnista Windowsin systeemirekisteriin (engl. *system registry*). Systeemirekisteri on käyttöjärjestelmän hallinnoima yhtenäinen tiedon taltiointipaikka, joka rakentuu hierarkkisesti avaimista (engl. *registry key*) ja arvoista (engl. *registry value*) siten, että jokainen avain voi sisältää joko uusia aliavaimia tai varsinaisia arvoja. Avainhaara voi myös pitää sisällään jonkin oletusarvon. Yleisellä tasolla systeemirekisteri siis muistuttaa tutumpaa tiedosto- ja hakemistojärjestelmää.

Riippuen palvelinohjelman tyypistä (katso luku 5.2.1), tapahtuu olioita vastaavien luokkien rekisteröinti systeemirekisteriin eri tavalla. Asiakasprosessiin liitettävät palvelinohjelmat sekä erilliset sarjallistamiskomponentit (engl. *object handler*) sisältävät kaksi dynaamisesti kutsuttavaa funktiota `DllRegisterServer` sekä `DllUnregisterServer`, joista ensimmäisen avulla tapahtuu palvelinohjelman luokkien rekisteröinti ja jälkimmäisellä rekisteröinnin poisto. Näiden funktioiden kutsumiseen sisältää Windows `regsvr32` apuohjelman, jonka avulla rekisteröinnin voi suorittaa haluamalleen palvelinohjelmalle. Prosessin ulkopuoliset palvelimet ovat suoritettavia ohjelmia ja niiden rekisteröintioperaatiot tapahtuvat vakiintuneilla komentoriviparametreilla `"-regserver"` ja `"-unregserver"`.

Tieto luokkien toteutuksien sijainnista tallennetaan rekisteriin `CLSID`-avaimen alle. Jokaista rekisteröitävää luokkaa kohden luodaan uusi aliavain, jonka nimi muodostetaan luokan CLSID-tunnuksesta. CLSID on GUID-muotoinen globaali tunnus (katso luku 6.1), joka yksilöi luokan. `CLSID`-avaimen alle tuleva sijaintitieto riippuu palvelimen tyypistä taulukon 4 esittämällä tavalla.

Palvelintyyppi	**Rekisteriavain**	**Sijaintitieto**
Asiakasprosessiin liitettävä palvelin	`InprocServer32`	Tiedostopolku DLL-palvelinkirjastoon
Prosessin ulkopuolinen palvelin	`LocalServer32`	Tiedostopolku EXE-palvelinohjelmaan
Sarjallistamiskomponentti	`InprocHandler32`	Tiedostopolku DLL-sarjallistamiskomponenttiin

Taulukko 4. Rekisteröinnissä syntyvät avaimet.

Palvelinohjelman sijaitessa muulla kuin paikallisella koneella, kirjoitetaan tästä merkintä rekisteriin `AppId`-avaimen alle. `AppId`-tunnus yksilöidään tällöin GUID-tunnuksella, johon viitataan `CLSID`-avaimen alta. `AppId`-avain sisältää kootusti palvelinohjelman hajautukseen liittyvää tietoa, kuten palvelinohjelman omistavan koneen osoitteen sekä erilaisia käyttöoikeus- ja turvallisuustunnuksia yhteydenmuodostukseen liittyen.

Esimerkiksi asiakasprosessiin liitettävä ja koneella 10.5.234.21 sijaitseva palvelinohjelma ”Testi Server”, joka toteuttaa luokan {12345678-ABCD-1234-5678-9ABCDEF00000}, voisi olla rekisteröity systeemirekisteriin seuraavasti (arvot kursivoitu erotuksena avaimista):

```
CLSID
   {12345678-ABCD-1234-5678-9ABCDEF00000} = Testi Server
      InprocServer32 = "C:\COM\TESTSRVR.DLL"
      AppId = {456789123-CDFA-4321-1234-2ABCDEF12345}
```

```
AppId
   {456789123-CDFA-4321-1234-2ABCDEF12345}
      RemoteServerName = ”10.5.234.21”
```

6.2.2 Toteutusvarasto

Hieman DCOMin systeemirekisteriä vastaava komponentti CORBA-arkkitehtuurissa on toteutusvarasto IMR (*Implementation Repository*). Sen avulla voidaan tietty olioreferenssi liittää olion toteuttavaan palvelinohjelmaan ajon aikana. Tämä mahdollistaa esimerkiksi palvelinohjelman sijainnin vaihtamisen ilman, että asiakasohjelma huomaa muutosta. Toteutusvarastoon voidaan palvelinohjelman sijaintitiedon lisäksi tallentaa muutakin toteutukseen liittyvää informaatiota ja metatietoa, kuten käyttöoikeus-, virheenetsintä- ja versiotietoja, jos CORBA-implementaatio tähän vain kykenee.

Toteutusvarastoa käytettäessä asiakasohjelmien IOR-referenssit (katso luku 6.1.2) viittaavat IMR-palveluun palvelinohjelman sijaan. Kun asiakas tekee pyynnön tällaista referenssiä käyttäen, pyyntö ohjautuu IMR:lle, joka käynnistää palvelinsovelluksen OAD-väliohjelmapalvelinta (katso luku 6.3) käyttäen ja palauttaa asiakkaalle uuden etäreferenssin. Palautettu referenssi sisältää tiedon palvelinohjelman todellisesta sijainnista (koneen sekä yhteysportin) ja sen avulla asiakas pääsee käsiksi varsinaiseen palvelimen toteuttamaan olioon.

GIOP määrittelee tarkasti asiakasohjelman ja toteutusvaraston välisen toiminnan, jotta CORBA-yhteensopivat asiakasohjelmat pystyvät käyttämään eri CORBA-toteutusten toteutusvarastoja. Sen sijaan CORBA-arkkitehtuuri ei standardoi, millä tavalla palvelinohjelmien ja toteutusvaraston keskinäinen kommunikointi toteutetaan, vaan tämä jätetään implementaation ratkaistavaksi.

ORBacuksen IMR-varasto on toteutettu omana taustalla toimivana sovelluksenaan, joka voidaan asentaa myös Windows-palveluksi. Edeltävissä kappaleissa mainittujen toimintojen lisäksi se pitää yllä ajonaikaista tietoa OAD-väliohjelmistopalvelimien ja varaston hallinnoimien palvelinsovellusten tilasta. Palvelinsovellusten rekisteröinti IRM:lle sekä rekisteröinnin poisto tapahtuu erityisellä `imradmin`-apuohjelmalla. Tämän lisäksi

ORBacus sisältää graafisen Javalla toteutetun hallintaohjelman helpottamaan IRM:n käyttöä.

Toteutusvaraston rinnalle CORBA tarjoaa vaihtoehtoisen tavan olion sijainnin määrittämiseksi. CORBAn nimipalvelun (*Naming Service*) avulla palvelin- ja asiakassovellukset voivat vaihtaa tietoa olioiden sijainnista. Jotta järjestelmä toimisi, täytyy palvelinohjelman rekisteröidä toteuttamansa oliot ja niiden rajapinnat nimipalvelulle. Tämän jälkeen asiakassovellus voi rajapinnan nimen avulla kysyä nimipalvelulta rajapinnan toteuttavan olion IOR-referenssin.

6.3 Yhteyden hallinta

Asiakasohjelman ja palvelinohjelman välinen yhteyden hallinta käsittää yhteyden avaamiseen, sen ylläpitämiseen sekä lopettamiseen liittyvät toimenpiteet. Yhteyden hallinnasta vastaa väliohjelmistopalvelin. Esimerkkejä väliohjelmistopalvelimista ovat DCOMin SCM (*Service Control Manager*), CORBAn OAD (*Object Activation Daemon*) ja Javan RMID (*RMI Daemon*). Ne ovat toteutukseltaan taustalla pyöriviä prosesseja, jotka keskustelevat sekä väliohjelmiston että muilla palvelimilla toimivien väliohjelmistopalvelimien kanssa ja osaavat tarvittaessa luoda yhteyden asiakasohjelman ja palvelinohjelman välille. Luvussa käytetyt lähteet ovat [COM95], [COM98], [OMG] sekä [SUN].

6.3.1 Yhteyden avaaminen

Uuden yhteyden muodostaminen alkaa asiakasohjelman tarpeesta saada referenssi olioon, jonka toteutus ei sijaitse samalla koneella kuin itse asiakasohjelma. Väliohjelmiston toteutuksesta riippuen joko väliohjelmistokirjasto tai väliohjelmistopalvelin tunnistaa tämän tilanteen ja yrittää paikallistaa palvelimen, jolla palvelinohjelma sijaitsee. Paikallistaminen voi tapahtua joko luvussa 6.2 esiteltyjen sijaintitietovarastojen avulla tai vaihtoehtoisesti asiakasohjelma voi olion luonnin yhteydessä eksplisiittisesti määrätä palvelimen.

Seuraavassa vaiheessa asiakaskoneen väliohjelmisto ottaa yhteyden palvelinkoneen väliohjelmistopalvelimeen. Yhteydenmuodostustapa riippuu käytetystä siirtoprotokollasta. Esimerkiksi DCOMin tapauksessa **ORPC** (katso luku 5.3.1) aloittaa yhteyden kutsumalla palvelimen SCM:n `IRemoteActivation`-rajapinnan[3] (normaali RPC-rajapinta, ei siis COM-rajapinta) metodia `RemoteActivation`. Metodin avulla asiakaskoneen väliohjelmisto saa haettua haluamansa olion OXID- ja IPID-tunnukset, joiden avulla olion rajapinnan metodien kutsuminen on mahdollista. Usein ensimmäinen pyydetty olio on palvelimen **luokkatehdas** (engl. c*lass factory*). Se on erikoisluokka, jonka `IClassFactory`-rajapinnan kautta onnistuu kaikkien muiden palvelimen olioiden luonti [COM95, luvut 15.1.1–15.1.3 ja 15.5].

CORBAn tapauksessa **GIOP-spesifikaatio** ei määrittele, millä tavalla asiakasohjelmat palvelinyhteyksiä hallitsevat, vaan tämä jää toteutuksen vastuulle. Yksi mahdollisuus olisi avata ja sulkea uusi yhteys jokaista etäkutsua varten, mutta käytännössä tämä olisi hyvin tehotonta. GIOP:n TCP/IP-toteutus **IIOP** avaa ensimmäisellä kerralla normaalin TCP/IP-yhteyden asiakkaan ja palvelimen välille ja pitää yhteyttä auki niin kauan kuin palvelimen olioihin on asiakkaalla etäreferenssejä [OMG, luku 15.5].

6.3.2 Yhteyden olemassaolon tarkistus

Laadukkaimmissakin verkoissa yhteys saattaa väliaikaisesti katketa. Väliohjelmiston täytyy havaita tämä ja kyetä käsittelemään tilanne niin, että ohjelmiston toiminta voi jatkua verkkoyhteyden palattua. Virheen havaitsemiksi käytetään yleisesti tapaa, jossa asiakas lähettää palvelimelle säännöllisin väliajoin lyhyen viestin ilmoittaen olevansa yhä toiminnassa (tapaa kutsutaan englannissa termillä *pinging*). Jos palvelin ei tietyn ajan kuluessa saa ilmoitusta asiakkaalta, olettaa se asiakaskoneen olevan poissa toiminnasta ja vapauttaa kaikki asiakaskoneen luomat oliot muististaan.

---

[3] Lähteessä [COM95] käytetään kyseisestä rajapinnasta nimeä ISCMtoSCM.

Asiakaspuolella yhteyden katkeaminen ilmenee viimeistään olion metodikutsujen yhteydessä. Kutsu palaa tällöin joko tietyllä virhekoodilla (DCOM) tai aiheuttaa poikkeuksen (CORBA, JavaRMI). Virheen jatkokäsittely jätetään näin asiakasohjelman vastuulle. Tavallisesti asiakasohjelma lopettaa tällaisessa virhetilanteessa joko kokonaan toimintansa tai jää odottamaan yhteyden uudelleenmuodostumista.

6.3.3 Yhteyden lopetus

Kun kaikki etäreferenssit palvelinohjelman omistamiin olioihin ovat vapautuneet, eikä ohjelmalla ole minkäänlaisia sisäisiä lukituksia estämässä sammumista, voi palvelinohjelma lopettaa yhteydet asiakasohjelmiin ja sulkea itsensä. Riippuen palvelimen toteutustavasta ohjelma joko antaa luvan alasajoon väliohjelmistolle (asiakasprosessiin liitettävät palvelimet) tai lopettaa itsenäisesti toimintansa (asiakasprosessin ulkopuoliset palvelimet). Useimmat väliohjelmistot, DCOM ja CORBA mukaan lukien, antavat lisäksi palvelinohjelman toteuttajalle mahdollisuuden vaikuttaa yhteyksien ja ohjelman lopetukseen vaikuttaviin seikkoihin sovelluskoodin tasolla.

Asiakasprosessiin liitettävät DCOM-palvelimet voivat säädellä alasajoaan toteuttamalla ja julkaisemalla `DllCanUnloadNow`-funktion, joka kertoo väliohjelmakirjastolle, voidaanko kyseinen palvelinmoduuli vapauttaa muistista. Ilman tätä funktiota palvelinmoduuli vapautetaan vasta lopuksi koko COM-järjestelmän alasajon yhteydessä. **Asiakasprosessin ulkopuoliset DCOM-palvelimet** voivat säädellä alasajoaan yksinkertaisesti estämällä poistumisen palvelinohjelman pääsilmukasta, kunnes halutut kriteerit täyttyvät. Alasajon alkaessa palvelinohjelma poistaa aluksi rekisteröimänsä oliot COM-väliohjelmakirjaston ajonaikaisesta rekisteritaulusta kutsumalla funktiota `CoRevokeClassObjects`. Lopuksi prosessi irtaantuu COM-väliohjelmistosta `CoUninitialize`-kutsulla. Tämänkaltaisen normaalin alasajon lisäksi palvelinprosessilla on mahdollisuus pakottaa ulkopuolisten asiakasprosessien etäreferenssit irti tietystä oliosta `CoDisconnectObject`-kutsulla. Tätä kuitenkin suositellaan käytettäväksi vain erikoistilanteissa [COM95, luku 6.4].

CORBA-palvelimet eivät välttämättä sisällä ollenkaan näkyvää pääsilmukkaa, jossa oliokutsuja käsiteltäisiin, vaan tämä tapahtuu sisäisesti käytetyn CORBA-implementaation oliovälittimessä. Tällöin palvelinohjelma toimii niin kauan, kunnes se käyttäjän toimesta erikseen lopetetaan. Lopetusprosessi käynnistetään oliovälittimen `orb::shutdown`-kutsulla, joka estää oliovälitintä jatkokäsittelemästä oliokutsuja sekä lakkauttaa kaikki oliovälittimen hallinnoimat oliosovittimet. Oliovälittimen sammuttua sen varaamat resurssit vapautetaan ja oliovälitin tuhotaan `orb::destroy`-kutsua käyttäen [OMG, luku 4.2.4].

6.4 Olion vapautus

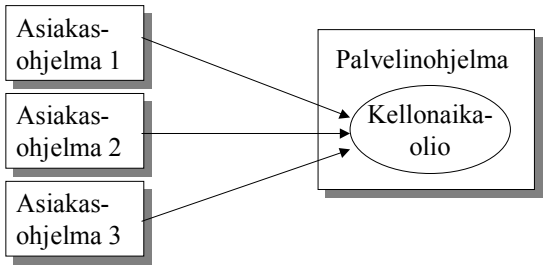
Kuten aiemmin luvussa 3.3.1 todettiin, asiakasohjelmat eivät käsittele hajautettuja olioita suoraan, vaan olion tarjoaman rajapinnan kautta. Tällöin myöskään olion vapautus ei voi olla suoraan asiakasprosessin hallinnassa. Asiakasohjelma voi myös keskeytyä epänormaalisti ja jättää ilmoittamatta rajapintojensa käytön lopettamisesta. Palvelinohjelman täytyisi pystyä tällaisissa tilanteissa havaitsemaan, milloin olioon ei ole enää referenssejä ja se voidaan vapauttaa. Tässä luvussa kuvattavat referenssilaskuri- sekä roskienkeruutekniikat on tarkoitettu tämän ongelman ratkaisuiksi.

6.4.1 Referenssilaskurit

Olioon viittaavien referenssien määrää pitää yllä **DCOM-väliohjelmistossa** oliokohtainen referenssilaskuri. Yleisenä sääntönä voidaan pitää sitä, että oliota ei voida vapauttaa niin kauan kuin siihen viittaa yksi tai useampi referenssi. Olion referenssien laskiessa nollaan, olio vapautetaan automaattisesti.

DCOM-asiakasohjelmien vastuulla on eksplisiittisesti hallinnoida käyttämiensä rajapintojen referenssejä. Referenssien käyttö tapahtuu kaikkien olioiden toteuttaman `IUnknown`-rajapinnan metodeilla `AddRef` ja `Release`, joiden sisäinen toteutus sijaitsee palvelinohjelman puolella. Ensimmäinen kasvattaa olion referenssilaskuria yhdellä, kun taas jälkimmäinen vähentää sitä yhdellä. `Release`-metodi huolehtii samalla olion vapauttamisesta, jos laskurin arvo laskee nollaan [COM95, luku 3.3].

Palvelinohjelman voi joskus olla tarpeen tehdä poikkeuksia yllä kuvattuun vapauttamislogiikkaan. Niin sanotut **singleton-oliot** eroavat tavallisista olioista siten, että niiden elinaika on sama kuin palvelinohjelman eikä niistä voi olla samanaikaisesti olemassa kuin yksi ilmentymä, jota kaikki asiakasohjelmat käyttävät. Esimerkkinä tällaisesta oliosta voisi olla kuvassa 22 kuvattu kellonajan kertova olio, joka pystyy palvelemaan useita asiakkaita samanaikaisesti. Luokan toteutuksessa täytyy tällöin huomioida moniajoon liittyvät synkronointiongelmat [Gri98, sivut 276–277].



Kuva 22: Singleton-tyyppinen kellonaikaolio.

CORBA-arkkitehtuurissa ei ole referenssilaskuria vastaavaa järjestelmää. Sen sijaan palvelimen tarjoamat oliot ovat periaatteessa aina tarjolla, kunhan vain palvelinohjelma on toiminnassa. CORBAn oliot ovat myös luonteeltaan pitempiaikaisia kuin DCOM-oliot, sillä CORBA tukee laajemmin olioiden varastointia (engl. *persistent objects*). Tämä tarkoittaa sitä, että asiakasohjelmat voivat viitata samalla etäreferenssillä samaan olioilmentymään vielä senkin jälkeen, kun palvelinohjelma on sammutettu ja käynnistetty uudelleen [ORB01, luku 5.5].

6.4.2 Roskienkeruu

Luvussa 6.3.2 kuvattua yhteyden katkeamista lievempi virhetilanne syntyy silloin, jos olioreferenssejä omistava asiakasohjelma lopettaa toimintansa vapauttamatta referenssejä oikeaoppisesti. Tällainen tilanne voi syntyä vaikkapa asiakasohjelmassa tapahtuneen vakavan virhetilanteen vuoksi. Palvelinohjelman pitäisi tällöin kyetä huomaamaan, mitkä oliot olivat kaatuneen asiakasohjelman omistuksessa ja kerätä nämä ”roskat” pois (engl. *garbage collection*) eli vapauttaa oliot muistista automaattisesti.

DCOM ratkaisee asian käyttämällä oliokohtaisia ping-kutsuja. Jokaisella palvelinohjelman ylläpitämällä oliolla oleva laskuri kertoo, milloin viimeksi olion ping-kutsuun on vastattu tai jotakin olion rajapinnan metodia kutsuttu. Kun laskuri ylittää tietyn aikarajan ja tarpeeksi moni ping-kutsu jää ilman vastausta[4], katsotaan kaikki olion rajapinnat vanhentuneiksi. Tällöin palvelinohjelma joko vapauttaa olion välittömästi tai laittaa sen jonoon odottamaan vapauttamista. Jälkimmäisessä tapauksessa palvelin voi aktivoida olion uudelleen, jos asiakasohjelma palaa toimintaan suhteellisen lyhyen katkoksen jälkeen.

Olioiden lukumäärän kasvaessa ping-kutsujen määrä saattaisi nousta yhteyttä kuormittavaksi tekijäksi, ellei DCOM kokoaisi samalle asiakkaalle välitettäviä ping-kutsuja ryhmiin. Ping-kutsut tapahtuvat tämän jälkeen näiden ryhmien välillä ja vain ryhmien muutoksista (uusista tai vapautetuista olioista) ilmoitetaan ping-kutsujen yhteydessä (engl. *delta-pinging*). Tällainen käytäntö vähentää yhteysliikennettä merkittävästi varsinkin silloin, jos olioryhmät ovat suhteellisen pysyviä [COM95, luku 15.1.5].

CORBA-spesifikaatio ei määrittele roskienkeruuta, mutta tietyin rajoituksin se on käytössä joissakin CORBA-implementaatioissa. Syynä automaattisen muistinsiivouksen puuttumiseen CORBA-spesifikaatiosta on mm. hyvin toimivan toteutuksen vaikeus, CORBAn riippumattomuus ohjelmointikielestä sekä referenssien erilaiset välitystavat. Esimerkiksi merkkijonomuodossa välitettävät IOR-referenssit (katso luku 6.1.2) eivät ole oliovälittimen suorassa valvonnassa, eikä se näin ollen voi tietää asiakasohjelmien etäreferenssien määrää.

[4] Oletuksena DCOM katsoo olion vanhentuneeksi kolmen vastaamattoman ping-kutsun jälkeen, joiden välillä kului aikaa 2 minuuttia. Täten olio vapautuu noin 6 minuutin kuluttua asiakasohjelman epänormaalista sammumisesta.

7 Etäkutsujen ja rajapintojen välitys

Kuten aiemmin on todettu, täytyy hajautetussa ympäristössä pystyä viittaamaan olioihin eri prosessien välillä. Jos olion ja sitä käyttävän asiakkaan prosessit toimivat eri muistiavaruuksissa, ei suoria muistiviittauksia voida käyttää, vaan kutsun sisältämä informaatio täytyy välittää kopioimalla. Kutsua ja sen mahdollisia parametreja ei kuitenkaan voida kopioida sellaisenaan, vaan niille täytyy tehdä **dataformaatin muunnos** (engl. *marshalling*). Siinä kutsu sarjallistetaan yhtenäiseksi tavuvirraksi (engl. *octet stream)* sovittua siirtosyntaksia käyttäen. Tietovirta voidaan tämän jälkeen kopioida asiakasprosessista kohdeprosessiin, jossa sille tehdään käänteinen operaatio ja suoritetaan varsinainen kutsu.

7.1 Esitystyyppien muunnos ja siirtosyntaksit

Hajautetun väliohjelmiston pitää kyetä välittämään verkon yli monimutkaistakin rakenteellista tietoa. Metodien parametrit voivat olla esimerkiksi kokonaislukuja, merkkijonoja (engl. *string)*, tietueita (engl. *struct*), yhdisteitä (engl. *union*) tai kokonaisia olioita. Ideaalitilanteessa kaikki väliohjelmiston käyttämän kuvauskielen tukemat ja määriteltävissä olevat tietotyypit ovat siirrettävissä.

Ennen siirtoa parametrit täytyy kuitenkin muuntaa muotoon, joka on laitteistoriippumaton ja välitettävissä käytettävällä siirtoprotokollalla. Muunnos tehdään jotakin ennalta sovittua siirtosyntaksia (engl. *transfer syntax)* käyttäen. Syntaksi kuvaa tarkasti aina bitti- ja tavutasolta lähtien tietotyyppien muodon siirron aikana. Luvuissa 7.1.1-7.1.4 esitellään joitakin yleisimpiä syntakseja. Lukujen pääasialliset lähteet ovat siirtosyntaksien määrittelyt [Sri95b], [DCE], [OMG, luku 15] ja [W3Cb, luku 3].

7.1.1 XDR (*External Data Representation*)

Melko yksinkertainen ja laajasti käytetty siirtosyntaksi on XDR, joka kehitettiin alunperin ONC/RPC-protokollaa (katso luku 4.2.1) varten. Sen päälle rakennetuista sovelluksista tunnetuimpia on NFS (*Network File System,* katso luku 4.2.2), joka mahdollistaa tiedostojen jakamisen hajautetuissa ympäristöissä [Sri95a].

XDR on tarkkaan määritelty kieli, jonka avulla voidaan kuvata verkon yli välitettävät tietotyypit ja rakenteet. Kieltä noudattavan kuvaustiedoston avulla pystytään automaattisesti generoimaan sovelluskoodia, joka osaa pakata ja purkaa tietotyypit verkon yli siirtoa varten. Esimerkki tällaisesta kuvaustiedostosta löytyy liitteestä 3.

Taulukossa 5 on esitetty XDR:n tukemat perustietotyypit eli primitiivit.

Tietotyyppi	**Primitiivi**	**Pituus**
Kokonaisluvut	`integer`	32 bittiä
Enumeraatio	`unsigned integer`	32 bittiä
	`hyper integer`	64 bittiä
	`hyper unsigned integer`	64 bittiä
	`enumeration`	32 bittiä
Totuusarvo	`boolean`	32 bittiä
Numeroarvo	`double`	64 bittiä
	`quadruple`	128 bittiä
	`hyper unsigned integer`	64 bittiä
Merkkijono	`string`	8 bittiä / tavu
Tulkitsematon tieto	`opaque`	8 bittiä / tavu

Taulukko 5. XDR:n primitiivit.

Kielen syntaksin perusyksikkö on yksi tavu (oktetti) ja sen oletetaan olevan siirrettävissä erilaisten laitteistojen välillä. Jokainen tuettu tietotyyppi vaatii aina neljän tavun monikerran verran tilaa. Jos tietotyypin koko poikkeaa tästä, lisätään tavuvirtaan tyhjiä tavuja ehdon toteuttamiseksi.

Taulukossa 5 olevien esitystyyppien lisäksi XDR tukee myös konstruoituja tietotyyppejä, kuten taulukoita, tietueita ja yhdisteitä. Nämä tietotyypit voidaan koostaa yhdestä tai useammasta primitiivistä.

7.1.2 NDR (*Neutral Data Representation*)

Toinen laajasti hyödynnetty siirtosyntaksi on NDR. Muun muassa DCE/RPC-protokolla (katso luku 4.2.1) ja DCOM käyttävät tätä syntaksia tiedonvälityksessään. Formaatti esitellään tässä lyhyesti - täydellinen kuvaus löytyy DCE:n RPC-protokollan spesifikaatiosta [DCE].

Yleisellä tasolla NDR tarjoaa syntaksin, jolla IDL-kielen kuvaamat tietorakenteet voidaan muuntaa 8-bittiseksi tavuvirraksi, joka puolestaan voidaan lähettää verkon yli RPC-protokollaa käyttäen. NDR tukee lisäksi erilaisia esitysmuotoja tukemilleen tietotyypeille mahdollistaen näin protokollan toiminnan erilaisten laitteistojen välillä. Taulukossa 6 on lueteltu NDR:n tukemat yksinkertaiset tietotyypit.

Tyyppi	Primitiivi	Pituus
Totuusarvo	`boolean`	8 bittiä
Kokonaisluvut	`small`	8 bittiä
	`short`	16 bittiä
	`long`	32 bittiä
	`hyper`	64 bittiä
Liukuluvut	`single`	32 bittiä
	`double`	64 bittiä
Merkki	`ASCII char`	8 bittiä
	`EBCDIC char`	8 bittiä
Tulkitsematon tieto	`uninterpreted data`	8 bittiä / tavu

Taulukko 6. NDR:n primitiivit.

Kokonaislukujen tavujärjestys (onko eniten merkitsevä tavu ensimmäisenä vai viimeisenä luvussa) sekä liukulukujen tarkempi muoto (IEEE-, Vax-, IBM- tai Cray-muoto) määrätään yhteydenoton alkaessa lähetettävässä tietueessa (*NDR Format Label*). Tietue kertoo myös, onko välitettävät merkkijonot koodattu yleisellä ASCII-standardilla, vaiko IBM:n EBCDIC-standardilla.

Monimutkaisemmat NDR:n tukemat tietorakenteet koostuvat yhdestä tai useammasta NDR:n primitiivistä. Tuettuja rakenteita on kuvattu taulukossa 7.

Tyyppi	Kuvaus
`array`	Yksi- tai useampiulotteinen taulukko primitiivejä tai muita rakenteita varten.
`string`	Vapaamittainen merkkijono.
`struct`	Joukko primitiivejä tai muita rakenteita.
`union`	Joukko primitiivejä tai muita rakenteita, joista vain yksi on kerrallaan validi.
`pipe`	Järjestetty sekvenssi peräkkäisiä primitiivejä tai muita rakenteita, joiden elementtien määrä ei ole rajoitettu.
`pointer`	Osoitin johonkin samassa siirrossa välitettyyn primitiiviin tai rakenteeseen.

Taulukko 7. NDR:n rakenteelliset tyypit.

Rakenteet `array`, `string`, `struct` ja `pipe` voivat olla kooltaan kiinnitettyjä tai muuttuvia. Muuttuvien rakenteiden koko voi vaihdella ajon aikana, ja siksi tieto rakenteen koosta täytyy välittää aina rakenteen mukana.

Kaikki NDR:n tietotyypit, primitiivit sekä tietorakenteet, on tasattu pääsääntöisesti oman kokonsa monikertaan tavuvirrassa. Tämä tarkoittaa sitä, että `X`:stä tavusta koostuvan tietotyypin alkupaikka täytyy löytyä tavuvirran paikasta `nX`, missä `n` on kokonaisluku. Tällainen sopimus nopeuttaa tavuvirran käsittelyä, vaikkakin virtaan saatetaan joutua toisinaan lisäämään tyhjiä elementtejä tasauksen aikaansaamiseksi.

7.1.3 CDR (*Common Data Representation*)

Kolmas esiteltävä siirtosyntaksi on CDR. Se on mm. perustana CORBAn GIOP-liitynnälle, joka välittää viestejä erilaisten CORBA-runkototeutusten välillä. Pääpiirteiltään CDR muistuttaa luvuissa 7.1.1 ja 7.1.2 esiteltyjä XDR- ja NDR-kuvaustapoja. Sen ominaisuuksia ovat mm. seuraavat:

- Se tukee kaikkia OMG IDL -kuvauskielen tarjoamia tietotyyppejä.
- Se tukee vaihtelevia tavujärjestyksiä välittämällä tietoa lähettäjän käyttämästä tavujärjestyksestä.
- Se tukee tietuetyyppien tasausta luonnollisiin tavurajoihin käsittelyn nopeuttamiseksi.

CDR:n primitiivit vastaavat melko tarkasti C- ja C++ -kielen tietotyyppejä. Tuettuja primitiivejä ovat mm. `char`, `boolean`, `long`, `unsigned long`, `float`, `double` ja `enum`. Konstruoiduista tietotyypeistä tuettuja ovat mm. `struct`, `array`, `union` ja `string`. Edellisten lisäksi CDR tukee myös eräitä CORBAn ns. ”valeolioita” (engl. *pseudo-object*) ja osaa välittää ne oktettivirrassa. Tällaisia valeobjekteja ovat esimerkiksi olioreferenssit ja poikkeukset.

7.1.4 XML (*Extended Markup Language*)

Melko uutena vaihtoehtona edellä mainituille siirtosyntakseille on yleistynyt XML-kielen hyödyntäminen tiedon välityksessä. XML-standardi [W3Ca] ei itsessään sisällä kuvausta siitä, kuinka erilaiset tietotyypit tulisi kielen avulla kuvata. Tätä tarkoitusta varten onkin alettu luoda uusia määrittelyjä.

Tunnetuin kehittäjä on W3C-yhteenliittymä (*World Wide Web Consortium*), jonka XML Skeema -spesifikaatio määrittää tarkasti rajatun joukon tietotyyppejä ja niiden esitysmuodon XML-kielellä [W3Cb]. Esimerkiksi taulukon 8 yksinkertaiset tietotyypit ovat skeemassa tuettuina.

Tietotyyppi	Kuvaus	Esimerkkejä
`string`	Vapaa merkkijono	`"  Hei  Maailma  "`
`token`	Kuten `string`, mutta erikoismerkit (rivinvaihto, tabulaattori) on korvattu yhdellä tyhjällä sekä alku- ja loppuosa siistitty.	`"Hei Maailma"`
`short`	Lyhyt rajoitettu kokonaisluku	`-12365, -1, 0, 1, 12365`
`byte`	Tavu	`-67, -1, 0, 1,67`
`unsignedByte`	Etumerkitön tavu	`0, 1, 67, 236`
`base64Binary`	Vapaa binääridata koodattuna base64-menetelmällä.	`GpM7`
`integer`	Rajoittamaton kokonaisluku	`-123658, -1, 0, 1, 123658656456853`
`positiveInteger`	Rajoittamaton positiivinen kokonaisluku	`1, 123658`

Taulukko 8. XML Skeeman yksinkertaiset tietotyypit (jatkuu).

Tietotyyppi	Kuvaus	Esimerkkejä
`negativeInteger`	Rajoittamaton negatiivinen kokonaisluku	`-123658, -1`
`long`	Rajoitettu kokonaisluku	`-1, 123658321`
`unsignedLong`	Etumerkitön kokonaisluku	`0, 1, 123658321`
`decimal`	Rajoittaman desimaaliluku	`-1.23, 0, 1.23, 3.357`
`float`	Rajoitettu desimaaliluku[1]	`-INF, -1E4, -1.23, 0, 1.24, 12.78E5, INF, NaN`
`double`	Rajoitettu desimaaliluku[2]	`-INF, -1E4, -1.23, 0, 1.24, 12.78E5, INF, NaN`
`boolean`	Binäärinen totuusarvo	`true,false,1,0`
`time`	Joka päivä tapahtuva aikaleima varustettuna mahdollisella aikavyöhykkeellä[3]	`13:20:00.000,` `15:30:00.000-05:00`

Taulukko 8. XML Skeeman yksinkertaiset tietotyypit (jatkoa).

Tietotyyppi	Kuvaus	Esimerkkejä
`dateTime`	Tietty ajanhetki varustettuna aikavyöhykkeellä[3]	`1999-05-31T13:20:00.000-05:00`
`duration`	Ajankesto[3]	`P1Y2M3DT10H30M12.3S`
`date`	Päivämäärä[3]	`1999-05-31`
`Name`	XML-nimi[4]	`shipTo`
`anyURI`	Referenssi viittaus mahdollisella tarkenteella	`http://www.comp.com/`, `http://www.example.com/doc.html#ID5`

(1) Standardin IEEE 754-1985 mukainen 32-bittisen liukuluvun esitystapa. (2) Standardin IEEE 754-1985 mukainen 64-bittisen liukuluvun esitystapa. (3) Standardin ISO8601 mukainen esitystapa. (4) Mikä tahansa lähteessä [W3Ca] esitetyn `Name`-symbolin sallittu arvo.

Taulukko 8. XML Skeeman yksinkertaiset tietotyypit (jatkoa).

XML Skeema mahdollistaa myös uusien tietotyyppien julistamisen perimällä ne jo olemassa olevista ja muuttamalla jokaiseen tietotyyppiin liittyviä rajoitteita (engl. *constraining facets*). Moni yllä olevan taulukon 8 tietotyypeistäkin on muodostettu tällä tavalla. Esimerkiksi `byte`-tyyppi on peritty `short`-tyypistä muokkaamalla rajoitteiden `minInclusive` ja `maxInclusive` arvoja, jotka määräävät sallitut ala- ja ylärajat tietotyypille.

Käyttämällä XML-kieltä tietotyyppien välitykseen saavutetaan useita etuja. Ensinnäkin XML on tekstimuotoista dataa, jota voidaan siirtää helposti järjestelmien välillä eri siirtotapoja kuten HTTP-, SMTP- tai FTP-protokollia käyttäen. Toinen etu on tiedonkulun läpinäkyvyys, sillä tekstimuotoinen XML-kuvaus on helposti luettavassa muodossa, jolloin tiedonkulun tarkkailu ja mahdollinen virheenpaikannus on yksinkertaisempaa. Kolmantena

etuna on XML-kielen automaattinen validointimahdollisuus käyttämällä DTD-kuvauksia (*Document Type Definition)* tai ennalta määriteltyjä XML-skeemoja. Niillä voidaan mahdollistaa myös versiointi eri toteutusten välillä. Saavutetun avoimuuden eräs haittapuoli on kuitenkin siirrettävän tiedon määrän kasvu verrattuna binääriformaatteihin perustuviin kuvauskieliin [And00, sivut 497–501].

7.2 Sarjallistaminen ja tiedon välitys

Yleensä väliohjelmiston monimutkaisin tekninen osa liittyy sarjallistamisen toteutukseen. Siinä mielivaltaisen monimutkainen tietorakenne (monimutkaisuutta rajoittaa ainoastaan väliohjelmiston käyttämän kuvauskielen tukemat tietotyypit) täytyy saada muunnettua yhtenäiseksi tavuvirraksi, välitettyä prosessilta toiselle sekä palautettua jälleen takaisin alkuperäiseksi tietorakenteeksi. Tässä luvussa kuvataan yksityiskohtaisemmin niitä operaatioita ja toimenpiteitä, joiden avulla sarjallistaminen toteutetaan. Lähteinä luvussa on käytetty DCOM- ja CORBA-väliohjelmistojen määrittelyitä [COM95], [COM98] sekä [OMG].

7.2.1 Sarjallistamiskomponentit

Monissa toteutuksissa sarjallistamisen suorittavat rutiinit on eristetty olioiden toteutuksista omiin moduuleihin eli tynkiin (engl. *proxy, stub* tai *skeleton).* Tämä antaa käyttäjälle mahdollisuuden toteuttaa omia oliokohtaisia sarjallistamisrutiineja, jos siihen on tarvetta. Syitä tähän voivat olla esim. tarve kontrolloida koko sarjallistamisprosessia jotakin muuta kuin väliohjelmiston tukemaa siirtoprotokollaa käyttäen tai halu optimoida tiedon pakkaus- ja purkurutiinit datan sisällön perusteella tehokkaammiksi. Jos oliot ovat muuttumattomia tai säilyttävät kaiken tilatietonsa jaetussa muistissa, tarjoavat omat sarjallistamisrutiinit myös mahdollisuuden yksinkertaistaa olioiden sarjallistamista eri prosessien välillä.

DCOMissa sarjallistamiskomponentit ovat itsessään olioita, jotka toteuttavat `IMarshall`-rajapinnan. Niiden käyttöön on kehittäjällä kolme eri mahdollisuutta. Yksinkertaisin tapa on hyödyntää DCOMin **oletussarjallistamista** (engl. *standard marshalling*). Tällöin DCOM huolehtii olion rajapintojen ja kutsujen välityksestä, kunhan

olio toteuttaa jonkin DCOMin sarjallistamisesta huolehtivan vakiorajapinnan, kuten `IDispatch` tai `IDataObject`. Toinen mahdollisuus on käyttää **räätälöityä sarjallistamista** (engl. *custom marshalling*), jolloin ohjelmoija kirjoittaa kokonaan omat `IMarshall`-rajapinnan toteuttavat tynkäluokat. Kolmas mahdollisuus sarjallistamisen toteutukseen on **käsittelijäsarjallistaminen** (engl. *handler marshalling),* joka on kompromissi kahden aiemmin mainitun tavan välillä. Tällöin kaikki olion kutsut kulkevat erityisen ohjelmoijan toteuttaman käsittelijän läpi, joka voi toteuttaa erikoisoperaatioita kutsujen johdosta. Käsittelijä voi kuitenkin aina halutessaan ohjata kutsut takaisin oletussarjallistajalle [COM98, sivut 17–22].

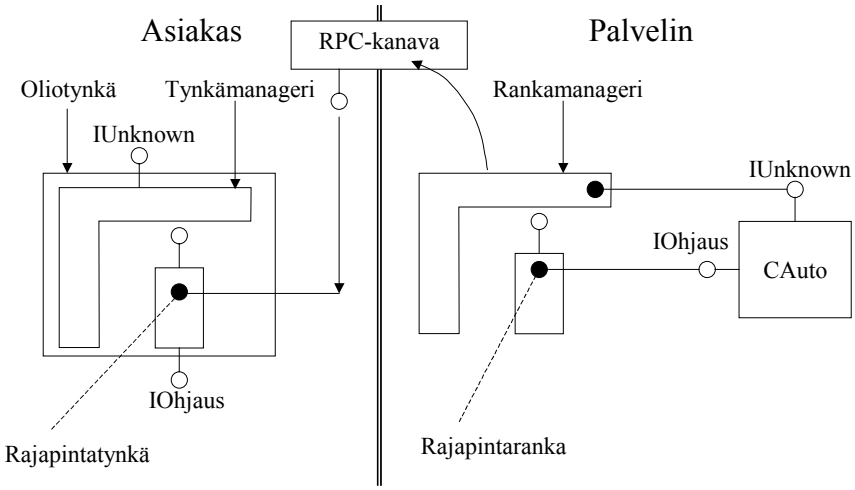
CORBAn sarjallistamiskomponentit eivät ole yhtä dynaamisia kuin DCOMin, vaan niiden lähdekoodi generoidaan automaattisesti IDL-käännöksen yhteydessä ja linkataan staattisesti ohjelmaan mukaan. Ennen käännöstä lähdekoodia on tietenkin mahdollista muokata haluamakseen, mutta jotkin CORBA-toteutukset tarjoavat parempiakin tapoja. ORBacus-implementaatiossa on mahdollista periä omia **älykkäitä tynkiä** (engl. *smart proxy*) automaattisesti generoiduista tyngistä ja tätä kautta vaikuttaa sarjallistamisprosessiin. Samaan luokkaan voi jopa liittää useita eri tynkiä, joista ajonaikana sitten valitaan käytettävä tynkä tilanteen mukaan [OMG, luku 2.1] [ORB02].

7.2.2 Rajapintojen välitys

Ennen kuin ensimmäistäkään hajautettua metodikutsua voidaan välittää väliohjelmiston kautta, täytyy asiakasprosessin saada referenssi palvelimella sijaitsevaan olioon tai sen rajapintaan. Referenssin voi palauttaa jokin olioiden luomiseen käytettävä väliohjelmiston funktio, kuten DCOMin `CoCreateInstance`, tai jonkin jo olemassa olevan rajapinnan metodi.

DCOMissa rajapinnan välitykseen kuuluu aina rajapintaan liittyvien sarjallistamiskomponenttien luonti. Oletetaan, että meidän täytyy palauttaa palvelimella sijaitsevan `CAuto`-olion `IOhjaus`-rajapintaosoitin (katso kuva 23) asiakassovellukselle oletussarjallistusta (katso luku 7.2.1) käyttäen. Aluksi palvelimella toimiva ja väliohjelmistoon kuuluva rankamanageri paikallistaa systeemirekisteristä rajapinnan IID-

tunnuksen avulla oikean sarjallistamiskomponentin ja luo tämän kautta rankaluokan. DCOMissa sarjallistamiskomponentit sisältävät aina sekä sarjallistamis- että purkurutiinit. Tästä johtuen rankamanageri voi seuraavaksi pakata `IOhjaus`-osoittimen rankaluokan avulla ja välittää sarjallistetun datan RPC-kanavalle, jonka kautta itse siirto tapahtuu. Rajapinnan sarjallistamisessa pakataan tietovirtaan rajapinnan IID-tunnuksen lisäksi palvelimen OXID-tunnus, olion OID-tunnus sekä rajapinnan prosessikohtainen PID-tunnus (katso luku 5.3.1). Näiden avulla asiakassovellus voi ottaa myöhemmin yhteyden kyseiseen rajapintaan.



Kuva 23. DCOM-rajapintaosoittimen välitys.

Asiakassovelluksen puolella RPC-kanava on yhteydessä asiakkaan tynkämanageriin ja havaitessaan uuden `IOhjaus`-rajapintaosoittimen vastaanotetussa tietovirrassa, käskee se tynkämanageria luomaan uuden sarjallistamiskomponentin rajapintaa varten. Tynkämanageri osaa palvelimen rankamanagerin tapaan luoda systeemirekisteristä löytyvän rajapinnan IID-tunnuksen perusteella oikean tynkäluokan, jonka avulla se purkaa (engl. *unmarshall*) saapuneen sarjallistetun datan. Lopuksi tynkämanageri kysyy tynkäluokalta `QueryInterface`-metodilla varsinaista rajapintaosoitinta ja saa vastaukseksi `IOhjaus`-osoittimen, joka välitetään asiakassovellukselle. Etäkutsun välitykseen osallistuvat DCOM-väliohjelmiston osat on esitetty kuvassa 23 [COM95, luvut 7.1–7.3].

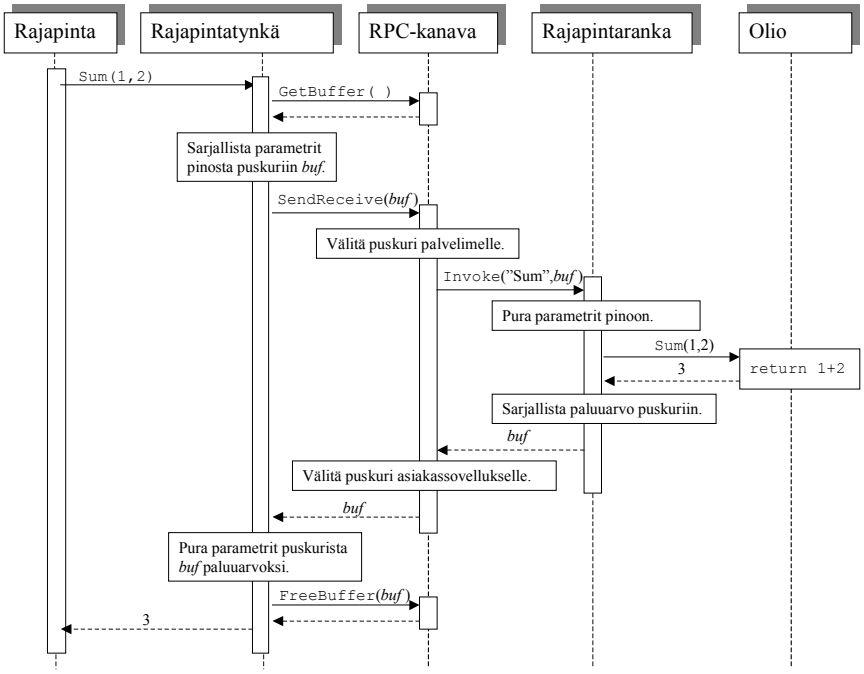
CORBA välittää rajapintaosoittimien sijaan etäolioreferenssejä. Ensimmäisen olion etäreferenssin saaminen voi tapahtua eri tavoin, kuten tietokantaa, jaettua tiedostoa tai CORBAn nimipalvelua (*Naming Service*) käyttäen. Kaikissa näissä tapauksissa palvelin muuntaa etäreferenssin toteutuskohtaisesta esitystavasta standardoituun merkkijonomuotoon, ennen kuin välittää sen asiakassovellukselle. Asiakassovellus tekee referenssille käänteisen muunnoksen ja pystyy tämän jälkeen kysymään palvelimelta varsinaista olio-osoitinta, jonka rajapinnan kautta etäkutsut tapahtuvat. Olioreferenssin välitys etäkutsun mukana (joko kutsun kohteena tai parametrina) tapahtuu hajautusprotokollakohtaista olioavainta (engl. *object key*) käyttäen. Olioavaimen sisältö voi vaihdella siirtoprotokollasta riippuen, mutta yleistä GIOP-hajautusprotokollaa käytettäessä sen täytyy olla IOR-rakenteen (katso luku 6.1.2) mukaisessa muodossa [OMG, luvut 4.5.2, 13.6.9 ja 15.3.6].

CORBA sisältää sarjallistamiskomponentit DCOMin tapaan, mutta nämä ovat luonteeltaan staattisempia ja tiukemmin sidoksissa olion toteutukseen. Asiakaspään sarjallistamiskomponenttia nimitetään CORBA-terminologiassa asiakastyngäksi (engl. *stub)* ja palvelinpään komponenttia palvelintyngäksi tai rungoksi (engl. *skeleton*). Näiden tehtävät ovat hyvin pitkälle samat kuin niiden DCOM-vastineilla [OMG, luku 2.1].

7.2.3 Kutsujen välitys

Asiakassovelluksen saatua referenssin palvelimella olevan olion rajapintaan, voi se kutsua kyseisen rajapinnan metodeja ja välittää kutsun mukana parametreja. Kutsun tekoon liittyy yleisesti eri väliohjelmistoissa samanlaiset vaiheet, jotka ovat kutsun sarjallistaminen, kutsun välitys palvelimelle, sarjallistetun kutsun purku, itse kutsun suoritus sekä paluuarvojen välitys takaisin asiakkaalle.

DCOM toteuttaa etäkutsun kuvan 24 mukaisesti.

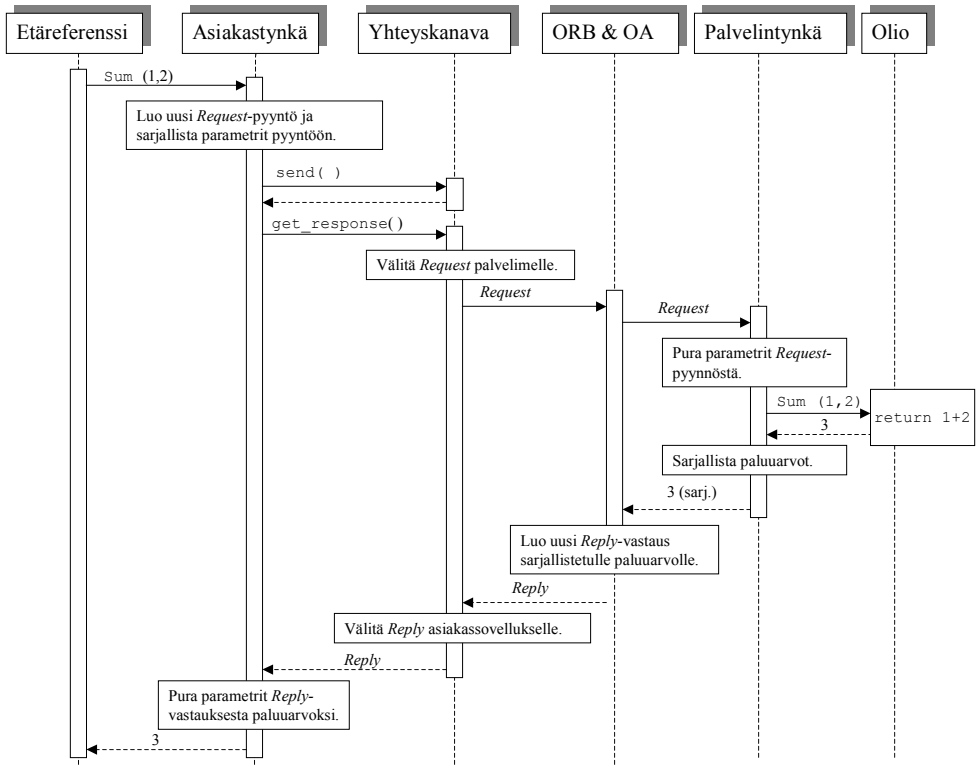


Kuva 24. DCOM-etäkutsun välitysvaiheet.

Aluksi kutsu ohjautuu rajapintatyngälle, joka sarjallistaa kutsun parametrit pinosta muistipuskuriin. Seuraavaksi tynkä kutsuu RPC-kanavan `SendReceive`-metodia. Tämän jälkeen sekä kutsu että sen parametrit välittyvät palvelimelle ORPC-protokollaa käyttäen. Palvelimella RPC-kanava on yhteydessä rajapintarankaan ja ohjaa sille saapuneen kutsun. Ranka purkaa sarjallistetut parametrit takaisin palvelinohjelman pinoon ja kutsuu varsinaista toteutusta. Mahdolliset paluuarvot ranka sarjallistaa takaisin puskuriin ja antaa sen RPC-kanavalle välitettäväksi asiakassovellukselle. Asiakassovelluksen tynkä purkaa paluuarvot puskurista, vapauttaa puskuria varten varatun muistitilan ja palauttaa arvot kuin ne olisivat syntyneet paikallisen kutsun seurauksena [COM95, luvut 7.3 ja 7.7.4.3].

CORBAn kuvassa 25 havainnollistettu etäkutsun välitys on pääpiirteissään samanlainen kuin DCOMin. Ensimmäiseksi etäkutsu ohjautuu olion asiakastyngälle, joka luo uuden *Request*-pyynnön ja sarjallistaa siihen kutsun parametrit. Tämän jälkeen tynkä lähettää `Request:send`-metodilla kutsun GIOP-yhteyskanavalle ja jää odottamaan vastausta `Request::get_request`-metodilla. Yhteyskanava välittää pyynnön palvelimen

oliosovittimelle (OA, katso luku 5.1.2), joka etsii oikean palvelintyngän ja ohjaa sille *Request*-pyynnön. Palvelintynkä purkaa parametrit pyynnöstä ja kutsuu varsinaista toteutusta. Kutsun mahdolliset paluuarvot palvelintynkä sarjallistaa ja antaa ne oliovälittimelle (ORB, katso luku 5.1.2). Oliovälitin luo uuden *Reply*-vastauksen, johon se tallentaa sarjallistetut paluuarvot. Seuraavaksi *Reply*-vastaus siirtyy yhteyskanavan avulla takaisin asiakastyngälle, joka purkaa vastauksen ja asettaa kutsun paluuarvot kohdalleen [OMG, luvut 7.3, 15.4 ja 15.5].



Kuva 25. CORBA-etäkutsun välitysvaiheet.

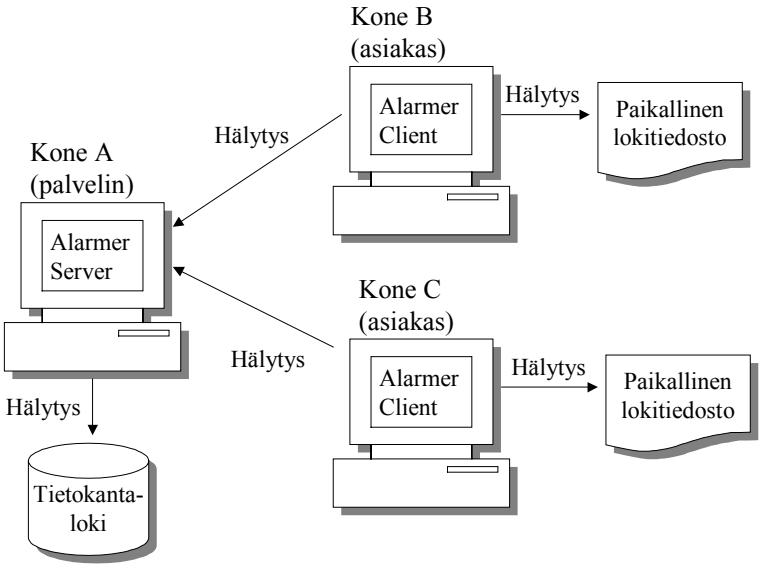
8 Käytännön esimerkki: Generis Alarmer

Hajautetun väliohjelmiston käyttöä todellisessa sovelluskehityksessä tarkastellaan Generis Alarmer -ohjelmiston toteutuksen pohjalta. Esimerkissä on keskitytty tutkielman aiheen kannalta olennaiseen eli ohjelmiston hajautukseen liittyviin yksityiskohtiin ja jätetty muilta osin ohjelmiston tarkastelu vähemmälle huomiolle.

Luvussa kuvataan aluksi Generis Alarmer -ohjelmiston rakennetta ja käyttötarkoitusta. Näiden jälkeen määritellään ohjelmistolle asetetut vaatimukset sekä kuinka hajautus suunniteltiin toteutettavaksi. Lopuksi esitellään ohjelmiston testaussuunnitelma.

8.1 Ohjelmiston kuvaus

Generis Alarmer on suunniteltu virhe- ja hälytysviestien keskitettyyn käsittelyyn. Sen tarkoituksena on kerätä eri työasemilla toimivilta Generis-ohjelmilta viestit yhteiseen tietokantaan, josta niitä voidaan myöhemmin tarkastella sitä varten suunnitellulla käyttöliittymällä. Hälytyksiä tuottavilla asiakasohjelmilla on myös mahdollisuus tallentaa viestit paikalliseen lokitiedostoon. Generis Alarmerin rakenteen yleiskuva selviää kuvasta 26.



Kuva 26. Generis Alarmerin yleiskuva.

Ohjelmisto rakentuu palvelimella sijaitsevasta Alarmer Server -sovelluksesta sekä yhdestä tai useammasta asiakassovelluksesta. Asiakassovellukset kommunikoivat palvelimen kanssa Alarmer Client -moduulia käyttäen.

8.2 Vaatimukset

Tässä luvussa määritellään, minkälaisia vaatimuksia Generis Alarmer -ohjelmistolle asetettiin yrityksen puolelta ennen toteuttamista. Tärkeimmät vaatimukset liittyivät suorituskykyyn sekä luotettavuuteen virhetilanteissa.

8.2.1 Käyttöympäristö

Ohjelmiston tuli toimia seuraavilla Microsoft Windows -käyttöjärjestelmän versioilla:

- Windows NT 4.0,
- Windows NT 4.0 Terminal Server,
- Windows 2000 ja
- Windows 2000 Terminal Server.

Lisäksi Alarmer Server -sovelluksen tuli toimia klusteriympäristöissä. Klusteriympäristö on kahdesta tai useammasta palvelinkoneesta rakennettu järjestelmä, jossa ajossa olevat prosessit voivat vaihtaa konetta ilman ulospäin näkyviä katkoksia. Klustereita käytetään pääsääntöisesti kohteissa, joissa vaaditaan erityisen hyvää virhetilanteiden sietokykyä.

8.2.2 Suorituskyky

Ohjelmiston suorituskyvyn haluttiin olevan hyvä erityisesti tilanteissa, joissa hälytysviestien määrä kasvaa hetkellisesti suureksi. Esimerkiksi tietokannassa tapahtuvat virheet saattavat poikia ongelmia useissa eri sovelluksissa ja aiheuttaa suuria väliaikaisia viestimääriä.

Ohjelmiston suorituskyvylle asetettiin seuraavat vaatimukset:

- Palvelimen täytyy pystyä palvelemaan sataa eri asiakassovellusta samanaikaisesti.
- Palvelimelle täytyy pystyä välittämään yksittäiseltä asiakassovellukselta kymmenen hälytysviestiä sekunnissa.
- Palvelimelle täytyy pystyä välittämään kaikkien asiakassovellusten hälytysviestit yhteenlaskettuna tuhat viestiä sekunnissa.

8.2.3 Luotettavuus

Koska ohjelmiston tarkoitus oli käsitellä muissa sovelluksissa syntyneitä virheitä, täytyi sen itse kyetä toipumaan vakavistakin virhetilanteista. Alarmer Server -sovelluksen täytyi pystyä jatkamaan toimintaansa seuraavien virhetilanteiden jälkeen:

- verkkoliikennekatkos,
- tietokannan alasajo tai tietokantayhteyden katkos tai
- asiakassovelluksen epänormaali irtikytkeytyminen.

Tietokannan ollessa alhaalla palvelimen täytyi yhä pystyä keräämään hälytyksiä asiakassovelluksilta ja tallentamaan ne tietokantaan heti, kun kanta palaa toimintaan. Palvelinsovelluksen täytyi myös raportoida virhetilanteet omaan paikalliseen lokitiedostoon jälkikäteen tapahtuvaa analysointia varten.

8.2.4 Turvallisuus

Turvallisuusvaatimuksissa todettiin riittävän, että käytettävän väliohjelmiston toteutus ei sisällä tunnettuja turva-aukkoja, ja että väliohjelmistoa on mahdollista käyttää palomuurin lävitse. Jälkimmäinen vaatimus tarkoittaa käytännössä sitä, että väliohjelmiston käyttämät tietoliikenneportit ovat vakioita tai ne ovat konfiguroitavissa. Koska järjestelmän käyttökohteet tulevat pääsääntöisesti olemaan lähiverkkojen sisällä, eikä välitettävä aineisto ole sisällöltään arkaluonteista, siirrettävien viestien ei tarvinnut olla salattuja.

8.3 Hajautuksen suunnittelu

Hajautuksen suunnitteluun kuului järjestelmän arkkitehtuurin valinta, luokkamallien ja rajapintojen suunnittelu sekä ohjelman sisäiseen toteutukseen liittyvien ratkaisujen teko. Suunnitteluvaiheessa päätettiin myös, mitä väliohjelmistoa ja työkaluja kehityksessä käytetään.

8.3.1 Järjestelmän arkkitehtuuri

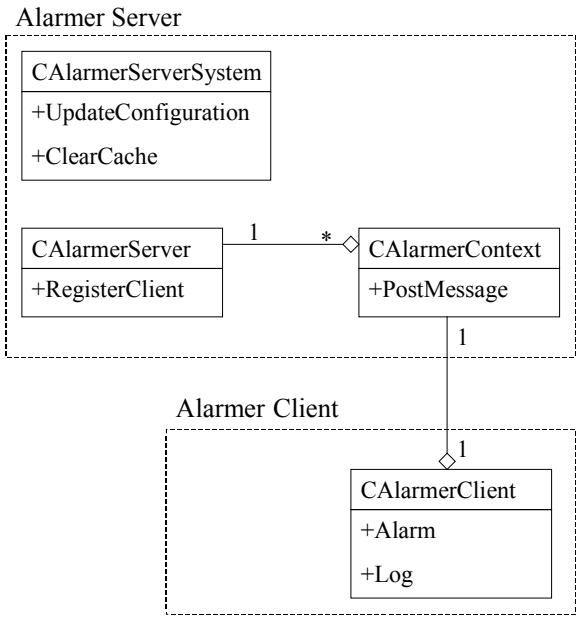
Generis Alarmer -ohjelmiston hajauttaminen oli tarpeen, sillä asiakassovelluksia ei tahdottu tehdä riippuvaisiksi tietokannasta. Alarmer Client -moduuli haluttiin pitää mahdollisimman yksinkertaisena ja siirtää viestien käsittelyyn ja tallennukseen liittyvä logiikka Alarmer Server -palvelinsovelluksen puolelle. Tämä pienentäisi mahdollisuutta, että Alarmer Client -moduuli aiheuttaisi virhetilanteita asiakassovelluksissa.

Ohjelmiston järjestelmäarkkitehtuuriksi valittiin luvussa 3.2.1 esitelty **asiakas-palvelinmalli**, jossa kaikki asiakassovellukset ovat yhteydessä samaan palvelinsovellukseen. Malli ei estä konfiguroimasta useampia palvelinsovelluksia eri palvelinkoneille, mutta asiakassovelluksille palvelinkoneen osoite pysyy aina vakiona. Klusteriympäristöissä palvelinsovellus voi kuitenkin vaihtaa palvelinkonetta esimerkiksi palvelimella tapahtuneen virheen vuoksi ilman, että asiakassovellukset sitä huomaavat. Tätä kautta asiakas-palvelinarkkitehtuurista voi räätälöidä hyvin luotettavan, kunhan käyttöympäristö muokataan oikeaksi.

Alarmer Server -palvelinsovellus päätettiin toteuttaa **monisäiearkkitehtuuria** hyödyntäen. Tämä mahdollistaa usean asiakkaan rinnakkaisen palvelemisen samanaikaisesti ja ratkaisu useimmissa tapauksissa parantaa ohjelman suorituskykyä. Jos valittu väliohjelmisto ei kuitenkaan sisäisesti tue monisäiearkkitehtuuria, jää saavutettu hyöty tällöin pieneksi, koska kaikki pyynnöt asiakassovelluksilta joudutaan synkronoimaan sarjallisiksi kutsuiksi ennen niiden käsittelyä. Monisäiemalli aiheuttaa jonkin verran lisätyötä ohjelman suunnittelussa ja toteutuksessa, koska osia ohjelmakoodista joudutaan suojaamaan säikeiden yhtäaikaiskäytöltä.

8.3.2 Luokkamalli ja rajapinnat

Generis Alarmer -järjestelmän hajautettu luokkamalli koostuu neljästä luokasta, jotka on esitetty kuvassa 27. Luokka `CAlarmerServerSystem` tarjoaa metodit Alarmer Server -sovelluksen ajonaikaiseen hallintaan käyttöliittymästä käsin. Sen `UpdateConfiguration`-metodilla voi käskeä sovellusta lukemaan toiminta-asetuksensa uudelleen ja `ClearCache`-metodilla tyhjentämään viestien väliaikaisvaraston.

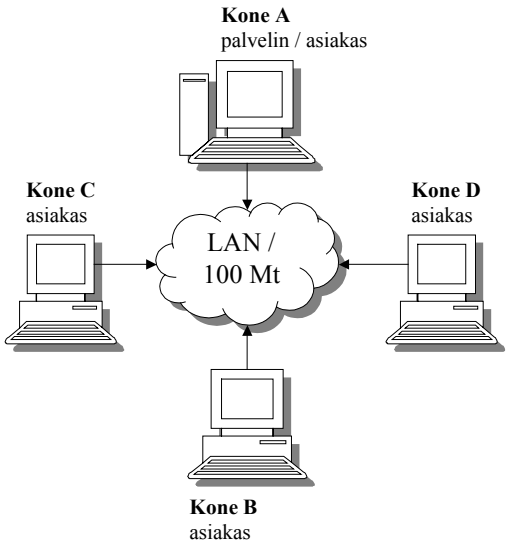


Kuva 27. Generis Alarmer -järjestelmän UML-luokkadiagrammi.

Luokan `CAlarmerServer` kautta asiakassovellukset saavat rekisteröityä itsensä hälytysviestien lähteeksi ja kerrottua palvelimelle sijaintinsa sekä sovellustunnuksen, jota käytetään tietokannassa identifioimaan viestejä. Jokaista rekisteröityä asiakassovellusta kohden palvelin luo uuden ilmentymän `CAlarmerContext`-luokasta, jonka `PostMessage`-metodin avulla hälytyksien lähetys palvelimelle tapahtuu. Asiakassovellukset eivät kuitenkaan käytä suoraan tätä luokkaa, vaan ne ovat yhteydessä palvelimeen `CAlarmerClient`-luokan välityksellä, jonka `Alarm`-metodilla asiakas voi lähettää hälytysviestin palvelimelle. Saman rajapinnan `Log`-metodilla hälytys kirjautuu palvelimen lisäksi paikalliseen lokitiedostoon.

8.4 Testaussuunnitelma

Ohjelman testauksessa kiinnitettiin erityistä huomiota suorituskyky- ja luotettavuusvaatimuksiin. Testeillä haluttiin tarkistaa, että ohjelman tehokkuus ja virheensietokyky täyttävät asetetut vaatimukset. Testaus tapahtui kuvan 28 esittämässä ympäristössä neljän paikallisverkon avulla yhdistetyn työaseman kesken. Yksi ja sama työasema (A) toimi jokaisessa testissä palvelimena, jolla Alarmer Server -sovellus oli toiminnassa. Kolme muuta työasemaa (B, C ja D) toimivat hälytyksiä lähettävinä asiakaskoneina.

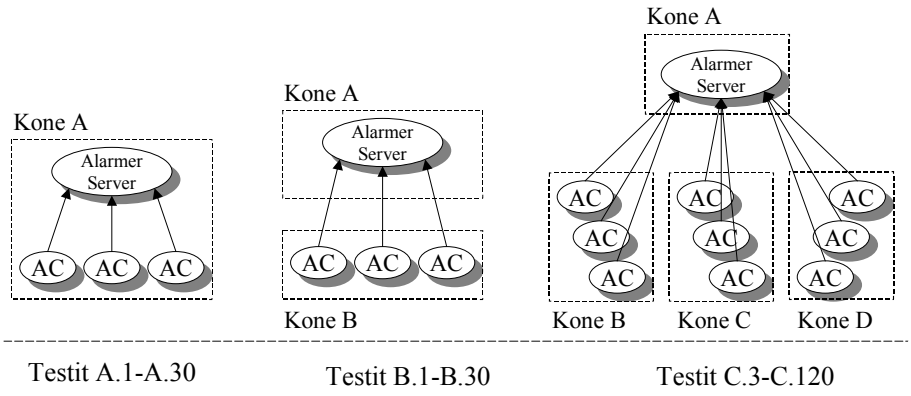


Kuva 28. Testausympäristö.

8.4.1 Suorituskykytestit

Suorituskyvyn testaamiseksi (vaatimukset luvussa 8.2.2) toteutettiin kolmentyyppisiä testejä, joiden koeasetelmat on esitetty kuvassa 29. Testeihin viitataan tästä eteenpäin kirjaimen ja numeron yhdistelmällä, jossa kirjain kertoo testityypin (A, B, C, D, E tai F) ja numero (1–120) testissä käytetyn asiakasprosessien kokonaismäärän. Suoritetut testit olivat seuraavat:

- Testeissä A.1–A.30 selvitettiin, kuinka monta hälytystä sekunnissa asiakassovellukset voivat välittää Alarmer Server -sovellukselle, kun Alarmer Server ja kaikki asiakassovellukset toimivat samalla koneella.
- Testeissä B.1–B.30 selvitettiin, kuinka monta hälytystä sekunnissa asiakassovellukset voivat välittää Alarmer Server -sovellukselle, kun Alarmer Server toimii yhdellä koneella ja kaikki asiakassovellukset toisella.
- Testeissä C.3–C.120 selvitettiin, kuinka monta hälytystä sekunnissa asiakassovellukset voivat välittää Alarmer Server -sovellukselle, kun Alarmer Server toimii yhdellä ja kaikki asiakassovellukset kolmella eri koneella.



Kuva 29. Suorituskykytestien koeasetelmat
(AC = Alarmer Client).

Testiä varten kehitettiin oma testaussovellus `gatest.exe`, joka Alarmer Client -moduulia käyttämällä lähetti käsketyn määrän hälytyksiä Alarmer Server -sovellukselle mahdollisimman nopeassa ajassa. Kuluneen ajan perusteella ohjelma laski ja tulosti keskimääräisen viestien lähetysnopeuden sekuntia kohden. Testien tulokset on käsitelty luvussa 9.3.1.

8.4.2 Luotettavuustestit

Sovelluksen virheensietokyvyn testaamiseksi väliohjelmiston toiminnan osalta käytettiin kolmea erilaista testiä palvelin- ja asiakaskoneen välillä:

- Testissä D.1 asiakassovellus oli välittämässä hälytyksiä Alarmer Server -sovellukselle, kun verkkoyhteys asiakkaaseen ja tietokantaan katkaistiin 60 sekunniksi.
- Testissä E.1 asiakassovellus oli välittämässä hälytyksiä Alarmer Server -sovellukselle, kun tietokantapalvelin sammutettiin ja käynnistettiin uudelleen.
- Testissä F.1 asiakassovellus oli välittämässä hälytyksiä Alarmer Server -sovellukselle, kun asiakassovellus keskeytettiin epänormaalisti ja käynnistettiin uudelleen.

Jokainen testi toistettiin viidesti ja katsottiin läpäistyksi, jos Alarmer Server -sovellus pystyi jatkamaan normaalia toimintaansa ilman uudelleenkäynnistystä virhetilanteen korjaamisen jälkeen, eikä sovellukselle saakka välitettyjä hälytyksiä jäänyt tallentamatta kantaan yhdelläkään testikerralla. Testien tuloksia käsitellään luvussa 9.3.2.

9 Generis Alarmerin toteutus ja testaus

Generis Alarmerin rakenteen ja vaatimusten määrittelyn jälkeen alkoi varsinaisen ohjelmiston toteutus. Toteutuksessa hyödynnettiin yritykselle tuttuja ja käytössä olevia työkaluja, jotka esitellään tarkemmin luvussa 9.1. Hajautuksen toteuttamisessa päädyttiin käyttämään DCOM-väliohjelmistoa, jonka valintaa luvussa myös perustellaan yksityiskohtaisemmin. Luvussa 9.2 selvitetään toteutettujen moduulien sisäistä rakennetta sekä toimintaa ja viimeisessä alaluvussa käsitellään ohjelmiston testituloksia.

9.1 Työkalut ja tekniikka

Tarjolla olevista väliohjelmistoista päätettiin käyttää Microsoftin DCOM-teknologiaa järjestelmän toteutukseen. Ensisijainen syy päätökseen oli **teknologian saatavuus**. Yrityksen ohjelmistot toimivat eri Windows-käyttöjärjestelmien alla ja täten DCOM oli automaattisesti käytettävissä kaikista sovelluksista. Lisäksi yrityksen käyttämät **kehitystyökalut tukivat hyvin DCOM-teknologiaa**. Siirtyminen uusien kehitystyökalujen käyttöön olisi vaatinut paljon aikaa ja investointeja, joita projektisuunnitelma ei sallinut. Kolmas valintaan vaikuttava seikka oli projektisuunnittelun loppuvaiheessa esiin tullut **tarve käyttää sovellusta Visual Basic -pohjaisista ohjelmointikielistä**. Koska Visual Basic ei kyseisellä hetkellä tukenut kuin COM-komponenttien käyttöä, DCOM alkoi vaikuttaa ainoalta varteenotettavalta vaihtoehdolta.

Toteutustyökaluksi valittiin Microsoft Visual C++ 6.0. Syynä valintaan oli aikaisempi kokemus kyseisellä työkalulla ohjelmoinnista sekä yrityksen hyvä tuntemus kyseisestä kääntäjästä. Visual C++ tukee hyvin DCOMin käyttöä ohjelmistokehityksessä sisältäen käyttöliittymässään automatisoidut komennot uusien DCOM-sovellusten luomiseen, luokkien lisäämiseen sekä niiden metodien ja ominaisuuksien käsittelyyn.

Varsinaisten DCOM-luokkien kehityksessä käytettiin apuna Microsoftin ATL 3.0 -luokkamallikirjastoa (*Active Template Library*), joka yksinkertaistaa ja helpottaa luokkien ja rajapintojen toteutusta. Kirjasto sisältää valmiit luokkamallit mm. `IUnknown`- ja `IDispatch`-rajapinnoille vapauttaen näin kehittäjän referenssilaskureiden ja

rajapintakyselymetodien toteuttamiselta. Kirjasto sisältää myös yleiskäyttöiset rutiinit tyyppikirjaston ja sen sisältämien rajapintojen rekisteröimiseksi järjestelmärekisteriin.

Tutkielmaa varten kehitetyssä CORBA-toteutuksessa käytettiin väliohjelmistona IONA-yhtiön CORBA 2.0 -standardin kanssa yhteensopivaa ORBacus 4.1.0 -kirjastoa. Kirjastosta oli saatavilla lähdekoodimuodossa ilmainen Unix- ja Windows-käyttöjärjestelmissä toimiva evaluointiversio.

9.2 Ohjelmiston toteutus

Järjestelmän toteutus jakaantui kolmeen eri moduuliin, joiden kehitys tapahtui lähes samanaikaisesti testauksen mahdollistamiseksi. Suurin työmäärä kului palvelinsovelluksen kehitykseen. Toteutettujen DCOM-luokkien rajapintojen IDL-kuvaukset on esitetty liitteessä 1.

9.2.1 Alarmer Server -palvelinsovellus

Viestejä keräävä Alarmer Server -sovellus toteutettiin **ulkoisena palvelimena** eli prosessina, joka toimii irrallaan asiakasprosesseista. Sovellus suunniteltiin siten, että sitä on mahdollista suorittaa sekä normaalisti ajettavana ohjelmana että taustalla Windows-palveluna. Jälkimmäinen optio antaa mahdollisuuden pitää sovellusta toiminnassa, vaikkei yksikään käyttäjä olisi kirjautunut sisään palvelimen käyttöjärjestelmään.

Alarmer Server -sovellus sisältää hajautettujen luokkien `CAlarmerServerSystem`, `CAlarmerServer` sekä `CAlarmerContext` toteutukset (katso kuva 27 sivulla 84). Jokainen luokka toteuttaa myös vastaavan nimisen rajapinnan. Luokkien toteutuksessa täytyi huomioida monisäiearkkitehtuurin asettamat vaatimukset ja suojata luokkien jäsenmuuttujia yhtäaikaiselta käytöltä. Suojauksessa hyödynnettiin käyttöjärjestelmän ohjelmointirajapinnan tarjoamia kriittisiä alueita (engl. *critical section)* sekä mutex-primitiivejä. Niiden avulla on mahdollista rajoittaa osia ohjelmakoodista siten, että vain yksi säie kerrallaan pystyy suorittamaan kyseistä koodinosaa.

Luotettavuudelle (katso luku 8.2.3) asetetut vaatimukset pyrittiin täyttämään kehittämällä sovellukseen levylle tallennettava väliaikaisvarasto, johon hälytysviestit säilötään tietokannan häiriöiden aikana. Tietokantaan tallennuksen epäonnistuessa jatkaa sovellus kantayhteyden muodostamisyrityksiä, kunnes sovellus sammutetaan tai yhteys saadaan uudelleenmuodostettua.

Sovelluksen suorituskyvyn (katso luku 8.2.2) parantamiseksi päädyttiin ratkaisuun, jossa verkon yli ei välitetä hälytysviestejä kokonaisuudessaan, vaan ainoastaan hälytyksen tunnusnumero. Tunnusnumeron perusteella voidaan tietokannasta hakea todellinen hälytysviesti, joka on sinne aiemmin rekisteröity tarkoitusta varten suunnitellulla ohjelmalla. Tunnusnumeron lisäksi on hälytysviestissä mahdollista välittää vaihteleva määrä merkkijonomuotoisia parametreja, jotka sijoitetaan lopulliseen hälytysviestiin niille varatuille paikoille ennen hälytyksen loppukäyttäjälle raportointia.

Palvelinsovelluksen toiminta on pääpiirteissään seuraavanlainen. Aluksi sovellus alustaa COM-järjestelmän sekä joitakin COMin suojausasetuksia, jotka sallivat ulkopuolisten työasemien ottaa yhteyttä prosessiin. Seuraavaksi palvelinsovellus rekisteröi luokkatehtaansa (katso luku 6.3.1) COM-järjestelmän ROT-tauluun (*Running Object Table*). ROT-taulun kautta asiakassovellukset pääsevät käsiksi palvelimen luokkatehtaaseen, jonka avulla tapahtuu palvelimen toteuttamien muiden luokkien luonti. Rekisteröinnin jälkeen palvelinsovellus jää viestisilmukkaan käsittelemään asiakassovelluksilta tulevia pyyntöjä, kunnes sovellus sammutetaan käyttäjän toimesta. Alasajon loppuvaiheessa palvelinsovellus poistaa luokkatehtaansa ROT-taulusta.

9.2.2 Alarmer Client -moduuli

Asiakassovelluksien käyttämä Alarmer Client -moduuli toteutettiin **asiakasprosessiin liitettävänä palvelimena**. Moduuli sisältää `CAlarmerClient`-luokan toteutuksen sekä standardikutsutapaa noudattavan C-kielisen funktiorajapinnan, jonka kautta DCOM-tekniikkaa tukemattomatkin sovellukset voivat käyttää moduulia.

Asiakassovelluksen lähettäessä ensimmäistä kertaa hälytyksen `CAlarmerClient`-luokan `Alarm`-metodilla, luo Alarmer Client -moduuli uuden instanssin Alarmer Server -palvelimella sijaitsevasta `CAlarmerServer`-luokasta ja rekisteröi itsensä `RegisterClient`-metodia käyttäen. Rekisteröinnin tuloksena moduuli saa rajapintaosoittimen uuteen `CAlarmerContext`-luokkaan, jonka `PostMessage`-metodilla moduuli lähettää hälytysviestin Alarmer Server -palvelimelle. Alarmer Client -moduuli pitää rajapintaosoitinta tallessa uusien hälytyksien varalta ja vapauttaa sen vasta asiakassovelluksen sammutuksen yhteydessä.

9.2.3 Alarmer Server Proxy -moduuli

Jottei hälytyksiä lähettäville asiakaskoneille olisi tarpeen asentaa Alarmer Server -sovellusta, generoitiin erillinen **asiakasprosessiin liitettävä moduuli** Alarmer Server Proxy, joka sisältää Alarmer Server -sovelluksen rajapintojen sarjallistamisrutiinit.

Moduuli on mahdollista generoida automaattisesti Alarmer Server -sovelluksen käännöksen yhteydessä ja se täytyy rekisteröidä asiakaskoneella, kuten Alarmer Client -moduulikin. Rekisteröinnin yhteydessä moduuli kirjoittaa järjestelmärekisteriin rajapintojen `IAlarmerServer`, `IAlarmerServerSystem` ja `IAlarmerContext` IID-tunnukset sekä rekisteröi itsensä näiden rajapintojen sarjallistamiskomponentit toteuttavaksi moduuliksi.

9.3 Ohjelmiston toteutuksen testitulokset

Generis Alarmer -järjestelmän valmistuttua suoritettiin sille testaussuunnitelman (katso luku 8.4) mukaiset testit. Tuloksien analysoinnissa kiinnitettiin erityisesti huomiota siihen, kuinka paikallis- ja etäkutsut eroavat suorituskyvyltään toisistaan sekä millä tavalla asiakassovellusten hajauttaminen usealle työasemalle vaikuttaa hälytysten välitysnopeuteen.

Testeissä A.1–A.30 käytettiin yhtä työasemaa, testeissä B.1–B.30 kahta ja testeissä C.1–C.120 yhteensä neljää työasemaa. Kaikki luotettavuustestit toteutettiin kahta työasemaa käyttämällä. Riippumatta testitapauksesta yksi työasema toimi aina palvelimena. Käytetty testauslaitteisto on esitelty yksityiskohtaisemmin liitteessä 5.

Ohjelmiston toimivuutta eri käyttöympäristöissä ja turvallisuusvaatimuksien täyttymistä ei erikseen testattu tutkielman teon aikana. Samoin testiympäristö ei tukenut testausta klusterilaitteistolla. Suorituskykytestien aikana ohjelmiston havaittiin kuitenkin toimivan sekä Windows 2000 että Windows XP -käyttöjärjestelmissä. Ongelmia kuitenkin esiintyi, jos palvelinsovellus sijaitsi vanhemman käyttöjärjestelmän (Windows 2000) alaisuudessa kuin asiakassovellukset (Windows XP). Tällöin sovellusten välille ei saatu muodostettua toimivaa yhteyttä kohtuullisella puolen päivän asennustyöllä.

9.3.1 Suorituskyky

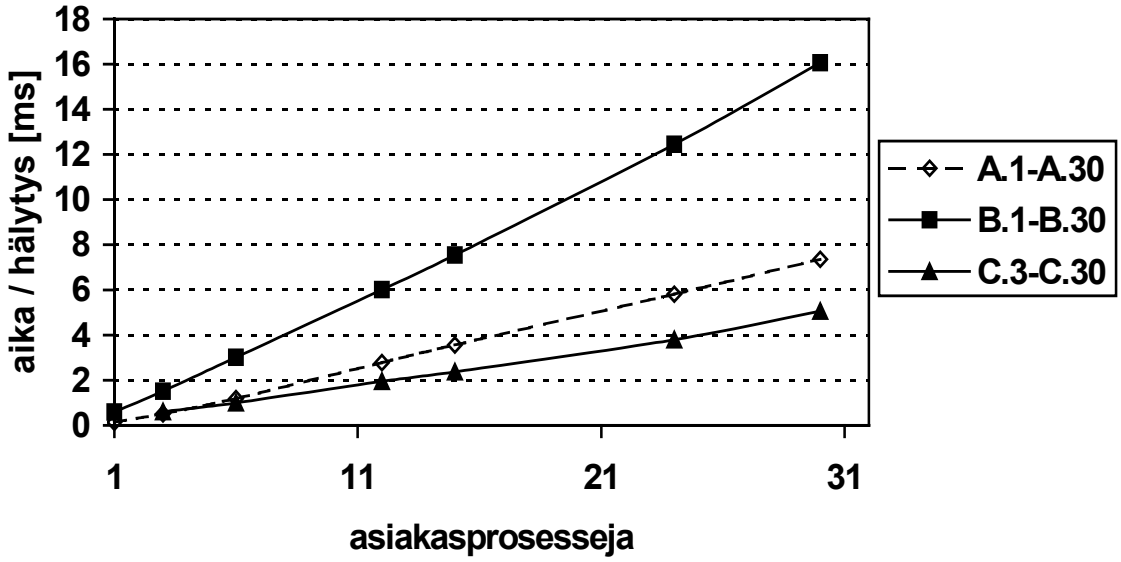
Suorituskykytestaus suoritettiin varsinaisen työajan ulkopuolella, jolloin muuta verkkoliikennettä oli mahdollisimman vähän. Testaus tapahtui luvussa 8.4.1 kuvatun suunnitelman mukaan. Jokaisessa testitapauksessa mitattiin aika, joka asiakasprosesseilta kului yhteensä 120 000 hälytysviestin välittämiseen palvelimelle. Tämän ajan perusteella voitiin laskea taulukon 9 tulokset.

Taulukon 9 ensimmäinen arvo kertoo ajan, joka keskimäärin kului yhden hälytysviestin välitykseen asiakassovellukselta palvelimelle. Arvo on muodostettu jakamalla koko testiin kulunut aika välitettyjen viestien määrällä. Jälkimmäinen suluissa oleva kokonaislähetysnopeus kertoo, kuinka monta viestiä prosessit pystyivät yhteensä keskimäärin välittämään palvelimelle sekunnissa.

Asiakas-prosesseja (n)	**Testityyppi**		
	A.n	**B.n**	**C.n**
1	0,14 ms (7356)	0,60 ms (1660)	-
3	0,51 ms (5921)	1,52 ms (1975)	0,61 ms (4958)
6	1,19 ms (5026)	3,02 ms (1989)	1,00 ms (5972)
12	2,78 ms (4322)	6,02 ms (1993)	1,95 ms (6158)
15	3,56 ms (4215)	7,54 ms (1989)	2,37 ms (6327)
24	5,82 ms (4122)	12,45 ms (1928)	3,80 ms (6322)
30	7,36 ms (4076)	16,07 ms (1867)	5,06 ms (5931)
60	-	-	10,36 ms (5791)
75	-	-	14,44 ms (5195)
120	-	-	24,14 ms (4970)

Taulukko 9. Generis Alarmerin suorituskykytestin tulokset.

Saatujen tuloksien ja kuvan 30 pohjalta voidaan aluksi havaita suuri suorituskykyero paikallis- ja etäkutsujen välillä (testit A.1–A.6 ja B.1–B.6), kun asiakasprosesseja on vähän toiminnassa ja niitä suoritetaan yhdellä työasemalla. Tällöin etäkutsut ovat yli neljä kertaa (343%) paikalliskutsuja hitaampia. Asiakasprosessien määrän kasvaessa ero alkaa kutistua, kunnes prosessien määrän ollessa 30 (testit A.30 ja B.30) ero on noin kaksinkertainen (118%). Tämä trendi on selitettävissä siten, että samalla työasemalla suoritettavien prosessien lukumäärän kasvaessa työaseman yleinen suorituskyky nousee verkkoliikenteen nopeutta tärkeämmäksi tekijäksi ja alkaa rajoittaa hälytysviestien lähetysnopeutta. Testien A.1–A.30 tuloksia tarkastellessa täytyy myös huomioida palvelimena käytetyn testikoneen A suurempi tehokkuus (2.5 GHz) verrattuna muihin testikoneisiin (katso liite 5).



Kuva 30. Generis Alarmerin suorituskykytestien tuloksia
eri asiakasprosessimäärillä.

Tutkittaessa testien C.3–C.120 tuloksia nähdään, että asiakasprosessien hajauttaminen useammalle työasemalle auttaa nostamaan hälytysten välitysnopeutta selvästi, vaikka prosesseja olisikin sama määrä kuin yhden työaseman tapauksessa. Esimerkiksi verrattaessa testiä B.12, jossa yhdellä työasemalla oli toiminnassa 12 asiakasprosessia, testiin C.12, jossa kolmella eri työasemalla oli kullakin toiminnassa 4 asiakasprosessia,

pystyivät prosessit jälkimmäisessä tapauksessa välittämään viestejä yli kolme kertaa (209%) tehokkaammin.

Yhteenvetona tuloksista voidaan sanoa, että asiakasprosessien hajauttaminen usealle eri työasemalle vaikuttaisi olevan kannattavaa jo kolmen asiakasprosessin tapauksessa. Pienemmillä määrillä verkkoliikenteeseen kuluva aika on suhteettoman suuri verrattuna itse asiakasprosessin suoritukseen kuluvaan aikaan. Testien perusteella huonoin vaihtoehto on sijoittaa palvelinprosessi ja kaikki asiakasprosessit kahdelle eri työasemalle. Tällöin verkkoliikenteeseen kuluva aika dominoi suoritusta ja hidastaa asiakasprosessien toimintaa.

Tuloksia arvioidessa täytyy ottaa huomioon kolmen eri tekijän vaikutus testeihin: asiakaskoneen suorituskyky, palvelinkoneen suorituskyky sekä verkon siirtonopeus. Näiden kunkin osuutta tuloksista on mahdotonta erotella tarkasti ja tästä johtuen testien perusteella ei voida tehdä suoria johtopäätöksiä asiakas- tai palvelinsovelluksen suorituskyvystä. Testin C.15 pohjalta voidaan kuitenkin päätellä, että palvelin kykenee käsittelemään ainakin 6327 viestiä sekunnissa. Maksimaalisen suorituskyvyn selville saamiseksi olisi tarvittu enemmän työasemia asiakaskoneiksi, mutta tähän ei testiympäristö antanut mahdollisuutta.

Suorituskykytestien pohjalta voidaan todeta **DCOMin täyttävän Generis Alarmer -järjestelmälle asetetut suorituskykyvaatimukset** ja osittain ylittävänkin ne. Palvelin kykenee palvelemaan 100 asiakassovellusta, kuten testi C.120 osoittaa. Jokainen asiakassovellus pystyy tällöin välittämään vähintään 10 hälytysviestiä sekunnissa palvelimelle, kunhan kaikki asiakassovellukset eivät toimi samalla työasemalla. Näiden tuloksien valossa Generis Alarmer pystyy skaalautumaan laajoihinkin ympäristöihin, jos asiakassovelluksia hajautetaan riittävästi.

9.3.2 Virheiden käsittely

Virhetilanteiden käsittelytestit (katso luku 8.4.2) suoritettiin työaikana verkon ollessa lievästi kuormitettuna. Testien tulokset on esitetty taulukossa 10.

Testikerta	**D.1**	**E.1**	**F.1**
1	OK	OK	OK
2	OK	OK	OK
3	OK	OK	OK
4	OK	OK	OK
5	OK	OK	OK
Koko testi	Hyväksytty	Hyväksytty	Hyväksytty

Taulukko 10. Generis Alarmerin virheensietokykytestin tulokset.

Tuloksista **voi päätellä Alarmer Server -palvelinohjelman kestävän vaaditulla tavalla erilaisia verkkoyhteyteen liittyviä virhetilanteita**. Testissä D.1 palvelinohjelman verkkoyhteys katkaistiin väliaikaisesti 60 sekunniksi. Tämä aika ei ollut riittävä DCOMin automaattisen roskienkeruun aktivointiin palvelinsovelluksen puolella, ja siksi Alarmer Server ei katkokseen reagoinut. Asiakasohjelma havaitsi virheen noin 20 sekunnin kuluttua katkoksesta, ja tästä eteenpäin jokainen uusi lähetysyritys epäonnistui 15-30 sekunnin viiveellä, kunnes verkkoyhteys palasi toimintaan. Verkkoyhteyden palautumisen jälkeen viestit välittyivät normaalisti palvelimelle.

Testissä E.1 tietokantapalvelinta käynnistettiin ja sammutettiin toistuvasti. Tällöin Alarmer Server havaitsi testin D.1 lailla tietokantayhteyden katkenneen ja tallensi katkon aikana saapuvat hälytykset levylle väliaikaisvarastoon. Tietokantayhteyden palatessa hälytysviestit siirrettiin tietokantaan.

Testissä F.1 asiakassovellus sammutettiin epänormaalisti[5]. Tällöin tuli esiin DCOMin roskienkeruun toimivuus virhetilanteissa. Epänormaalin sammutuksen vuoksi asiakassovellus ei voinut vapauttaa etäreferenssiään palvelimen `IAlarmerContext`-rajapintaan. Kuitenkin tarkkailemalla Alarmer Serverin lokitiedostoa, voidaan nähdä etäreferenssien vapautuvan noin 10 minuutin päästä ensimmäisen asiakassovelluksen sammumisesta. Etäreferenssin vapauttamisen hoitaa tällöin DCOMin automaattinen vapautusjärjestelmä.

[5] Prosessin äkilliseen ”tappamiseen” käytettiin ilmaiseksi tarjottavaa PsKill-ohjelmaa, joka on saatavilla URL-osoitteesta `http://www.sysinternals.com/ntw2k/freeware/pskill.shtml`.

10 Väliohjelmiston suorituskyvyn testaus

Generis Alarmerilla tehtäviin suorituskykytesteihin vaikutti väliohjelmiston suorituskyvyn lisäksi myös palvelinsovelluksen ja asiakassovellusten suorituskyky, joten Generis Alarmerilla suoritettujen testien perusteella ei voinut vetää johtopäätöksiä väliohjelmiston tehokkuudesta. Tästä syystä Alarmer Serveristä toteutettiin lisäksi yksinkertaistettu versio eli **DCOM-testisovellus**, jonka toteutus on selvitetty luvussa 10.1.1.

Jotta DCOM-testisovelluksen suorituskykytesteille olisi ollut vertailuaineistoa, toteutettiin lisäksi sitä vastaava ohjelmisto käyttämällä CORBA-arkkitehtuuriin perustuvaa ORBacus-väliohjelmistoa. Ohjelmiston kehittämisen tarkoituksena oli vertailuaineiston saamisen lisäksi tutustua CORBAn soveltamiseen käytännön ohjelmistokehityksessä. Toteutettuun ohjelmistoon viitataan jatkossa nimellä **CORBA-testisovellus,** ja sen toteutus on kuvattu luvussa 10.2.

10.1 DCOM-testisovellus

DCOM-testisovelluksen toteuttaminen oli yksinkertainen tehtävä Generis Alarmerin toteutuksen jälkeen. Luvussa kuvataan tarkemmin, mitä osia Alarmer Server -palvelinsovelluksesta poistettiin testisovellusta varten, sekä millaisia tuloksia testisovellukselle tehdyt suorituskykytestit antoivat.

10.1.1 Toteutus

Väliohjelmiston suorituskyvyn testauksessa käytetty DCOM-testisovellus toteutettiin poistamalla valmiista Alarmer Server -palvelinsovelluksesta (katso luku 9.2.1) kaikki saapuvien viestien käsittelyyn liittyvä toimintalogiikka, joka voisi hidastaa palvelinsovelluksen toimintaa. Näihin toimintoihin kuului mm. viestien tallennus tietokantaan sekä väliaikaisvaraston käyttö tietokantatallennuksen epäonnistuessa. Muutosten tavoitteena oli saada palvelinsovelluksen suorituskyky riippumaan mahdollisimman suuresti ainoastaan DCOM-väliohjelmiston suorituskyvystä.

Generis Alarmerin testauksessa käytettyä asiakassovellusta `gatest.exe` muokattiin testausta varten siten, että se käytti suoraan Alarmer Serverin `IAlarmerServer`- ja `IAlarmerContext`-rajapintoja ohittaen Alarmer Client -moduulin (katso luku 9.2.2). Tällöin asiakassovelluksen ja palvelinsovelluksen väliin ei jäänyt ylimääräisiä sovelluskerroksia, jotka voisivat hidastaa asiakassovelluksen tehokkuutta.

10.1.2 Suorituskyky

DCOM-testisovellukselle suoritettiin samat luvussa 8.4.1 kuvatut suorituskykytestit kuin valmiille Generis Alarmer -sovellukselle. Testitulokset on kirjattu taulukkoon 11. Taulukon ensimmäinen arvo kertoo millisekunneissa ajan, joka keskimäärin kului yhden hälytysviestin välitykseen palvelimelle. Suluissa oleva kokonaislähetysnopeus kertoo, kuinka monta viestiä prosessit pystyivät yhteensä keskimäärin välittämään palvelimelle sekunnissa

Testien A.1–A.30 tuloksista voi nähdä DCOMin olevan erittäin tehokas välittämään viestejä kahden samalla työasemalla sijaitsevan prosessin välillä. DCOM pystyy tällöin välittämään yli 30 000 viestiä sekunnissa, kunhan palvelinprosessin sisäinen toiminta ei hidasta viestinvälitystä. Kommunikointi kahden työaseman välillä on testien B.1–B.30 perusteella selvästi hitaampaa verrattuna paikallisiin kutsuihin. Yhden asiakasprosessin kohdalla nopeusero on yli kymmenkertainen (964%), mutta ero alkaa hiljalleen pienetä, kunnes kolmenkymmenen prosessin tapauksessa paikalliskutsut ovat enää 256% etäkutsuja nopeampia

Asiakas-prosesseja (n)	**Testityyppi**		
	A.n	**B.n**	**C.n**
1	0,03 ms (32680)	0,33 ms (3073)	-
3	0,10 ms (31226)	0,58 ms (5178)	0,35 ms (8576)
6	0,20 ms (29426)	1,15 ms (5230)	0,47 ms (12712)
12	0,52 ms (23064)	2,50 ms (4791)	0,87 ms (13823)
15	0,74 ms (20318)	3,06 ms (4901)	1,10 ms (13576)
24	1,46 ms (16445)	5,24 ms (4584)	1,78 ms (13515)
30	1,85 ms (16203)	6,58 ms (4560)	2,28 ms (13166)
60	-	-	5,10 ms (11754)
75	-	-	6,62 ms (11326)
120	-	-	12,42 ms (9660)

Taulukko 11. DCOM-testisovelluksen suorituskykytestien tulokset.

Tarkasteltaessa testien C.1–C.30 tuloksia nähdään hajautuksen usealle työasemalle selvästi kannattavan jo kolmen asiakasprosessin tapauksessa verrattuna testien B.1–B.30 tuloksiin. Nopeusero alkaa entisestään korostua asiakasprosessien määrän noustessa ja kolmenkymmenen prosessin kohdalla ero on jo lähes kolminkertainen (189%).

10.2 CORBA-testisovelluksen toteutus

DCOM-testisovellusta vastaava CORBA-testisovellus jakaantui palvelin- ja asiakassovellukseen. Palvelinsovellus vastasi DCOM-testisovellusta ja asiakassovellus testeissä käytettyä `gatest.exe` ohjelmaa. Asiakassovellus sisälsi lisäksi testausrutiinit, joiden avulla suorituskykytestit toteutettiin.

Palvelinsovelluksen yksinkertaistamiseksi jätettiin toteuttamatta testeissä tarpeeton `CAlarmerServerSystem`-luokka. Myöskään `CAlarmerClient`-luokkaa ei toteutettu, vaan asiakassovellus käytti suoraan `CAlarmerContext`-luokkaa hälytysten lähetyksessä. Lisäksi tietokantatuki jätettiin toteuttamatta. Toteutus vastasi mahdollisimman tarkasti luvussa 10.1 DCOM-testisovellusta toiminnoiltaan. Järjestelmän IDL-rajapintakuvaus on esitetty liitteessä 2.

10.2.1 Palvelinsovellus

Palvelinsovellus toteutettiin omana EXE-tyyppisenä sovelluksena ja se sisältää `CAlarmerServer`-luokan toteutuksen. Palvelinsovelluksen toiminta on yksinkertaista. Sovellus alustaa ensimmäiseksi CORBA-järjestelmän ja kysyy siltä referenssiä oliosovittimeen (katso luku 5.1.2). Seuraavaksi palvelin luo uuden ilmentymän `CAlarmerServer`-luokasta ja muuntaa sen referenssin IOR-muotoiseksi (katso luku 6.1.2) merkkijonoksi.

Tämän jälkeen palvelin kirjoittaa merkkijonomuotoisen referenssin jaettuun hakemistoon vakionimiseen tiedostoon asiakassovelluksen saataville. Lopuksi palvelinsovellus aktivoi oliosovittimen aiemmin saadun referenssin kautta ja jää odottamaan viestejä asiakassovelluksilta. Palvelinsovellus ei sammuta itseään automaattisesti, vaan tämä on tehtävä käyttäjän toimesta.

10.2.2 Asiakassovellus

Asiakassovellus toteutettiin palvelinsovelluksen lailla itsenäisesti suoritettavana sovelluksena. Ensimmäiseksi asiakassovellus alustaa CORBA-järjestelmän. Seuraavaksi sovellus käy hakemassa palvelinsovelluksen tiedostoon kirjoittaman IOR-referenssin ja muuntaa tämän todelliseksi `CAlarmerServer`-olion etäreferenssiksi.

Tämän jälkeen sovellus rekisteröi itsensä palvelinsovellukselle `RegisterClient`-metodilla ja saa kutsun paluuarvona etäreferenssin `CAlarmerContext`-olioon. Lopuksi asiakassovellus suorittaa testaussuunnitelmassa (katso luku 8.4.1) kuvatut suorituskykytestit lähettämällä viestit `CAlarmerContext`-olion `PostMessage`-metodin avulla.

10.3 CORBA-testisovelluksen testitulokset

Valmiilla CORBA-testisovelluksella toteutettiin samat luvun 8.4 testaussuunnitelmassa kuvatut testit kuin Generis Alarmer -järjestelmälle lukuun ottamatta testiä E.1. Testi E.1 mittaa tietokannan virheensietokykyä verkkokatkoksissa ja se jätettiin suorittamatta, koska CORBA-testisovellukseen ei toteutettu tietokantatallennusta. Muiden testien osalta testaus suoritettiin samassa testausympäristössä kuin DCOM-testisovelluksen testaus. Testausympäristön tarkempi kuvaus on liitteessä 5.

10.3.1 Suorituskyky

CORBA-testisovelluksen suorituskykytestaus suoritettiin työajan ulkopuolelle verkon kuormituksen ollessa pientä. Jokaisessa testitapauksessa mitattiin aika, joka asiakasprosesseilta kului yhteensä 120 000 hälytysviestin välittämiseen palvelimelle. Testien tuloksena saatiin hälytyksen välitykseen käytetty aika sekä kokonaislähetysnopeus. Testien tulokset on esitetty taulukossa 12. Taulukon ensimmäinen arvo kertoo millisekunneissa ajan, joka keskimäärin kului yhden hälytysviestin välitykseen palvelimelle. Suluissa oleva kokonaislähetysnopeus kertoo, kuinka monta viestiä keskimäärin prosessit pystyivät yhteensä välittämään palvelimelle sekunnissa

Asiakkaita yhteensä (n)	**Testityyppi**		
	A.n	**B.n**	**C.n**
1	0,14 ms (7357)	0,39 ms (2563)	-
3	0,42 ms (7111)	0,95 ms (3171)	0,40 ms (7536)
6	0,85 ms (7020)	1,84 ms (3266)	0,66 ms (9074)
12	1,78 ms (6761)	3,71 ms (3237)	1,25 ms (9582)
15	2,25 ms (6678)	4,64 ms (3230)	1,57 ms (9557)
24	3,63 ms (6604)	7,38 ms (3251)	2,54 ms (9442)
30	4,62 ms (6497)	9,43 ms (3183)	3,22 ms (9306)
60	-	-	6,43 ms (9332)
75	-	-	8,10 ms (9260)
120	-	-	14,00 ms (8405)

Taulukko 12. CORBA-testisovelluksen suorituskykytestin tulokset.

Taulukon 12 arvojen perusteella testitulosten trendi asiakasprosessien määrän muuttuessa on samansuuntainen kuin DCOM-testisovelluksen. Kahdelle työasemalle hajautettu järjestelmä, jossa palvelinsovellus sijaitsee toisella ja kaikki asiakasprosessit toisella, vaikuttaa olevan huonoin vaihtoehto myös ORBacus-väliohjelmistoa käytettäessä. Asiakasprosessien hajauttaminen kolmelle eri työasemalle yhden sijaan kannattaa kuitenkin DCOMia selvemmin jo kolmen asiakasprosessin (testi C.3) tapauksessa. CORBA-palvelimen maksimaalinen käsittelynopeus on tulosten perusteella vähintään 9584 viestiä sekunnissa, kuten testi C.12 osoittaa. DCOM- ja CORBA-testisovellusten suorituskykytestien tuloksia vertaillaan yksityiskohtaisemmin toisiinsa luvussa 11.1.

10.3.2 Virheiden käsittely

Koska CORBA-testisovellukseen ei sisällytetty tietokantatukea, oli sille järkevää suorittaa luvun 8.4.2 luotettavuustesteistä vain D.1 ja F.1. Näiden testien tulokset on kuvattu taulukossa 13.

Testikerta	**D.1**	**F.1**
1	OK	OK
2	OK	OK
3	OK	OK
4	OK	OK
5	OK	OK
Koko testi	Hyväksytty	Hyväksytty

Taulukko 13. CORBA-testisovelluksen virheensietokykytestin tulokset.

Testissä D.1 verkkoyhteyden katkeaminen aiheutti asiakasohjelmassa poikkeuksen noin seitsemän sekunnin sisällä hälytysviestin lähetysyrityksestä. Tämän jälkeen poikkeus tapahtui välittömästi hälytyksen lähetyksen yhteydessä, kunnes verkkoyhteys palasi

toimintaan. Tällöin asiakassovellus toipui hyvin ja viestien lähetys jatkui onnistuneesti. Palvelinsovellus ei reagoinut millään tavalla verkkoyhteyden katkeamiseen.

Testin F.1 mielekkyys oli kyseenalainen, sillä CORBA ei sisällä automaattista roskienkeruuta, eikä palvelinsovellus näin ollen reagoinut millään tavalla asiakasohjelman epänormaaliin sammutukseen. Testin aikana palvelinsovellus toimi tästä johtuen ennalta odotetusti moitteettomasti.

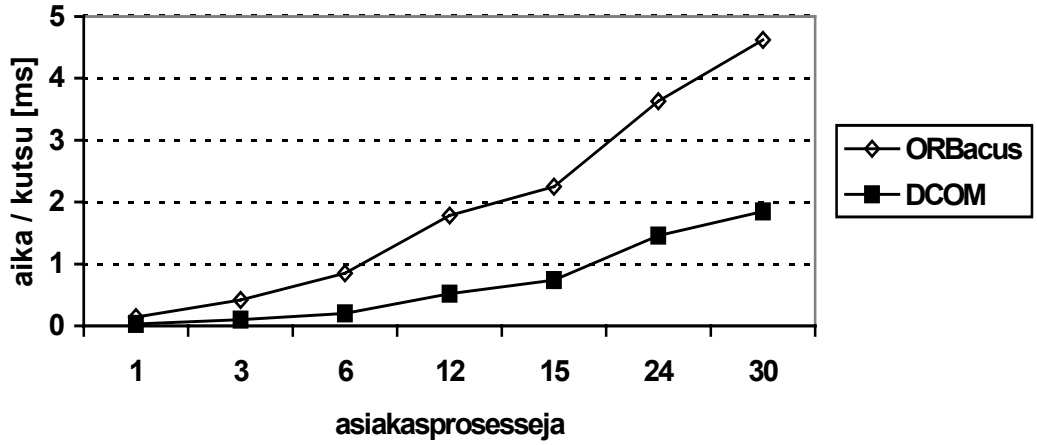
11 Testitulosten analyysi ja johtopäätöksiä

Luvussa verrataan DCOM- ja CORBA-testisovelluksille suoritettujen suorituskyky- ja virheenkäsittelytestien tuloksia keskenään. Lisäksi luvussa tehdään yhteenkokoavaa vertailua DCOM- ja CORBA-väliohjelmistojen teknisistä ominaisuuksista sekä tarkastellaan käytännön toteutuksien kehitystyössä esiin tulleita eroja. Tehtyjen suorituskykytestien tulokset ovat nähtävissä kokonaisuudessaan liitteessä 6.

Generis Alarmer -ohjelmistolle suoritetut testit eivät ole vertailukelpoisia DCOM- ja CORBA-testisovelluksien testien kanssa, sillä sovellukset poikkeavat liian suuresti toisistaan sekä asiakas- että palvelinohjelman toteutuksen osalta.

11.1 Suorituskyky ja virheenkäsittely

Verrattaessa lukujen 10.1.2 ja 10.3.1 tuloksia toisiinsa, havaitaan **DCOM-väliohjelmiston olevan tehokkaampi kuin käytetty CORBA-väliohjelmisto ORBacus**. Erot ovat suurimmat tehtäessä paikalliskutsuja kahden samalla työasemalla sijaitsevan prosessin välillä (katso kuva 31). Hajautettaessa asiakasprosesseja laajemmin useammalle työasemalle erot alkavat pienetä (katso kuvat 32 ja 33).



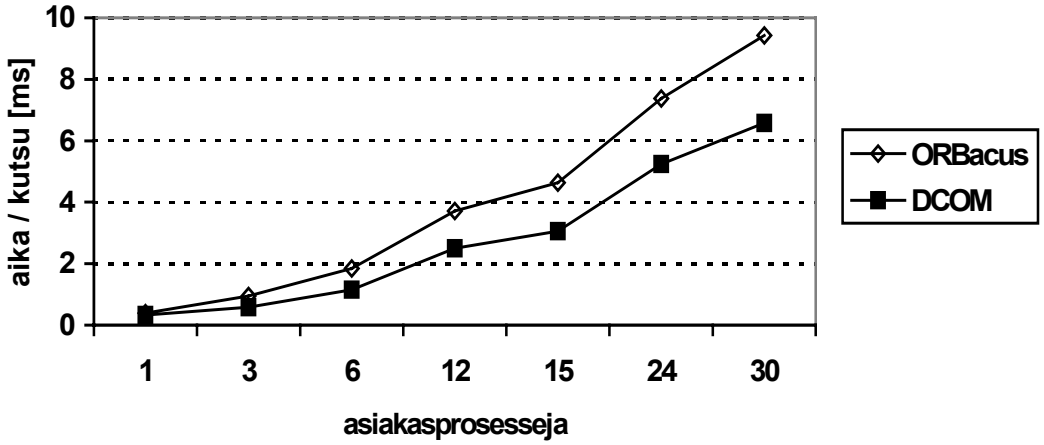
Kuva 31. Paikalliskutsujen nopeudet (testit A.1–A.30).

Kuvassa 31 on esitetty graafisesti testien A.1–A.30 tulokset DCOM- ja CORBA-testisovelluksille (katso taulukot 11 ja 12). Yhden asiakasprosessin tapauksessa DCOM on yli neljä kertaa (344%) ORBacusta nopeampi. Asiakasprosessien määrän kasvaessa

tehoerot alkavat pienetä ja kolmenkymmenen prosessin tapauksessa nopeusero DCOMin eduksi on enää 149%. Tehoerojen pienentyminen asiakasprosessien määrän kasvaessa johtunee siitä, että käyttöjärjestelmän prosessien vuorotteluun kuluva aika alkaa rajoittaa prosessien suoritusnopeutta enemmän kuin väliohjelmiston tehokkuus.

Pääsyynä DCOM-väliohjelmiston selvästi parempiin testituloksiin paikalliskutsuja tehtäessä on todennäköisesti DCOMin paremmin optimoidut sarjallistamisrutiinit (katso luku 7.2). Tutkimalla samalla työasemalla sijaitsevien asiakas- ja sovellusprosessien varaamia käyttöjärjestelmäresursseja, voi havaita DCOM-väliohjelmiston käyttävän jaettua muistia prosessien välisessä kommunikoinnissa. Jaetun muistin käyttö RPC-kanavan (katso luku 7.2.3) korvaajana paikalliskutsuja välitettäessä nopeuttaa toimintaa suuresti. ORBacus-väliohjelmisto ei jaettua muistia vaikuta hyödyntävän. Syynä tähän voi olla ORBacuksen toteutustapa, joka sallii väliohjelmistoa käytettävän myös muissa kuin Windows-käyttöjärjestelmissä.

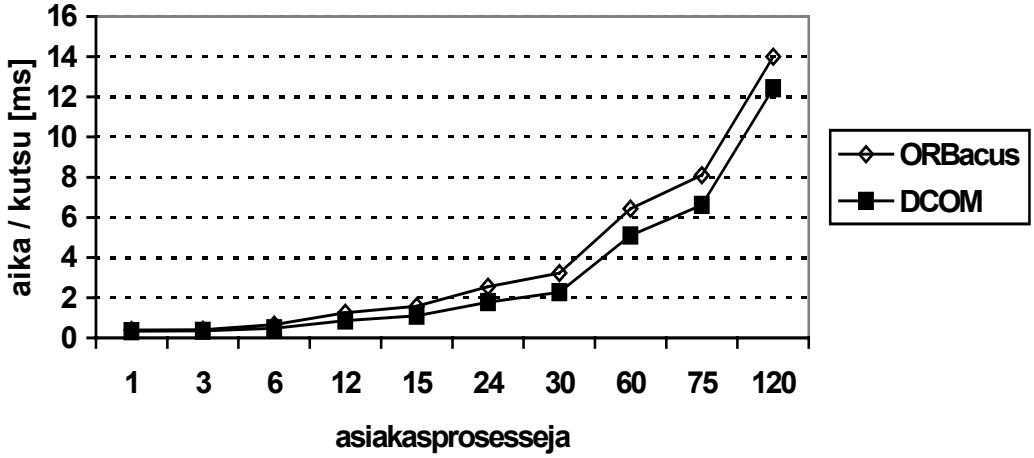
Hajautettaessa asiakasprosessit eri työasemalle kuin millä palvelinsovellus sijaitsee, eivät DCOMin ja ORBacuksen tehoerot ole yhtä suuret kuin testien A.1–A.30 kohdalla. Kuvassa 32 on kuvaajamuodossa testien B.1–B.30 tulokset.



Kuva 32. Etäkutsujen nopeudet yhdeltä asiakaskoneelta (testit B.1–B.30).

Tuloksista havaitaan DCOMin olevan hieman yli 30% ORBacusta nopeampi, kun hajautus on toteutettu kahden koneen kesken. Tehoero on tätäkin pienempi (17%), kun

asiakasprosesseja on vain yksi (testi B.1). Hajautettaessa asiakasprosessit kolmelle eri työasemalle pienenevät tehoerot DCOMin ja ORBacuksen välillä entisestään. Kuvassa 33 on esitetty graafisesti testien C.1–C.120 tulokset.



Kuva 33. Etäkutsujen nopeudet kolmelta asiakaskoneelta (testit C.3–C.120).

Testien C.1–C.120 tuloksista voi laskea DCOMin olevan hieman alle 30% ORBacusta nopeampi, kun asiakasprosessit on hajautettu kolmelle työasemalle. Ero on pienimmillään testissä C.120, jolloin DCOM on vain 11% tehokkaampi. Näin pienet erot eivät ole enää käytännön toteutuksissa merkittäviä.

Tarkasteltaessa yksittäisen viestin välitysnopeuden sijaan liitteen 6 taulukosta 15 löytyviä palvelimen suorituskykytuloksia, voi testeissä B.1–B.30 ja C.1–C.120 havaita mielenkiintoisen seikan. Molemmissa testeissä on tietty testitapaus, jolloin palvelimen suorituskyky saavuttaa maksiminsa ja kääntyy tämän jälkeen laskuun. Kyseinen testitapaus on sama sekä DCOM- että ORBacus-väliohjelmistoilla. Testeissä B.1–B.30 tämä kohta on kuuden asiakasprosessin kohdalla (testi B.6), kun taas testeissä C.1–C.120 suurin suorituskyky saavutetaan 12 asiakasprosessilla (testi C.12). Koska palvelimelle saapuvien viestien kokonaismäärä sekuntia kohden riippuu kokonaisuudessaan asiakasohjelman, verkon ja palvelinohjelman suorituskyvystä, on vaikeaa arvioida, mikä näistä nousee rajoittavaksi tekijäksi kyseisissä kohdissa. Asian selvittäminen olisi vaatinut laajempien testien tekemistä.

Yhteenvetona testisovelluksien suorituskykytesteistä voi todeta DCOMin ja CORBA-arkkitehtuuria käyttävän ORBacuksen olevan suorituskyvyiltään samaa luokkaa, kun asiakasprosessit toimivat eri työasemilla kuin palvelinprosessi. Pienet tehoerot väliohjelmistojen välillä kutistuvat entisestään, kun asiakasprosessien määrä kasvaa. Jos asiakasprosessit toimivat kuitenkin samalla työasemalla kuin palvelinprosessi, on DCOM tällöin lähes kertaluokkaa nopeampi kuin ORBacus. Tulos kannattaa huomioida väliohjelmiston valinnassa, jos etukäteen voidaan tietää, kuinka järjestelmän prosessit tulevat sijoittumaan hajautusympäristössä. Pitää myös muistaa, että käytetty ORBacus-väliohjelmisto on vain yksi mahdollinen CORBA-implementaatio. Laajemmat kaupalliset CORBA-väliohjelmistot saattavat olla suorituskyvyltään parempia.

Virheiden käsittelyssä sekä DCOM- että CORBA-toteutukset toimivat moitteettomasti. Tärkein ero virheiden käsittelyssä väliohjelmistojen välillä oli niiden havaitsemistapa asiakasohjelman puolella. DCOM ei tue perusrajapinnassaan poikkeuskäsittelyä, joten kutsun onnistuminen täytyy päätellä etäkutsun paluuarvosta. CORBAssa puolestaan epäonnistunut etäkutsu aiheuttaa `CORBA::SystemException`-poikkeuksen, jonka kautta saa lisätietoa tapahtuneesta virheestä. Jälkimmäinen virheenkäsittelytapa mahdollistaa loogisemman ja hienovaraisemman virhekäsittelyn asiakassovelluksessa kuin mihin DCOM kykenee. Suhteellisen yksinkertaisissa järjestelmissä tällä ei kuitenkaan ole juuri merkitystä

11.2 Teknologioiden vertailu

Luvuissa 5–7 tarkasteltiin sekä yleisellä tasolla väliohjelmistoja rakennetta että yksityiskohtaisemmin DCOM- ja CORBA-väliohjelmistojen toimintaa. Kuten tarkastelusta kävi ilmi, molemmat teknologiat sisältävät useita hajautetuille oliopohjaisille väliohjelmistoille luonteenomaisia piirteitä eri tavoin toteutettuna. Taulukossa 14 on lueteltu näitä piirteitä sekä niiden toteutustapoja DCOM- ja CORBA-väliohjelmistoissa.

Väliohjelmiston piirre	**DCOM**	**CORBA**
Hajautusprotokolla	ORPC	GIOP
Siirtosyntaksi	NDR	CDR
Perusluokka	`IUnknown`	`CORBA::Object`
Rajapinnan tunniste	`IID`	Rajapinnan nimi
Olion tunniste	`CLSID`	Rajapinnan nimi
Olion käyttötapa	Osoitin rajapintaan	Etäolioreferenssi
Virheidenkäsittelytapa	`HRESULT`-paluuarvo	Poikkeukset
Rajapintojen tyyppitiedon varasto	Tyyppikirjastot	Rajapintavarasto
Olioiden paikallistaja	Väliohjelmistopalvelin (SCM)	Oliovälitin (ORB)
Olioiden aktivoija	Väliohjelmistopalvelin (SCM)	Oliosovitin (OA)
Toteutuksien sijaintitiedon varasto	Systeemirekisteri	Toteutusvarasto

Taulukko 14. DCOM- ja CORBA-väliohjelmistojen toteutuspiirteitä.

Merkittävimmät erot DCOM- ja CORBA-väliohjelmistojen välillä sovelluskehittäjän näkökulmasta liittyvät palvelimen toteuttamien **olioiden elinikään** sekä **olioiden paikallistamiseen** liittyviin yksityiskohtiin. CORBA-palvelinsovelluksen toteuttamien olioiden olemassaolo ei juurikaan riipu asiakassovelluksen olemassaolosta. DCOM-palvelimen oliot taas syntyvät ja kuolevat enimmäkseen asiakasohjelman tarpeiden

mukaisesti. Kummankin väliohjelmiston politiikka olioiden eliniän suhteen on kuitenkin muokattavissa ohjelmallisesti. Esimerkiksi DCOMin singleton-oliot (katso luku 6.4.1) vastaavat melko tarkasti CORBAn olioiden perustoteutusta.

DCOM käyttää olioiden paikallistamiseen pääsääntöisesti käyttöjärjestelmän tarjoamaa systeemirekisteriä, kun taas CORBA-sovellukset joko toteutusvarastoa, nimipalvelua tai suoraan merkkijonomuotoisia IOR-referenssejä. CORBAn vaihtoehtoiset tavat ovat sovelluskehittäjälle läpinäkyvämpiä ja paremmin kontrolloitavissa verrattuna DCOMiin. Palvelinsovellus voi esimerkiksi välittää IOR-referenssejä asiakassovellukselle tietokantaa tai jaettuja tiedostoja käyttäen.

CORBA on toteutukseltaan avoimempi järjestelmä kuin DCOM. Sovelluskehittäjällä on mahdollisuus päästä paremmin vaikuttamaan hajautukseen liittyviin yksityiskohtiin ja säätelemään väliohjelmiston toimintaa. Toisaalta tämä lisää sovelluskehittäjän vastuuta ja jättää kehittäjän toteutettavaksi osia, jotka DCOM sisältää jo itsessään käyttöjärjestelmän tukemana.

Loppukäyttäjän näkökulmasta erot DCOM- ja CORBA-väliohjelmistojen välillä koskevat suurimmaksi osaksi asennuksen helppoutta. DCOM on tiukasti sidoksissa Windows-käyttöjärjestelmään, jonka palveluja se hyödyntää mm. käyttöoikeustarkistuksissaan. Näihin liittyvät käyttöjärjestelmän ominaisuudet täytyy ymmärtää hyvin, ennen kuin pystyy asentamaan DCOM-sovelluksia laajempaan käyttöön. Samoin DCOM-sovellusten rekisteröintiin ja päivityksiin liittyi erinäisiä ongelmia, jotka tulivat ilmi Generis Alarmerin testauksen aikana. CORBA-toteutus sitä vastoin toimi ilman erillistä konfigurointia. Täytyy kuitenkin huomata, ettei CORBA-toteutus sisältänyt minkäänlaisia käyttöoikeustarkistuksia, vaan nämä on erikseen toteutettava sovellukseen esim. CORBAn tarjoamaa turvapalvelua (*CORBA Security*) käyttämällä.

11.3 Käytännön toteutustyö

Kummallakin väliohjelmistolla toteutetut projektit muistuttavat käytännön tasolla suuresti toisiaan. Kumpikin sisältää selvästi erilliset asiakas- ja palvelinsovellukset, joita voi käsitellä ja kehittää omina kokonaisuuksina. Kummassakin rajapinnat on kuvattu IDL-tiedostossa, jonka muokkaus on olennainen osa kehitystyötä. Molemmissa on lisäksi oma IDL-kääntäjä, jolla generoidaan sovelluksen sarjallistamiseen liittyvä koodi.

Uuden DCOM-sovelluksen aloittaminen on helppoa, sillä Microsoftin Visual Studio tarjoaa projektin luontiin **ohjattuja toimintoja** (engl. *wizards*). Ne rakentavat valmiin sovellusrungon, jota lähdetään laajentamaan. ORBacus ei tällaista toimintoa tarjoa, ja uuden sovelluksen luonti on jonkin verran vaativampaa[6]. ORBacus myös toimitetaan lähdekoodimuodossa, jonka kääntäminen binäärikirjastoiksi ei ole välttämättä täysin yksinkertaista.

Koska **IDL-tiedostot** näyttelevät kummankin väliohjelmiston kohdalla merkittävä osaa, on kyseisen tiedoston käsittelyn helppous tärkeää. CORBAn käyttämä IDL-kieli erottuu DCOMin IDL-kielestä yksinkertaisuudellaan ja siisteydellään. DCOMin IDL-kieli sisältää lukuisan määrän erityyppisiä kääntämiseen ja luokkien tunnistukseen liittyviä attribuutteja, jotka tekevät kielestä hankalasti luettavaa ja muokattavaa. Visual Studio -kehitysympäristö tarjoaa tähän kuitenkin avuksi **automatisoituja toimintoja** mm. uusien luokkien ja rajapintojen lisäämiseksi.

DCOM-sovelluksen **testaus ja käyttöönotto** on myös jonkin verran hankalampaa kuin vastaavan CORBA-sovelluksen. DCOM-väliohjelmistoa käyttävä palvelinsovellus sekä mahdolliset sarjallistamiskomponentit täytyy ensin rekisteröidä järjestelmään. Lisäksi eri työasemien väliseen DCOMin käyttöön liittyy useita turvallisuusasetuksia, jotka ovat melko hankalasti hallittavissa. Toisaalta CORBA-sovellukset eivät oletuksena sisällä

---

[6] IONA-yhtiön kaupallinen CORBA-implementaatio Orbix sisältää ohjatut toiminnot uuden sovelluksen luontiin olemassaolevan IDL-tiedoston pohjalta.

käyttöoikeus- tai turvallisuustarkistuksia ja näiden toteutus ohjelmallisesti voi olla vaativaa.

Sovellusohjelmoijan kannalta katsottuna DCOM on hankalampi järjestelmä kuin CORBA, ellei kehitystyössä hyödynnä valmiita apukirjastoja (kuten ATL tai MFC), eikä kehitysympäristön tukemia ohjattuja toimintoja. Hankaluus johtunee osaksi siitä, että DCOM-väliohjelmakirjaston ohjelmointirajapinta on toteutettu perinteisiä aliohjelmakutsuja käyttäen. Kuitenkin DCOM-järjestelmä sisältää vakiorajapintoja, joita C++ -kielessä käsitellään kuin olioita. Näiden yhtäaikainen käyttö voi aiheuttaa sekaannuksia. Lisäksi DCOM-komponenttien ohjelmallinen rekisteröinti ilman apukirjastoja on monimutkaista.

11.4 Tulokset tavoitteisiin nähden

Tutkielman tulosten perusteella **DCOM täyttää tällä hetkellä kelvollisesti yrityksen ohjelmakehityksen hajautustarpeet**. DCOMin suorituskyky sekä luotettavuus ovat riittävän korkeat ja kehitystyökalut tukevat hyvin kyseistä väliohjelmistoa. Lisäksi DCOMin käyttö Visual Basic -pohjaisista kielistä on yksinkertaista, ja useat skriptikielet tukevat myös DCOM-teknologiaa. Järjestelmien koon kasvaessa DCOMin asennukseen ja ylläpitoon liittyvät hankaluudet saattavat kuitenkin kasvaa suuriksi. Myös DCOMin riippuvuus Windows-käyttöjärjestelmästä voi kehittyä jossakin vaiheessa rajoittavaksi tekijäksi järjestelmiä toimitettaessa.

Toisena mahdollisuutena tarkasteltu **CORBA-väliohjelmisto vaikuttaa tehtyjen vertailujen pohjalta olevan myös hyvä vaihtoehto hajautettujen ohjelmistojen toteutukseen**. Suorituskykynsä tai luotettavuutensa osalta se ei häviä DCOMille hajautuksen tapahtuessa eri työasemien kesken. Lisäksi CORBAn selkeämmät asennus- ja käyttötavat suosivat sen hyödyntämistä varsinkin laajemmissa, mahdollisesti lähiverkkoa suuremmissa ympäristöissä. Yrityksen tällä hetkellä käytössä olevat kehitystyökalut eivät kuitenkaan tue CORBAa yhtä hyvin kuin DCOMia ja siirtyminen uuden väliohjelmiston käyttöön saattaisi olla kallista ja resursseja vievää.

JavaRMI-väliohjelmisto on teknisten ominaisuuksiensa osalta (katso luku 5.1.3) vertailukelpoinen DCOMin ja CORBAn kanssa. Lisäksi JavaRMI:n soveltaminen kehitystyössä on yksinkertaista verrattuna esim. CORBAn käyttöön C++ -kielestä. JavaRMI:lle ei kuitenkaan suoritettu tämän tutkielman puitteissa suorituskyky- ja luotettavuustestejä, joten väliohjelmistoja ei voi näiltä osin verrata toisiinsa. Yrityksen ohjelmistokehitys on myös pääsääntöisesti tapahtunut C++ -kieltä käyttäen ja olemassa olevien kirjastojen hyödyntäminen Javasta olisi siksi hankalaa tai jopa mahdotonta. Näiden syiden vuoksi Java-kielen käyttöönotto ei vaikuta hyvältä vaihtoehdolta DCOM- ja CORBA-väliohjelmistojen rinnalla. JavaRMI:n käytön lopulliseen hylkäämiseen olisi kuitenkin tarvittu laajempia tutkimuksia ja vertailuja kuin mitä tutkielmassa toteutettiin.

12 Yhteenveto

Hajautettujen väliohjelmistojen tunnetuimmat teknologiat DCOM ja CORBA ovat olleet laajassa käytössä kohta kymmenen vuoden ajan. Tänä aikana kumpikin on vakiinnuttanut paikkansa ohjelmistotuotannon alalla, mutta todellista kilpailua ei teknologioiden kesken ole päässyt syntymään. Teknologioiden käyttökohteet ovat nimittäin olleet melko erityyppisiä. DCOM on ollut suosittu suppeissa ja keskisuurissa Windows-käyttöjärjestelmän alaisuudessa toimivissa sovelluksissa, kun taas CORBA on yleistynyt laajoissa heterogeenisissä järjestelmissä.

Tutkielma paljasti, että teknologioilla on runsaasti yhteisiä piirteitä ja kummankin kohdalla on jouduttu ratkomaan samantyyppisiä hajautukseen liittyviä ongelmia. Etäkutsujen ja parametrien välitys, olioiden ja palvelinprosessien paikannus, rajapintojen kuvaus sekä etäolioihin viittaaminen ovat näistä esimerkkejä. Näiden teknisten ominaisuuksien keskinäinen paremmuusvertailu on kuitenkin hankalaa, sillä erot tulevat selkeästi esiin vasta suuremmissa käytännön toteutuksissa.

Suuria suorituskyky- tai virheenkäsittelyeroja ei DCOMin ja CORBAn väliltä löytynyt. DCOM on kylläkin huomattavasti nopeampi paikalliskutsujen välityksessä, mutta todellisessa eri työasemien välisessä hajautuksessa tehokkuuserot väliohjelmistojen välillä ovat merkityksettömiä. DCOM osoittautui kelvolliseksi valinnaksi yrityksen hajautustarpeisiin, mutta CORBAn käyttöä tulevaisuudessa ei pidä sulkea pois. JavaRMI:n osalta tutkielmassa ei suoritettu laajempia vertailuja, mutta alustavan tarkastelun pohjalta sen käyttö ei vaikuta tällä hetkellä mielekkäältä. Lopullinen päätös vaatisi kuitenkin lisäselvityksiä mm. suorituskyvyn, luotettavuuden ja käytettävyyden osalta.

Tulevaisuudessa DCOMin merkitys tulee vähenemään, sillä sen korvaajaksi on tulossa Microsoftin .NET-teknologia. DCOM on kuitenkin yhä hyvin olennainen osa Windows-käyttöjärjestelmää, joten siirtymävaihe tulee olemaan pitkä. CORBA puolestaan on erittäin muuntautumiskykyinen ja sen IIOP-standardi sekä hyvä Java-tuki kasvattavat sen suosiota. CORBAn melko tuore laajentuminen langattomien verkkojen hajautustekniikaksi tulee myös kohentamaan sen asemaa DCOMiin verrattuna. Näiden uusien teknologioiden

kriittinen vertailu olemassa oleviin teknologioihin jäi tutkielman ulkopuolelle, mutta olisi varmasti tutkimuksen arvoinen kohde.

Yleisellä tasolla hajautetut väliohjelmistot, kuten CORBA ja DCOM, ovat kuitenkin osoittautumassa liian yhteensidotuiksi ja suljetuiksi järjestelmiksi muissa kuin lähiverkoissa käytettynä. Tällä hetkellä yleinen suuntaus on kohti väljästi yhteensidottuja palveluita. Uudet teknologiat pyrkivät integroimaan eri väliohjelmistot toimimaan keskenään saman palvelupisteen kautta. XML-RPC:n ja SOAPin kaltaiset tekniikat ovat vieneet hajautusta kohti entistä avoimempia järjestelmiä, joissa kommunikointi verkkopalvelujen (engl. *Web Services*) välillä tapahtuu XML:ää käyttämällä. Tämä muutos tulee vaikuttamaan myös DCOMin ja CORBAn kohtaloon tulevaisuudessa.

Lähteet

[And00] Anderson Richard, et al., “Professional XML”, Wrox Press, 2000.

[Bor89] Borghoff Uwe M. and Nast-Kolb Kristof, “Distributed Systems: A Comprehensive Survey”, Technical Report No TUM-18909, Technical University of Munchen, November 1989, saatavilla PDF-muodossa <URL: http://citeseer.nj.nec.com/416664.html>, viitattu 12.11.2003.

[Cal95] Callaghan B., et al., “NFS Version 3 Protocol Specification”, Internet RFC 1813, Sun Microsystems, 1995, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc1813.txt>, viitattu 1.11.2002.

[Cal96a] Callaghan B, “WebNFS Client Specification”, Internet RFC 2054, Sun Microsystems, 1996, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc2054.txt>, viitattu 16.11.2002.

[Cal96b] Callaghan B, “WebNFS Server Specification”, Internet RFC 2055, Sun Microsystems, 1996, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc2055.txt>, viitattu 16.11.2002.

[Chu97] Chung P. Emerald, Huang Yennun, Yajnik Shalini, et.al., “DCOM and CORBA Side by Side, Step by Step, and Layer By Layer”, Bell Laboratories and Lucent Technologies, Murray Hill, 1997, saatavilla HTML-muodossa <URL:http://research.microsoft.com/~ymwang/papers/HTML/DCOMnCORBA/S.html>, viitattu 8.1.2003.

[COM95] Microsoft Corporation and Digital Equipment Corporation, “The Component Object Model Specification”, Draft Version 0.9, 1995.

[COM98] Microsoft Corporation, “DCOM Architecture – White Paper”,1998, saatavilla PDF-muodossa <URL: http://socrates.coloradotech.edu/~cs630/DCOM_Architecture.pdf>, viitattu 28.1.2003.

[Cou01] Coulouris George, Dollimore Jean and Kindberg Tim, “Distributed Systems – Concepts and Design”, Third Edition, Addison-Wesley Publishing, 2001.

[Cou94] Coulouris George, Dollimore Jean and Kindberg Tim, “Distributed Systems – Concepts and Design”, Second Edition, Addison-Wesley Publishing, 1994.

[DCE] The Open Group, “DCE 1.1 : Remote Procedure Call – CAE Specification”, saatavilla HTML-muodossa <URL: http://www.opengroup.org/onlinepubs/009629399/toc.htm>, viitattu 11.10.2002.

[Gri98] Grimes, Richard,”Professional ATL COM Programming”, Wrox Press, 1998.

[ISO96] ISO, “ISO/IEC 10746-2: Open Distributed Processing – Reference Model: Foundations”, 1996, saatavilla PDF-muodossa <URL: http://www.iso.ch/iso/en/ittf/PubliclyAvailableStandards/s018836e.zip>, viitattu 10.10.2002.

[MSDN] Microsoft Developer Network Library, April 2000.

[Mur98] Murhammer Martin W, et al., “TCP/IP Tutorial and Technical Overview”, IBM Redbooks 1998, saatavilla HTML-muodossa <URL: http://www.redbooks.ibm.com/redbooks/GG243376.html>, viitattu 10.10.2002.

[OMG] Object Management Group, Inc., “CORBA Core Specification 01-12-35”, saatavilla PDF-muodossa <URL: http://www.omg.org/cgi-bin/doc?formal/01-12-35.pdf>, viitattu 10.10.2002.

[ORB01] IONA Technologies, Inc., “ORBacus For C++ and Java version - Version 4.1.0”, saatavilla WWW-muodossa <URL: http://www.iona.com>, viitattu 24.1.2003.

[ORB02] IONA Technologies, Inc., “How Can I Implement a Smart Proxy in Orbix”, Knowledge Base Article ID:1506.472, saatavilla WWW-muodossa <URL: http://www.iona.com/support/articles/1506.472.xml>, viitattu 12.11.2003.

[Orf98] Orfail Robert and Harkey Dan, “Client/Server Programming with JAVA and CORBA”, Second Edition, John Wiley & Sons, Inc., 1998.

[Pos80] Postel Jon (ed.), “User Datagram Protocol”, Internet RFC 768, Internet Engineering Task Force, 1980, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc768.txt>, viitattu 11.10.2002.

[Pos81a] Postel Jon (ed.), ”Internet Protocol”, Internet RFC 791, Internet Engineering Task Force, 1981, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc791.txt>, viitattu 11.10.2002.

[Pos81b] Postel Jon (ed.), “Transmission Control Protocol”, Internet RFC 793, Internet Engineering Task Force, 1981, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc793.txt>, viitattu 11.10.2002.

[Sri95a] Srinivasan R (ed.), “Remote Procedure Call”, Internet RFC 1831, Sun Microsystems, 1995, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc1831.txt>, viitattu 11.10.2002.

[Sri95b] Srinivasan R, “External Data Representation Standard”, Internet RFC 1832,1995, saatavilla ASCII-muodossa <URL: http://www.ietf.org/rfc/rfc1832.txt>, viitattu 11.10.2002.

[SUN] Sun Microsystems, Inc., ”Java™ Remote Method Invocation Specification – Revision 1.8”, saatavilla HTML-muodossa <URL: http://java.sun.com/products/jdk/rmi>, viitattu 13.6.2003.

[W3Ca] W3C, “Extensible Markup Language (XML) 1.0, Second Edition”, saatavilla HTML-muodossa <URL: http://www.w3.org/TR/REC-xml>, viitattu 10.10.2002.

[W3Cb] W3C, “XML Schema Part 2: Datatypes”, saatavilla HTML-muodossa <URL: http://www.w3.org/TR/2001/REC-xmlschema-2-20010502/datatypes.html>, viitattu 10.10.2002.

[Yan96] Yang Zhonghua and Duddy Keith, *CORBA: A Platform for Distributed Object Computing (A State-of-the-Art Report on OMG/CORBA)*,Operating System Review, 30(2), 1996, 4-31.

Liitteet

Liite 1. Generis Alarmer -järjestelmän IDL-kuvaus

```
/************************************************
Alarmer Server –sovelluksen toteuttamat rajapinnat.
************************************************/
[
     object,
     uuid(80660D5B-0E28-11D5-BF41-00D0B70EDAC8),
     dual,
     helpstring("IAlarmerServer rajapinta"),
     pointer_default(unique)
]
interface IAlarmerServer : IDispatch
{
     [id(1), helpstring("Rekisteröi asiakkaan")]
     HRESULT RegisterClient(
          [in] BSTR _bsHost,
          [in] BSTR _bsProgPath,
          [in] BSTR _bsPrefix,
          [in] long _lDatabaseId,
          [in] BSTR _bsDomainId,
          [out,retval] IAlarmerContext** _pNewContext);
};

[
     object,
     uuid(79772E71-3949-11d5-BF49-00D0B70EDAC8),
     dual,
     helpstring("IAlarmerServerSystem rajapinta"),
     pointer_default(unique)
```

```
]
interface IAlarmerServerSystem : IDispatch
{
     [id(1), helpstring("Päivittää konfiguraation")]
     HRESULT UpdateConfiguration();

     [id(2), helpstring("Tyhjentää viestivaraston")]
     HRESULT ClearCache();
};

[
     object,
     uuid(E093E106-0EF0-11D5-BF41-00D0B70EDAC8),
     dual,
     helpstring("IAlarmerContext rajapinta"),
     pointer_default(unique)
]
interface IAlarmerContext : IDispatch
{
     [id(1), helpstring("Välittää hälytyksen")]
     HRESULT PostMessage(
          [in]           BYTE _bLevel,
          [in]           UINT _uiError,
          [in]           BSTR _bsParams,
          [out,retval]   VARIANT_BOOL* _pbRet);
};
```

```
/************************************************
Alarmer Client –moduulin toteuttamat rajapinnat.
************************************************/
[
     object,
     uuid(E9EDDC9F-0E31-11D5-BF41-00D0B70EDAC8),
     dual,
     helpstring("IAlarmerClient rajapinta"),
     pointer_default(unique)
]
interface IAlarmerClient : IDispatch
{
     [id(1), helpstring("Lähettää hälytyksen palvelimelle”)]
     HRESULT Alarm(
          [in] enumLogLevel             _eLevel,
          [in] LONG                     _lError,
          [in,defaultvalue(NULL)]       BSTR _bsParams,
          [out,retval] VARIANT_BOOL*    _pbRet);

     [id(2), helpstring("Lähettää hälytyksen palvelimelle
     sekä lokitiedostoon")]
     HRESULT Log(
          [in] enumLogLevel             _eLevel,
          [in] LONG                     _lError,
          [in] BSTR                     _bsErrorDesc,
          [in,defaultvalue("|")]        BSTR _bsParams,
          [out,retval]VARIANT_BOOL*     _pbRet);
};
```

Liite 2. CORBA-testisovelluksen IDL-kuvaus

```
module Alarmer
{
     /***** Hälytysten vakavuusvakiot ********/
     const unsigned short AL_DEBUG      =    1;
     const unsigned short AL_INFO       =    2;
     const unsigned short AL_WARNING    =    3;
     const unsigned short AL_ERROR      =    4;
     const unsigned short AL_FATAL      =    5;

     /*****************************************/
     /* Alarmer-kohtainen poikkeus.           */
     /*****************************************/
     exception AlarmException
     {
          string description;
     };

     /*****************************************/
     /* AlarmerContext rajapinta ja sen metodit.*/
     /*****************************************/
     interface AlarmerContext
     {
          /* Asiakasohjelman tunniste */
          attribute string    Tag;

          /* Välittää hälytyksen  */
          boolean   Alarm(
               in unsigned short level,
               in short msgid,
```

```
            in string message
        ) raises (AlarmException);

        /* Sulkee kontekstin käytön jälkeen */
        void CloseContext()
            raises (AlarmException);
    };

    /*****************************************/
    /* AlarmerServer rajapinta ja sen metodit. */
    /*****************************************/
    interface AlarmerServer
    {
        /* Palvelimen nimi-attribuutti (vain luku) */
        readonly attribute  string Server;

        /* Rekisteröi asiakkaan */
        AlarmerContext GetContext(
            in string tag,
            in string client
        ) raises (AlarmException);
    };
};
```

Liite 3. Esimerkki XDR-kuvauksesta

```
/*************************************************
Yksinkertainen kuvaus RPC-palvelusta, joka lukee ja
kirjoittaa tiedostoja.
*************************************************/

constMAX = 100;
typedef int FileIdentifier;
typedef int FilePointer;
typedef int Length;

/*************************/
/* Tietue datan siirtoon.*/
/*************************/
struct Data {
     int Length;
     char Buffer[MAX];
};

/************************************************/
/* Tietue kirjoitusparametrien välittämiseksi.   */
/************************************************/
struct writeargs {
     FileIdentifier f;
     FilePointer position;
     Data data;
};
```

```
/*************************************************/
/* Tietue lukuparametrien välittämiseksi.        */
/*************************************************/
struct readargs {
     FileIdentifier;
     FilePointer position;
     Length length;
};

/*************************************************/
/* Varsinaisen RPC-rajapinnan operaatioiden kuvaus.*/
/*************************************************/
program FILEREADWRITE {
     version VERSION {
          void WRITE(writeargs)=1;
          Data READ(readargs)=2;
     }=2;
}=9999;
```

Liite 4. Esimerkit XML-RPC- ja SOAP-viesteistä.

XML-RPC -viesti liitettynä POST-komentoon:

```
POST /RPC2 HTTP/1.0
User-Agent: Frontier/5.1.2 (WinNT)
Host: www.stocktrades.com
Content-Type: text/xml
Content-length: 181
<?xml version="1.0"?>
<methodCall>
   <methodName>examples.GetLastTradePrice</methodName>
   <params>
      <param>
         <value><string>DIS</string></value>
      </param>
   </params>
</methodCall>
```

SOAP-viesti liitettynä POST-komentoon:

```
POST /StockQuote HTTP/1.1
Host: www.stocktrades.com
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
SOAPAction: "Some-URI"
<SOAP-ENV:Envelope
   xmlns:SOAP-ENV="schemas.xmlsoap.org/soap/envelope/"
   SOAP-ENV:encodingStyle="Schemas.xmlsoap.org/soap/encoding/">
   <SOAP-ENV:Body>
      <m:GetLastTradePrice xmlns:m="Some-URI">
      <symbol>DIS</symbol>
      </m:GetLastTradePrice>
   </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Liite 5. Testausympäristön laitteisto

Kone A (palvelin)

Käyttöjärjestelmä:	Microsoft Windows XP Professional
Prosessori:	Intel 2.5 GHz
Keskusmuistia:	512 Mt
Verkkokortti:	Intel PRO/100 VM

Kone B

Käyttöjärjestelmä:	Microsoft Windows 2000 Professional
Prosessori:	Intel 500 MHz
Keskusmuistia:	256 Mt
Verkkokortti:	Intel PRO/100+

Kone C

Käyttöjärjestelmä:	Microsoft Windows 2000 Professional
Prosessori:	Intel 550 MHz
Keskusmuistia:	256 Mt
Verkkokortti:	3Com EtherLink XL 10/100 PCL

Kone D

Käyttöjärjestelmä:	Microsoft Windows 2000 Professional
Prosessori:	Intel 450 MHz
Keskusmuistia:	256 Mt
Verkkokortti:	3Com EtherLink XL 10/100 PCL

Liite 6. Yhteenveto tehdyistä suorituskykytesteistä

Asiakas-	**Generis Alarmer -ohjelmisto**			**DCOM-testisovellus**			**CORBA-testisovellus**		
prosesseja (n	**A.n**	**B.n**	**C.n**	**A.n**	**B.n**	**C.n**	**A.n**	**B.n**	**C.n**
1	7356	1660	-	32680	3073	-	7357	2563	-
3	5921	1975	4958	31226	5178	8576	7111	3171	7536
6	5026	1989	5972	29426	5230	12712	7020	3266	9074
12	4322	1993	6158	23064	4791	13823	6761	3237	9582
15	4215	1989	6327	20318	4901	13576	6678	3230	9557
24	4122	1928	6322	16445	4584	13515	6604	3251	9442
30	4076	1867	5931	16203	4560	13166	6497	3183	9306
60	-	-	5791	-	-	11754	-	-	9332
75	-	-	5195	-	-	11326	-	-	9260
120	-	-	4970	-	-	9660	-	-	8405

Taulukko 15. Palvelinsovelluksen keskimäärin käsittelemien saapuvien viestien määrä yhdessä sekunnissa.

Asiakas-	Generis Alarmer -ohjelmisto			DCOM-testisovellus			CORBA-testisovellus		
prosesseja (n)	A.n	B.n	C.n	A.n	B.n	C.n	A.n	B.n	C.n
1	0,14	0,60	-	0,03	0,33	-	0,14	0,39	-
3	0,51	1,52	0,61	0,10	0,58	0,35	0,42	0,95	0,40
6	1,19	3,02	1,00	0,20	1,15	0,47	0,85	1,84	0,66
12	2,78	6,02	1,95	0,52	2,50	0,87	1,78	3,71	1,25
15	3,56	7,54	2,37	0,74	3,06	1,10	2,25	4,64	1,57
24	5,82	12,45	3,80	1,46	5,24	1,78	3,63	7,38	2,54
30	7,36	16,07	5,06	1,85	6,58	2,28	4,62	9,43	3,22
60	-	-	10,36	-	-	5,10	-	-	6,43
75	-	-	14,44	-	-	6,62	-	-	8,10
120	-	-	24,14	-	-	12,42	-	-	14,00

Taulukko 16. Yhden viestin välitykseen asiakassovellukselta palvelinsovellukselle kulunut keskimääräinen aika millisekunneissa.