Timo Kuosmanen

Tuoterunko hajautetussa ympäristössä

Tietotekniikan
pro gradu -tutkielma
1. maaliskuuta 2007

Jyväskylän yliopisto

Tietotekniikan laitos

Jyväskylä

Tekijä: Timo Kuosmanen
Yhteystiedot: tkuo@iki.fi
Työn nimi: Tuoterunko hajautetussa ympäristössä
Title in English: Product Platform in Distributed Environment
Työ: Tietotekniikan pro gradu -tutkielma
Sivumäärä: 95
Tiivistelmä: Tässä tutkielmassa käsitellään hajautetussa, heterogeenisessä ympäristössä toimivan tuoterungon kehitystä. Tutkielma keskittyy hajautetun ympäristön tuoterungolle asettamien haasteiden ja tuoterungon hajautuksessa vaadittavien ominaisuuksien käsittelyyn. Tutkielman tarkoitus on toimia perustana hajautetussa ympäristössä toimivan tuoterungon suunnittelulle ja erityisesti tuoterungon hajautuksen toteutukselle.
English abstract: This thesis examines product platform development in distributed, heterogeneous environment. It focuses on presenting the challenges introduced by the distributed environment in product platform development and on techniques rising to these challenges. The aim of this thesis is to act as a guideline for the design and implementation of distributed product platform.
Avainsanat: Tuoterunko, Ohjelmistokehys, Hajautettu järjestelmä, Väliohjelmisto, Ohjelmistokomponentti
Keywords: Product platform, Software framework, Distributed system, Middleware, Software component

Esipuhe

Tämä tutkielma syntyi pääosin vuoden 2006 aikana työskennellessäni Patria Aviation Oy:n palveluksessa. Haluan kiittää Patriaa mahdollisuudesta tämän opinnäytetyön tekemiseen ja kaikkia työkavereita mukavasta hetkistä työn parissa. Erityisesti haluan kiittää Mikko Niemistä tutkielman ohjauksesta sekä mielenkiintoisista keskusteluista tutkielmaan liittyvistä ja vähän sen ulkopuolisistakin asioista. Kiitokset myös lehtori Ari Viinikaiselle tutkielman ohjauksesta.

Haluan kiittää myös perhettäni sekä ystäviäni olemassaolosta ja tuesta opiskelujeni aikana. Lopuksi vielä vastaus kaikille, joiden kärsivällisyys tämän tutkielman valmistumisen osalta on ehtinyt loppua ennen minua – Nyt se on valmis.

Sanasto

CORBA (Common Object Request Broker Architecture) Hajautettuihin olioihin perustuvien järjestelmien kehitystä tukeva OMG:n määritys.

COTS (Commercial off-the-shelf) Kaupallinen suoraan hyllystä saatava tuote. Ohjelmistoteollisuudessa valmis ohjelmistotuote, jota voidaan käyttää yrityksen sisäisen ohjelmistotuotannon tukena, kustannusten karsimiseksi ja kehitysprosessien nopeuttamiseksi.

Kehyksen erikoistamisrajapinta (specialization interface) Ohjelmistokehyksen laajennoskohtien muodostama rajapinta.

Koottava kehys (black-box framework) Ohjelmistokehys, jonka laajennoskohdat muodostavat kehyksen määrittelemät laajennoskohtien täydentämiseen vaadittavat rajapinnat ja jonka erikoistaminen tapahtuu vaadittujen rajapintojen mukaisia luokkia toteuttamalla ja liittämällä luokkien ilmentymiä kehykseen.

Kehyksen laajennoskohta (hot spot) Ohjelmistokehyksen toteuttamatta jätetty kohta, jonka toteuttamalla kehystä voidaan erikoistaa tapauskohtaisella toiminnallisuudella.

Kehyksen sovitus tai erikoistaminen (framework adaptation tai specialization) Ohjelmistokehyksen täydentäminen valmiiksi ohjelmistokokonaisuudeksi toteuttamalla kehyksen laajennoskohtia.

MOTS (Military off-the-shelf tai Modifiable off-the-shelf) Erityisesti sotateollisuudessa käytetty COTS-ohjelmistotuote, jonka mukana tuotteen tilaajalle toimitetaan myös toteutuksen lähdekoodi.

Muunneltava kehys (white-box framework) Ohjelmistokehys, jonka laajennoskohdat muodostavat kehyksen määrittelemät kantaluokat ja jonka erikoistaminen tapahtuu kehyksen määrittelemiä kantaluokkia perimällä.

Ohjelmistokehys (software framework) Tietyn ongelma-alueen ratkaisun arkkitehtuurin ja osittaisen toteutuksen sisältävä ohjelmarakenne. Kehyksen vaillinaista toteutusta täydentämällä kehyksestä voidaan erikoistaa valmiita ohjelmakokonaisuuksia.

Ohjelmistokomponentti (software component) Ohjelmarakenne, joka määritellään sen tarjoamien, vaatimien ja teknisten rajapintojen avulla. Tekninen rajapinta määrittelee, miten komponentin asetuksia voidaan säätää.

OMA (Object Management Architecture) OMG:n referenssiarkkitehtuuri, johon järjestön hajautettujen sovellusten kehitystä tukevat määritykset perustuvat.

OMG (Object Management Group) Ohjelmistoteollisuudessa toimivien organisaatioiden muodostama kansainvälinen, voittoa tavoittelematon järjestö, joka pyrkii standardoimaan hajautetussa, heterogeenisessä ympäristössä toimivien sovellusten kehitystä.

ORB (Object Request Broker) Hajautettujen olioden välisiä pyyntöjä ja vasteita hajautetussa järjestelmässä välittävä ohjelmistoväylä.

Tuoteperhe (product family) Saman tuoterungon avulla kehitettyjen ohjelmistojen joukko.

Tuoterunko (product platform) Uudelleenkäytettävät ohjelmistorakenteet ja näiden rakenteiden kehitystä sekä käyttöä tukevan ohjelmistoarkkitehtuurin yhdistävä ohjelmistotekniikan menetelmä kehittää ohjelmistoja.

Variaatiopiste (variation point) Tuoterungon auki jättämä suunnitteluratkaisu, jonka kiinnittämällä tuoterungon avulla kehitettävän ohjelmiston yksilöllisiä vaatimuksia voidaan täyttää.

Väliohjelmisto (middleware) Hajautetun järjestelmän ympäristön heterogeenisyyden hajautetuilta sovelluksilta peittävä ohjelmistokerros.

Sisältö

Kuvat

1 Johdanto

Ohjelmistotekniikassa on jo pitkään pyritty ohjelmistojen uudelleenkäytettävyyteen. Ohjelmistojen uudelleenkäyttö parantaa ohjelmistojen laatua ja nopeuttaa niiden kehitysprosesseja. Perinteisesti uudelleenkäytettävyyteen on pyritty kehittämällä ohjelmistojen modularisointia ja hyödyntämällä samoja moduuleja eri ohjelmistoissa. Tätä kehitystä ovat tukeneet esimerkiksi aliohjelmat, olioparadigma ja erilaiset ohjelmistokomponenttiteknologiat sekä näiden käyttöä tukevat kirjastot ja ohjelmistokehykset.

Ohjelmistojen modularisointi ei yksinään ole avain menestyksekkääseen uudelleenkäyttöön. Uudelleenkäytettävien moduulien kehitys ja hyödyntäminen uusien ohjelmistojen kehityksessä edellyttävät myös moduulien kehitystä ja käyttöä ohjaavaa ohjelmistoarkkitehtuuria. *Tuoterunko (product platform)* on osoittautunut tehokkaaksi ohjelmistojen uudelleenkäyttöä tukevaksi menetelmäksi, joka yhdistää uudelleenkäytettävät komponentit ja niiden kehitystä sekä käyttöä ohjaavan arkkitehtuurin.

Samaan aikaan ohjelmistojen modularisoinnin kehittyessä mikroprosessorien ja tietokoneverkkojen kehitys on mahdollistanut useista tietoliikenneyhteydellä toisiinsa kytketyistä koneista muodostuvien *hajautettujen järjestelmien (distributed system)* kehityksen. Hajautettu järjestelmä voi sisältää erilaisia laitteistojen ja käyttöjärjestelmien muodostamia alustoja, tietoliikenneyhteyksiä ja eri menetelmillä toteutettuja ohjelmistoja. Hajautettu, heterogeeninen ympäristö asettaa ohjelmistojen kehitykselle uusia haasteita, joita perinteisissä ohjelmistotekniikan menetelmissä ei ole otettu huomioon.

Tuoterungon sovellusalue määrittelee alueen, jolla tuoterunko tukee ohjelmistojen kehitystä. Kun tuoterungon sovellusalue sijoittuu hajautettuun, heterogeeniseen ympäristöön, on tuoterunkoarkkitehtuurissa ja komponenttien kehityksessä huomioitava monia seikkoja, joita ei vaadita ei-hajautetussa ympäristössä toimivalta tuoterungolta. Tämän tutkielman tarkoitus on selvittää tuoterungolta hajautetussa ympäristössä vaadittavia ominaisuuksia ja muodostaa viitekehys, jonka avulla hajautetussa ympäristössä toimivan tuoterungon kehityksen haasteisiin voidaan varautua.

Perusedellytys tuoterungon perustamiselle on hyvin määritelty sovellusalue, jota voidaan tutkia sovellusalueanalyysin avulla. Sovellusalueanalyysiin on kehitetty monia eriä menetelmiä, joiden käsittely tässä tutkielmassa kuitenkin sivuutetaan. Tämän tutkielman lähtökohtana voidaan pitää tilannetta, jossa tuoterungon sovellusalueanalyysi on jo suoritettu ja sovellusalueen vaatimukset selvitetty.

Tutkielma muodostuu kolmesta osasta. Ensimmäinen osa käsittelee tuoterungon

ominaisuuksia, perustamista ja käyttöä organisaation ohjelmistotuotannossa (luku 2) sekä tuoterungon toteutuksessa tärkeässä roolissa olevien oliopohjaisten ohjelmistokehysten ominaisuuksia ja kehitystä (luku 3). Toinen osa tarkastelee hajautetun ympäristön tuoterungon kehitykselle asettamia haasteita ja hajautetussa ympäristössä toimivilta ohjelmistoilta vaadittavia ominaisuuksia (luku 4) sekä järjestelmän hajautuksen hallintaa väliohjelmistojen avulla (luku 5). Tutkielman kolmannessa osassa muodostetaan aiempien lukujen pohjalta tuoterungon hajautuksen suunnittelua tukeva viitekehys (luku 6) ja käsitellään tutkielman esimerkkinä toimivan tuoterungon toteutusta (luku 7).

2 Tuoterunko ohjelmistotuotannossa

Tuoterunko on ohjelmistojen uudelleenkäyttöön perustuva organisaation ohjelmistotuotantoa tehostava menetelmä, joka yhdistää uudelleenkäytettävät ohjelmistokomponentit komponenttien kehitystä ja käyttöä tukevaan ohjelmistoarkkitehtuuriin. Tuoterunkopohjaisessa ohjelmistokehityksessä ohjelmistojen uudelleenkäytettävyys otetaan huomioon jo ohjelmistojen arkkitehtuurisuunnittelussa, jolloin uudelleenkäytöstä tulee hallitumpaa ja suunnitelmallisempaa [9, luku 7]. Tämä luku esittelee tuoterunkojen ominaisuuksia, kehitystä ja käyttöä yleisellä tasolla ja toimii pohjana myöhemmin tässä tutkielmassa tehtävälle hajautetun tuoterungon tarkastelulle.

2.1 Tuoterungon ominaisuuksia

Tuoterunkopohjainen ohjelmistokehitys on suhteellisen laaja tutkimusalue ja kirjallisuudesta löytyy siihen monia eri lähestymistapoja. Tämän tutkielman aihepiiriin ei kuulu laaja-alainen vertailu erilaisista lähestymistavoista tuoterunkopohjaiseen ohjelmistokehitykseen, vaan tutkielmassa tyydytään esittelemään tuoterunkojen yleisiä ominaisuuksia ja määrittelemään parhaiten tämän tutkielman tarkoitusta tukeva tuoterunkomalli.

Tässä luvussa tarkastellaan tuoterungon ominaisuuksia ja rakennetta. Lisäksi käydään lyhyesti läpi tuoterungon avulla saavutettavia etuja ohjelmistokehityksessä ja verrataan näitä etuja rungon kehityksen kustannuksiin. Luvussa 2.1.1 esitellään ensin luokittelu erilaisille ohjelmistoarkkitehtuureille ja esitetään, mihin tuoterunkoarkkitehtuuri tässä luokittelussa sijoittuu. Luvussa 2.1.2 käsitellään tarkemmin tuoterungon rakennetta ja luvussa 2.1.3 tuoterungon muunneltavuutta, joka on näkyvin tuoterunkoarkkitehtuurin muista ohjelmistoarkkitehtuureista erottava tekijä. Lopuksi luvuissa 2.1.4 ja 2.1.5 vertaillaan tuoterungon etuja ja kustannuksia.

2.1.1 Arkkitehtuuri

Ohjelmistoarkkitehtuurit voidaan luokitella *selittäviin*, *ohjaaviin* ja *mahdollistaviin* arkkitehtuureihin sen mukaan, mikä niiden rooli on ohjelmistotuotannossa [49, s. 157–158]. Selittävän arkkitehtuurikuvauksen tarkoitus on lähinnä kuvata ohjelmiston rakenne ohjelmiston eri sidosryhmille ja se voidaan laatia myös ohjelmiston rakentami-

sen jälkeen. Ohjaavalla arkkitehtuurikuvauksella on merkittävämpi rooli ohjelmistojen kehityksessä. Se ohjaa kehitysprosessia alusta lähtien ja vaikuttaa ohjelmiston kehitykseen koko kehitysprosessin ajan. Mahdollistavalla arkkitehtuurikuvauksella on selittävää ja ohjaavaa kuvausta konkreettisempi rooli ohjelmistojen kehityksessä. Se tarjoaa konkreettisia, korkean tason mekanismeja tuotteiden vaatimusten toteuttamiseen.

Tuoterunkoarkkitehtuuri (product-line architecture) toimii pohjana tuoterungon avulla kehitettävien yksittäisten ohjelmistojen arkkitehtuureille ja sen pitäisi pystyä pääosin täyttämään yksittäisten ohjelmistojen vaatimukset. Tuoterungon arkkitehtuuri *mahdollistaa* uusien ohjelmistojen rakentamisen rungon pohjalta ja toimii näiden ohjelmistojen ohjaavana sekä selittävänä arkkitehtuurina [49, s. 158–159]. Saman tuoterungon avulla kehitetyt ohjelmistot muodostavat rungon sovellusalueelle *tuoteperheen (product family)*. Tuoteperheen jäsenet noudattavat siis samaa, tuoterungon määrittelemää arkkitehtuuria.

2.1.2 Rakenne

Kirjallisuudessa tuoterunkoon liittyviä käsitteitä esitellään monista eri näkökulmista. Eräs näkökulma, johon myös tämän tutkielman määritelmä tuoterungolle perustuu, on tuoterungon käsitteistön esittely rungon ja sen avulla muodostetun tuoteperheen sisältämien rakenteiden perusteella [41] [9]. Nämä rakenteet ovat arkkitehtuuri, komponentti ja järjestelmä.

Arkkitehtuurilla tarkoitetaan tuoterunkoarkkitehtuuria, jonka tuoteperheen jäsenet jakavat. Komponenteilla tarkoitetaan ohjelmistokokonaisuuksia, jotka yhdessä toteuttavat rungon toiminnallisuuden ja joita hyödynnetään tuoteperheen jäsenten kehityksessä. Järjestelmällä tarkoitetaan tuoterunkoarkkitehtuurin ja tuoterungon sisältämien komponenttien avulla kehitettyä tuoteperheeseen kuuluvaa ohjelmistoa.

Bosch [9, luvut 10 ja 11] antaa komponenteille kaksi eri määritelmää: perinteiset komponentit ja *oliokehykset (object-oriented framework)*. Perinteinen komponentti on ohjelmistokokonaisuus, jolle on määritelty sen tarjoamat, vaatimat ja konfigurointiin liittyvät rajapinnat sekä sen laatuominaisuudet [9]:

> *A software component is a unit of composition with explicitly specified provided, required and configuration interfaces and quality attributes.*

Oliokehys sisältää tietyn ongelma-alueen ratkaisujen tuottamiseen tarvittavan arkkitehtuurin ja osittaisen toteutuksen [22]:

> *A framework is a set of classes that embodies an abstract design for solutions to a family of related problems.*

Boschin [9, luku 11] mukaan tuoterunko voidaan muodostaa useista kehyksistä, joista jokainen tarjoaa ratkaisun tietyn sovellusalueen osan toteuttamiseen ja jotka yhdessä, tuoterunkoarkkitehtuurin ohjaamana muodostavat koko tuoterungon sovellusalueen toiminnallisuuden.

Boschin [9] esittelemässä mallissa tuoterunko koostuu joukosta komponentteja, jotka voivat olla joko perinteisiä komponentteja tai oliokehyksiä. Tuoterunkoarkkitehtuurin sisältämät ominaisuudet ja ne toteuttavat komponentit voidaan luokitella sen mukaan, onko niiden käyttö valinnaista vai pakollista tuoteperheeseen kuuluvaa ohjelmistoa muodostettaessa [34].

Tuoterunkoon kuuluu yleensä tuoterunkoarkkitehtuurin mukainen *ohjelmistoalusta*, joka on pakollinen kaikissa tuoteperheen ohjelmistoissa [49, s. 168]. Lisäksi runko voi sisältää vaihtoehtoisten komponenttien ryhmiä sekä puhtaasti valinnaisia komponentteja [34]. Valinnaisia komponentteja voidaan valita vapaasti ohjelmiston käyttöön ja vaihtoehtoisten komponenttien ryhmästä on valittava jokin komponenteista mukaan ohjelmistoon. Lisäksi tuoteperheeseen kuuluva ohjelmisto voi sisältää pelkästään kyseistä ohjelmistoa varten kehitettyjä komponentteja, joiden on kuitenkin oltava tuoterunkoarkkitehtuurin mukaisia.

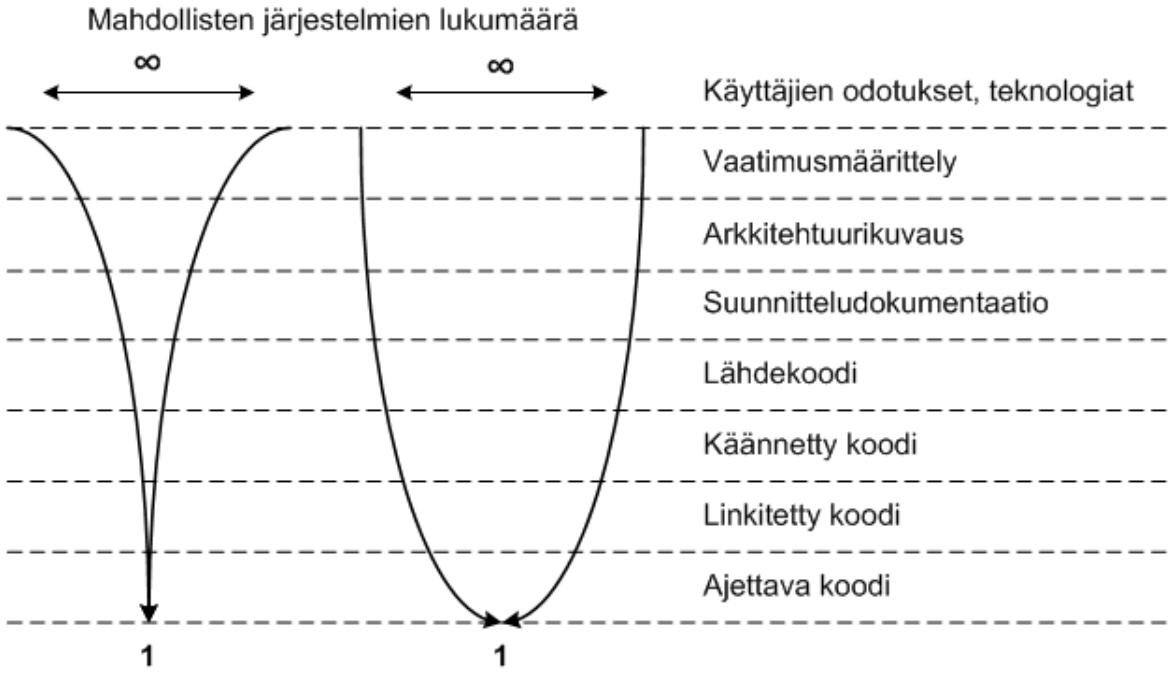
2.1.3 Muunneltavuus

Jotta tuoterunkopohjainen ohjelmistokehitys olisi mahdollista, on kehitettävään tuoteperheeseen kuuluvilla ohjelmistoilla oltava riittävästi yhteisiä ominaisuuksia. Tuoterungon on tarjottava keinot yhteisten ominaisuuksien toteuttamiseen, mutta myös mahdollistettava ohjelmistojen yksilöllisten ominaisuuksien toteuttaminen. Tähän tavoitteeseen päästään rungon *muunneltavuuden (variability)* avulla [34].

Ohjelmistokehitysprosessin eri vaiheissa tehtävät ratkaisut lisäävät jokainen vuorollaan rajoituksia kehitettävälle ohjelmistolle kunnes tuloksena on enää yksi tiettyyn käyttötarkoitukseen rajattu, ajettava sovellus. Toisin sanoen jokainen vaihe pienentää mahdollisten lopullisten sovelluksien lukumäärää ja vähentää ohjelmiston muunneltavuutta. Jotta tuoteperheeseen kuuluvien ohjelmistojen yksilölliset vaatimukset voidaan täyttää, on tuoterunkoa kehitettäessä näitä rajoittavia suunnitteluratkaisuja lykättävä yksittäisten ohjelmistojen kehitykseen asti. Tuoterungon lykättyjä suunnitteluratkaisuja kutsutaan rungon *variaatiopisteiksi (variation point)* [34].

Kuvassa 2.1 on vertailtu muunneltavuuden muutosta perinteisessä ohjelmistokehitysprosessissa ja tuoterunkoa hyödyntävässä prosessissa. Vasemmanpuoleinen kuvaaja esittää tilannetta perinteisessä ohjelmistokehitysprosessissa ja oikeanpuoleinen kuvaaja tuoterunkoa hyödyntävässä prosessissa. Kuvasta voidaan nähdä, kuinka tuoterun-

koa hyödyntävässä prosessissa mahdollisten lopullisten sovelluksien lukumäärä säilyy suurempana kehitysprosessin edetessä kuin perinteissä prosessissa.



Kuva 2.1: Muunneltavuus ohjelmistokehitysprosessin eri vaiheissa. [34]

Tuoterunkoa hyödyntävän ohjelmiston kehitysprosessin vaihetta, jossa valitaan jonkin tuoterungon variaatiopisteen täyttävä suunnitteluratkaisu, kutsutaan variaatiopisteen *kiinnittämisajankohdaksi (binding time)* [34]. Jos tuoterungon toteutuksessa hyödynnetään eksplisiittistä dynaamista linkitystä, kiinnittämisajankohta voi olla myös sovelluksen ajoaikana. Tässä tapauksessa variaatiopisteen kiinnityksen voi tehdä myös ohjelmiston loppukäyttäjä.

Tuoterunkoa kehitettäessä tuoteperheen yhteiset vaatimukset selvitetään *sovellusalueanalyysin (domain analysis)* avulla. Sovellusalueanalyysia käytetään apuna mallinnettaessa tuoteperheen vaatimukset täyttävän tuoterungon *piirteitä (feature)*. Piirteellä tarkoitetaan toiminnallista yksikköä, joka toteuttaa joukon toiminnallisia ja laadullisia vaatimuksia [34]:

> *A logical unit of behavior that is specified by a set of functional and quality requirements.*

Sovellusalueanalyysiin ja -mallinnukseen on kehitetty monia eri menetelmiä [36] [46] [45] [34] [24], joiden käsittely tässä tutkielmassa kuitenkin sivuutetaan.

Piirteet voidaan nähdä ohjelmistojen rakennusosina, jotka vastaavat ohjelmistojen vaatimuksiin [34]. Näin ollen ohjelmistojen välisiä eroja tuoteperheen sisällä, eli

tuoterungon muunneltavuutta, voidaan mallintaa piirteiden avulla. Tuoterunko sisältää joukon piirteitä, jotka voidaan luokitella pakollisiin, valinnaisiin ja vaihtoehtoisiin piirteisiin [36]. Pakolliset piirteet ovat piirteitä, joiden tulee sisältyä jokaiseen tuoteperheen ohjelmistoon. Valinnaisilla piirteillä voidaan tuoda lisäarvoa ohjelmiston käyttöön, mutta ne eivät ole pakollisia. Vaihtoehtoisten piirteiden joukosta jonkin piirteen on sisällyttävä ohjelmistoon.

Tuoterungon muunneltavuus on sovitettava siten, että tuoteperheen ohjelmistojen vaatimukset pystytään täyttämään ja toisaalta ohjelmistojen kehitys on tehokasta [34]. Mitä enemmän variaatiopisteitä eli muunneltavuusmahdollisuuksia tuoterunko sisältää, sitä enemmän erilaisia ohjelmistoja sen avulla voidaan kehittää. Liika muunneltavuus voi kuitenkin vaikuttaa heikentävästi tuoteperheen ohjelmistojen kehitysprosessien tehokkuuteen, koska tuoterunko sisältää vähemmän valmiita suunnitteluratkaisuja ja ne on toteutettava jokaisen ohjelmiston kohdalla erikseen.

2.1.4 Edut

Tuoterunko lisää ohjelmistojen uudelleenkäyttöä organisaatiossa, mikä tuo mukanaan monia etuja [9, luku 7] [49, s. 160–161]. Tuoteperheen jäsenten kehitysprosessit nopeutuvat, koska ohjelmiston arkkitehtuuri ja perustoiminnallisuus ovat jo olemassa. Ohjelmistojen laatu paranee kun uudelleenkäytettävät ohjelmistokokonaisuudet tulevat testatuiksi useissa eri ohjelmistoissa sekä ympäristöissä ja niiden ylläpito on hallittua. Tuoterungon käyttö yhtenäistää myös tuoteperheeseen kuuluvien ohjelmistojen kehitysprosesseja sekä käytettäviä menetelmiä ja työkaluja, mikä helpottaa uusien projektien aloittamista.

Vaikka uudelleenkäytön lisääntymisestä saadut hyödyt ovatkin tunnetuimpia tuoterungon etuja, ne eivät kuitenkaan ole ainoat perusteet tuoterungon käytölle. Perinteisessä ohjelmistokehitysprosessissa kehitettävän ohjelmiston laatu- ja toiminnalliset vaatimukset täyttävät suunnitteluratkaisut tehdään hyvin aikaisessa vaiheessa kehitysprosessia. Jos vaatimukset muuttuvat prosessin aikana tai jos ohjelmistoa halutaan käyttää alkuperäisestä suunnitelmasta poikkeavassa ympäristössä, suunnitteluvaiheessa tehtyjen päätösten muuttaminen voi tulla kalliiksi [32].

Tuoterungon variaatiopisteiden ansiosta tuoteperheen jäsenten suunnitteluun liittyviä ratkaisuja voidaan siirtää myöhempään vaiheeseen ohjelmistokehitysprosessissa. Näiden suunnittelupäätöksien lykkäämisien avulla voidaan varautua ohjelmiston vaatimusten ja suoritusympäristön muutoksiin [32]. Dynaamisen linkityksen mahdollistavan tuoterungon avulla päätöksiä voidaan lykätä aina sovelluksen ajoaikaan asti.

2.1.5 Kustannukset

Tuoterunkopohjaisen ohjelmistokehityksen avulla voidaan saavuttaa huomattavia taloudellisia etuja perinteiseen ohjelmistokehitykseen verrattuna, uudelleenkäytettävyyden lisääntymisen ja uusien tuoteperheen ohjelmistojen kehitysprosessien nopeutumisen ansiosta. Tuoterungon kehitys ja käyttö aiheuttavat kuitenkin myös kustannuksia ja yrityksen pitää pystyä vertaamaan näitä kustannuksia tuoterungon käytöstä saataviin taloudellisiin etuihin.

Tuoterungon kehitys ja ylläpito ovat toimia, jotka yrityksen on yleensä rahoitettava itse ilman ulkopuolista avustusta. Usein suurin haaste tuoterunkopohjaiseen ohjelmistokehitykseen siirryttäessä on, että rungon kehitys viivästyttää yhden tai useamman ohjelmiston julkaisemista [9, s. 321]. Lisäksi tuoterungon kehitys ja ylläpito vaativat huomattavan työpanoksen, joka ei tuo suoraa tuottoa myytävien tuotteiden muodossa [9, s. 321].

Tuoterungon taloudellisen kannattavuuden arviointiin on esitetty myös laskennallisia malleja. Weissin ym. esittämässä mallissa [24, s. 45–48] arvioidaan, että tuoterungon mukanaan tuomat taloudelliset edut peittävät rungon kehittämisen kustannukset tuoteperheen kolmannen tai neljännen jäsenen kohdalla. Mutta kuten Cohen [12] toteaa, Weiss ym. perustavat arvionsa oletuksille tuoterungon rakenteesta, erityisesti rungon avulla kehitettävien ohjelmistojen vaatimasta räätälöinnin määrästä.

Cohen [12] esittää myös, että tuoterungon uudelleenkäytettävyydestä seuraa muitakin kustannuksia kuin pelkästään uudelleenkäytettävien komponenttien kehittämisestä aiheutuvat kustannukset. Nämä kustannukset aiheutuvat tuoterungon tarjoamien komponenttien soveltamisesta uusien ohjelmistojen kehityksessä ja ovat verrannollisia uusien ohjelmistojen vaatimaan räätälöinnin määrään. Parhaassa tapauksessa ohjelmisto pystytään kokoamaan suoraan tuoterungon tarjoamien komponenttien avulla ilman räätälöintiä, jolloin lisäkustannuksia ei tule. Pahimmassa tapauksessa räätälöinnin kustannukset ylittävät ne kustannukset, jotka olisivat aiheutuneet ohjelmiston kehittämisestä ilman tuoterunkoa.

2.2 Tuoterungon perustaminen

Siirtyminen perinteisestä ohjelmistokehityksestä tuoterunkopohjaiseen ohjelmistokehitykseen on suuri muutos organisaatiossa ja onnistunut siirtyminen edellyttää organisaatiolta huomattavaa panosta siirtymisen suunnitteluun ja toteutukseen [9, s. 189]. Organisaation lähtöasetelmista riippuen tuoterungon perustamiseen on olemassa eri tapoja, joiden vaatima työmäärä vaihtelee. Tutkimusten ja kokemusten perusteella on

myös löydetty eri vaiheita, jotka toistuvat aina tuoterunkoa perustettaessa. Näitä vaiheita voidaan hyödyntää tuoterungon perustamista suunniteltaessa. Rungon perustamisen ja siihen liittyvien teknisten yksityiskohtien toteuttamisen lisäksi organisaation rakenteen on mukauduttava uuteen ohjelmistokehitysmalliin.

2.2.1 Perustamistavat

Organisaatio voi siirtyä tuoterunkopohjaiseen ohjelmistokehitykseen usealla eri tavalla. Bosch [9, s. 166–169] esittää eri tapoja tuoterungon perustamiselle, joista voidaan erottaa perustamistavan valintaan vaikuttavia tekijöitä. Näitä tekijöitä ovat ainakin tuoterungon kehitykseen käytettävissä olevat resurssit, jo aloitetut tulevaan tuoteperheeseen kuuluvat ohjelmistoprojektit sekä olemassa olevat ohjelmistot, jotka halutaan liittää tuoteperheeseen.

Edellä mainittujen tekijöiden pohjalta Bosch [9, s. 167–169] esittää kaksi eri lähestymistapaa tuoterungon perustamiseen: *evolutiivinen (evolutionary)* ja *kumouksellinen (revolutionary)* lähestymistapa. Lisäksi Bosch esittää kaksi eri lähtökohtaa tuoteperheen perustamiselle: olemassa olevien tuotteiden hyödyntäminen tuoteperheen perustamisessa ja kokonaan uuden tuoteperheen kehitys. Yhdistämällä esittämänsä lähestymistavat ja lähtökohdat Bosch muodostaa neljä eri tuoterungon perustamistapaa.

Ensimmäinen Boschin [9, s. 166] olemassa olevia ohjelmistoja hyödyntävästä perustamistavasta perustuu evolutiiviseen lähestymistapaan. Evolutiivisessa lähestymistavassa tuoterunkoarkkitehtuuri kehitetään olemassa olevien ohjelmistojen arkkitehtuurien pohjalta ja tuoterungon komponentit kootaan kehittämällä olemassa olevista ohjelmistokomponenteista yksi kerrallaan yleiskäyttöisiä. Toinen olemassa olevia ohjelmistoja hyödyntävä tapa, eli kumouksellinen lähestymistapa, on kehittää olemassa olevien ohjelmistojen vaatimukset täyttävä, kokonaan uusi tuoterunko ja korvata olemassa olevat ohjelmistot tuoteperheeseen tuoterungon avulla kehitetyillä ohjelmistoilla.

Kokonaan uuden tuoteperheen perustaminen Boschin [9, s. 168] evolutiivisen lähestymistavan mukaisesti tapahtuu kehittämällä tuoterunkoa asteittain, lisäämällä sen ominaisuuksia tuoteperheeseen lisättävien jäsenten vaatimusten perusteella. Tässä tapauksessa tuoterunko kehittyy vähitellen tuoteperheen jäsenten määrän lisääntyessä. Kumouksellisessa lähestymistavassa uusi tuoteperhe perustetaan puolestaan kehittämällä tuoterunko ensin valmiiksi ja hyödyntämällä uutta tuoterunkoa tuoteperheen jäsenten kehityksessä.

2.2.2 Perustamisen vaiheet

Tuoterungon perustamisen vaiheista esitetyt mallit ja menetelmät [46] [36] [9] [24] riippuvat usein tuoterungon rakenteesta tai keskittyvät johonkin tiettyyn perustamisen vaiheeseen. Seuraavat vaiheet ovat kuitenkin erotettavissa useista malleista ja ne kuvaavat hyvin myös tässä tutkielmassa käytettävän tuoterunkomallin mukaisen rungon perustamista: sovellusalueanalyysi ja -määrittely, arkkitehtuurisuunnittelu, komponenttien suunnittelu sekä komponenttien toteutus. Tuoterungon kehitykseen ja ylläpitoon liittyviä toimia voidaan kuvata yhteisellä termillä *sovellusaluemallinnus (domain engineering)*.

Sovellusalueanalyysin ja -määrittelyn avulla määritellään tuoteperheen jäsenten yhteiset vaatimukset ja piirteet sekä toisaalta myös eroavaisuudet, jotta voidaan tunnistaa rungon variaatiopisteet. Arkkitehtuurisuunnittelun tavoitteena on suunnitella sellainen arkkitehtuuri, jonka avulla pystytään vastaamaan kaikkiin tuoteperheen ohjelmistojen asettamiin vaatimuksiin [9, s. 190]. Arkkitehtuuri määrittelee komponenttien väliset suhteet siten, että määritellyt piirteet voidaan toteuttaa. Arkkitehtuurisuunnittelussa on erityisesti otettava huomioon rungon muuntelumahdollisuus eli variaatiopisteet.

Komponentin suunnittelussa määritellään komponentin toiminnallisuus, rajapinnat ja laatuominaisuudet [9, luku 10 ja 11]. Tuoterungon arkkitehtuuri ohjaa ja asettaa rajoituksia komponentin suunnittelulle. Komponentin tulee pystyä toteuttamaan arkkitehtuurin siltä vaatima toiminnallisuus. Komponentin tarjoama toiminnallisuus määrittyy sen tarjoaman rajapinnan kautta. Komponentin ei kuitenkaan välttämättä ole toteutettava koko toiminnallisuuttaan itse, vaan se voi edellyttää toisten komponenttien apua määrittelemällä vaadittuja rajapintoja. Komponentin suunnittelussa on otettava huomioon myös komponentin mukauttaminen osaksi runkoa ja rungon avulla muodostettua uutta ohjelmistoa. Mukauttamisessa voidaan hyödyntää komponentin konfigurointirajapintaa sekä oliokehyksen avulla toteutetuissa komponenteissa kehyksen tarjoamia mekanismeja.

Komponentin toteutuksessa toteutetaan komponentin tarjoaman rajapinnan määrittelemä toiminnallisuus huolehtien laatuominaisuuksien täyttymisestä [9, luku 10 ja 11]. Toteutuksessa voidaan myös hyödyntää tai joutua ottamaan huomioon aikaisemmista toteutuksista periytynyt ohjelmakoodi *(legacy code)*.

2.2.3 Rooli organisaatiossa

Tuoterunko voi olla tehokas keino toteuttaa organisaation ohjelmistotuotantoa. Teknisesti hyvin toteutettu tuoterunko yksinään ei kuitenkaan riitä takaamaan menestystä, myös rungon ylläpito ja käyttö on suunniteltava hyvin. Myös tuoterunkoa hyödyntävän

organisaation rakenteen on tuettava rungon ylläpitoa ja käyttöä ohjelmistotuotannossa.

Usein esitetty malli jakaa organisaation tuoterungon kehityksestä vastaavaan ja tuoteperheeseen kuuluvien tuotteiden kehityksestä vastaaviin yksiköihin [43] [39] [9]. Yksiköiden on oltava vuorovaikutuksessa keskenään, jotta tuotekehityksestä vastaava yksikkö saa tietoa runkoa koskevista päivityksistä ja toisaalta tuoterunkoa kehittävä yksikkö tietoa rungolta vaadittavista uusista ominaisuuksista [43, s. 334–337]. Myös markkinointiyksiköllä on mallissa tärkeä rooli, koska se välittää tietoa asiakkaiden uusista vaatimuksista.

Bosch [9, luku 14] esittää, että tämä perinteinen malli sopii erityisesti suurille organisaatioille, jotka työllistävät yli 100 henkilöä tuoterungon kehitykseen ja käyttöön. Boschin mukaan pienempien organisaatioiden huolena on työntekijöiden sijoittaminen yksikköön, joka ei tee tuotteita suoraan asiakkaiden käyttöön. Mallin suurin etu on, että rungon sisältämät komponentit pysyvät tarpeeksi yleiskäyttöisinä kun niitä ei kehitetä yksittäisen ohjelmiston kehityksen yhteydessä. Haittapuolina voidaan pitää kommunikoinnin hankaluutta ja sen myötä yksittäisen ohjelmiston tarvitsemien yleiskäyttöisten komponenttien kehityksen viivästymistä.

Bosch [9, luku 14] esittää lisäksi kolme vaihtoehtoista organisaatiomallia, joita voidaan käyttää tuoterunkopohjaisessa ohjelmistotuotannossa: kehitysyksikkö, liiketoimintayksiköt ja hierarkiset sovellusaluemallinnusyksiköt.

Kehitysyksikkömallissa kaikki ohjelmistokehitys, mukaan lukien tuoterungon ja tuoteperheen ohjelmistojen kehitys, on keskitetty samaan yksikköön [9, s. 302–304]. Myös tuoterungon yhteisten komponenttien kehitys hoidetaan projektimuotoisesti ja jokainen yksikön työntekijä voi olla mukana myös rungon kehityksessä. Mallin etuna on hyvä kommunikaatioyhteys kehitystyöhön osallistuvien välillä ja se soveltuu erityisesti pieneen alle 30 hengen organisaatioon, jonka liiketoiminta perustuu ohjelmistoprojekteihin eikä niinkään ohjelmistuotteiden kehitykseen. Jos organisaation koko kasvaa, se on jaettava yksiköihin jonkin toisen mallin mukaisesti.

Liiketoimintayksiköihin perustuvassa mallissa jokainen liiketoimintayksikkö kehittää omat tuoteperheeseen kuuluvat ohjelmistonsa [9, s. 304–309]. Myös yhteisten tuoterunkoon kuuluvien komponenttien kehitys on hajautettu liiketoimintayksiköihin. Malli soveltuu 30–100 hengen organisaatioon ja sen etuna on yhteisten komponenttien kehityksen tehokas hajautus. Mallin ongelmat aiheutuvat yksiköiden välisen kommunikaation hankaluudesta ja yhteisten komponenttien omininaisuuksien liian tuotekohtaisesta painottumisesta.

Hierarkisia sovellusaluemallinnusyksiköitä voidaan käyttää suurissa organisaatioissa, jotka käyttävät ohjelmistotuotannossaan useita keskinäisen hierarkian muodostavia tuoterunkoja [9, s. 312–315]. Mallin avulla voidaan hallita todella suuria tuoteperheitä,

jotka työllistävät satoja työntekijöitä.

2.3 Tuoterungon käyttö ja evoluutio

Tuoterungon käyttöön perustuvan ohjelmistokehitysprosessin vaiheet eroavat perinteisen ohjelmistokehitysprosessin vaiheista. Myös tuoterungon evoluutio on erilaista kuin perinteisen ohjelmiston tapauksessa.

2.3.1 Käyttö uusien ohjelmistojen kehityksessä

Tuoterungon kehityksessä pitäisi pyrkiä siihen, että *sovelluksien tuottaminen (application engineering)* tuoterungon avulla olisi mahdollisimman helppoa [40]. Mitä pienempi työmäärä on tehtävä tuoteperheen ohjelmistojen kehityksessä sitä kannattavampaa tuoterungon käyttö on. Usein tuoterungon kehityksessä keskitytään tuoterungon sisältämien yhteisten komponenttien kehitykseen. Jotta tuoterungon käyttö olisi mahdollisimman tehokasta, myös sovelluksien tuottamiseen liittyviin prosesseihin tulisi kiinnittää huomiota.

Tuoterunkoa hyödyntävän ohjelmiston kehitysprosessissa suoritetaan normaaliin tapaan ensin ohjelmiston vaatimusmäärittely. Tuoteperheen yhteisten vaatimuksien osalta määrittely on tehty jo tuoterungon kehityksen yhteydessä ja tuotekohtaisessa määrittelyssä keskitytään ohjelmiston yksilöllisiin vaatimuksiin. Vaatimuksia määriteltäessä tulee kuitenkin ottaa huomioon myös tuoterungon tarjoamat mahdollisuudet ja sen asettamat rajoitukset sekä informoida niistä myös asiakkaita [49, s. 167] [9, s. 260–261].

Tuoteperheeseen kuuluva ohjelmisto noudattaa pääasiallisesti tuoterunkoarkkitehtuuria. Varsinaista arkkitehtuurisuunnittelua ei siis tarvita, mutta tuoterunkoarkkitehtuuri on kuitenkin sovitettava ohjelmistoon sopivaksi. Tämä voi tarkoittaa tuoterunkoarkkitehtuurin karsimista, laajentamista ja mahdollisten ristiriitojen selvittämistä ohjelmistokohtaisen ja tuoterungon arkkitehtuurin välillä [9, s. 262].

Ohjelmiston ominaisuudet pyritään toteuttamaan mahdollisimman pitkälle tuoterungon tarjoamien komponenttien avulla. Ohjelmiston yksityiskohtainen suunnittelu ja toteutus koostuu käytettävien komponenttien valinnasta sekä näiden komponenttien ilmentymien luomisesta, jota Bosch [9, s. 269] nimittää yksinkertaisesti komponentin *ilmentämiseksi (instantiation)*. Tämän lisäksi halutun toiminnallisuuden saavuttamiseksi voidaan kehittää tuoterunkoarkkitehtuurin mukaisia ohjelmistokohtaisia komponentteja. Ohjelmistokohtaisista komponenteista on kuitenkin pyrittävä tekemään mahdollisimman yleiskäyttöisiä, jotta ne voitaisiin myöhemmin liittää osaksi tuoterunkoa [9, s. 259].

Kun ohjelmiston komponentit on valittu, ne on koottava yhteen ja määritettävä niiden väliset yhteydet eli on suoritettava tuotteen yhdistäminen. Komponenttien yhdistäminen voi vaatia ohjelmakoodin kirjoittamista, komponenttien ajonaikaista ilmentämistä sekä komponenttien asetusten määrittämistä [9, s. 277]. Viimeinen vaihe ennen ohjelmiston julkaisua on ohjelmiston testaus. Tuoterunkoa hyödyntävän ohjelmiston testauksessa voidaan joutua tekemään ohjelmiston toiminnan testauksen lisäksi myös valittujen komponenttien testausta ohjelmiston toimintaympäristössä.

Lähteessä [40] esitetään tuoterunkopohjaisen ohjelmistokehityksen malli, jossa kehitysprosessi jaetaan kahteen vaiheeseen. Ensimmäisessä vaiheessa suoritetaan edellä kuvatut vaiheet, jonka jälkeen niin sanotussa iteraatiovaiheessa verrataan ohjelmiston toimintaa sille asetettuihin vaatimuksiin. Jos ohjelmiston toiminta täyttää vaatimukset, se kelpuutetaan ja iteraatiovaihe päättyy. Jos toiminta ei täytä vaatimuksia, suoritetaan iteraatiokierros. Kierroksen aikana suoritetaan uudelleen arkkitehtuurin sovittaminen sekä komponenttien valitseminen ja asetusten määrittäminen. Iteraatiokierroksia suoritetaan kunnes ohjelma täyttää sille asetetut vaatimukset. Tämän mallin avulla voidaan varautua myös kehitysprosessin aikana muuttuviin vaatimuksiin.

2.3.2 Evoluutio

Tuoterungon evoluutio on monitahoista ja siihen vaikuttavia tekijöitä on enemmän kuin perinteisen ohjelmiston tapauksessa, koska kaikkien tuoteperheeseen kuuluvien ohjelmistojen vaatimukset vaikuttavat tuoterungon evoluutioon [9, luku 13] [49, luku 7].

Tuoterungon komponenttien kehitys ja niihin tehtävät muutokset vaikuttavat kaikkiin tuoteperheen ohjelmistoihin, joissa kyseiset komponentit ovat käytössä. Tuoterunko voi myös jakautua useammaksi tuoterungoksi, jos esimerkiksi tuoteperheen ohjelmistojen väliset erot kasvavat liian suuriksi [49, s. 163]. Uusien tuoteperheen jäsenien vaatimukset vaikuttavat myös tuoterungon evoluutioon.

Tuoteperheen ohjelmistojen kehityksen yhteydessä kehitettävistä komponenteista pyritään tekemään yleiskäyttöisiä ja näitä komponentteja voidaan lisätä tuoterungon komponenttien joukkoon. Tällä voi myös olla haitallinen vaikutus tuoterunkoon, jos lisätyt komponentit aiheuttavat rungon suuntautumisen tukemaan liikaa tietyn tyyppisiä tuotteita [49, s. 163].

3 Oliopohjaiset ohjelmistokehykset

Ohjelmistokehyksiä (software framework) käytetään usein apuna tuoterunkojen toteutuksessa. Tässä luvussa käsitellään ohjelmistokehyksiä siitä lähtökohdasta, miten ne tukevat tuoterunkojen toteutusta. Monet tuoterunkojen ominaisuudet ja suunnittelumenetelmät soveltuvat myös ohjelmistokehyksiin. Ohjelmistokehykset voidaankin nähdä myös alemman abstraktiotason toteutusmenetelmänä tuoterungoille.

Luku esittelee ensin oliopohjaisten ohjelmistokehysten ominaisuuksia, kehysten suunnitteluprosessia sekä kehysten suunnitteluun ja käyttöön liittyviä haasteita. Lopuksi esitellään komponenttien käyttöön perustuva kehys sekä lähteessä [20] kuvattu arkkitehtuurimalli komponenttipohjaisen kehyksen rakentamiseksi. Komponenttipohjaisen kehyksen voidaan ajatella vastaavan edellisessä luvussa esitetyn tuoterunkomallin mukaista toteutusta.

3.1 Ohjelmistokehysten ominaisuuksia

Olio-ohjelmointi lisää ohjelmakoodin ja -rakenteiden uudelleenkäyttöä, mutta ei ratkaise suurempien ohjelmistokokonaisuuksien ja ohjelmistojen arkkitehtuurien uudelleenkäyttöön liittyviä haasteita. Oliopohjainen ohjelmistokehys laajentaa olio-ohjelmoinnin rakenteiden uudelleenkäytettävyyden koskemaan tietyn kohdealueen ohjelmistokokonaisuuden arkkitehtuuria ja koko toteutusta [27]. Ohjelmistokehys on runko, jota täydentämällä voidaan toteuttaa erilaisia ohjelmistokokonaisuuksia.

Luokkakirjastot (class library) koostuvat yleiskäyttöisistä luokista, joita voidaan uudelleenkäyttää eri sovelluksissa. Kirjastojen tarjoamat luokat on yleensä tarkoitettu hyvin yleisen tason ongelmien ratkaisuun eivätkä siten tarjoa suoraa ratkaisua rajatun kohdealueen ongelmiin [27]. *Suunnittelumallit (design patterns)* ovat valmiita ratkaisumalleja jonkin tietyn ongelman ratkaisemiseksi, joita on kehitetty yleistämällä ohjelmistoissa usein toistuvia ja hyväksi todettuja suunnitteluratkaisuja. Suunnittelumallit eivät tarjoa valmista toteutusta ongelman ratkaisemiseksi vaan ainoastaan abstraktin ratkaisumallin [51].

Ohjelmistokehyksellä on aina rajattu kohdealue ja se tarjoaa keinot ohjelmistokokonaisuuden toteuttamiseen kyseisellä kohdealueella [27]. Kehys sisältää kohdealueen ohjelmistokokonaisuuden toteuttamiseen tarvittavan arkkitehtuurin ja osittaisen toteutuksen. Kehyksen vaillinaista toteutusta täydentämällä voidaan rakentaa kohdealueen

ongelman ratkaiseva toteutus. Kehys tarjoaa korkeamman abstraktiotason kohdealueen ohjelmistokokonaisuuksien toteuttamiseen kuin luokkakirjastot ja sisältää yleensä useita suunnittelumallien konkreettisia ilmentymiä. Kehyksen rakennetta voidaan usein kuvata suunnittelumallien avulla.

Ohjelmistokehykset mahdollistavat ohjelmarakenteiden uudelleenkäytön lisäksi siis myös suunnitteluratkaisujen uudelleenkäytön. Ohjelmistokehykseen sisältyvä ohjelmistorakenne tekee kuitenkin oletuksia omasta ympäristöstään, mikä asettaa enemmän vaatimuksia rakenteen käyttäjälle kuin luokkakirjaston luokan käyttö. Käyttäjän on tunnettava ohjelmistorakenteen tarjoaman palvelun lisäksi myös rakenteen *konteksti* eli rakenteen suhteet muihin kehyksen rakenteisiin [51]. Kehykseen sisältyvän ohjelmistorakenteen ympäristöstään tekemien oletuksien vuoksi kehykseen sisältyvää rakennetta ei yleensä voi käyttää erillään sen kontekstista.

3.1.1 Kohdealue

Ohjelmistokehys on aina suunnattu tietylle *kohdealueelle (domain)*, jolla sen on tarkoitus tukea ohjelmistokokonaisuuksien kehitystä. Kohdealue määräytyy usean tekijän summana. Yksi osatekijä kohdealueen määrityksessä on sen avulla kehitettävä ohjelmistokokonaisuus. Kehyksen avulla voi olla tarkoitus kehittää kokonaisia ajettavia sovelluksia, jolloin kyseessä on *sovelluskehys (application framework)* [27]. Jos tarkoitus on kehittää pienempiä *alijärjestelmiä (subsystem)* tai *komponentteja*, kyseessä on *komponenttikehys (framelet)* [37].

Ohjelmistokokonaisuuden lisäksi kehykset ovat kohdennettu tietylle *liiketoimintaalueelle (business unit)* kuten mobiilijärjestelmiin ja tietylle *sovellusalueelle (application domain)* kuten käyttöliittymiin [27]. Ohjelmistokehyksen kohdealue voisi siis esimerkiksi olla mobiilijärjestelmien käyttöliittymäkomponentit.

3.1.2 Arkkitehtuuri

Ohjelmistokehys muodostuu kohdealueella tarvittavista uudelleenkäytettävistä ohjelmarakenteista eli kehyksen *jäädytetyistä kohdista (frozen spot)* sekä *laajennoskohdista (hot spot)* [55] [49]. Laajennoskohdat ovat kehyksen arkkitehtuurissa määriteltyjä, toteuttamatta jätettyjä kohtia, joita voidaan täydentää tapauskohtaisilla ohjelmistorakenteilla. Täydentämällä kehyksen laajennoskohtia eli *kehyksen sovituksella (framework adaptation*) kehyksestä saadaan tuotettua erilaisia valmiita ohjelmistokokonaisuuksia, jotka voivat kehyksen kohdealueesta riippuen olla esimerkiksi ajettavia sovelluksia tai komponentteja.

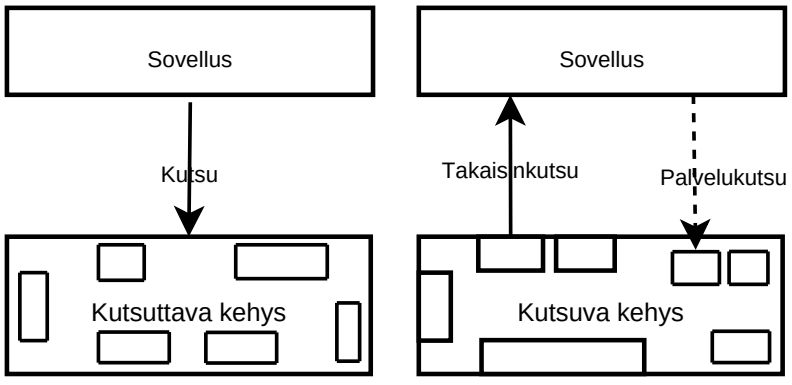
Jäädytetyt kohdat muodostavat kehyksen kohdealueen yhteiset ominaisuudet. Ke-

hyksen arkkitehtuuri määrittelee jäädytettyjen kohtien väliset suhteet sekä jäädytettyjen kohtien suhteet kehyksen laajennoskohtiin. Kehyksen laajennoskohtia täydentämällä muodostetut ohjelmistokokonaisuudet noudattavat siis kehyksen määrittämää arkkitehtuuria.

3.1.3 Kontrolli

Perinteisen luokkakirjastoja hyödyntävän sovelluksen kontrolli on sovelluksella, joka kutsuu kirjastojen palveluja. Ohjelmistokehystä hyödyntävällä sovelluksella voi olla myös niin sanottu *käänteinen kontrolli (inversion of control)* [38]. Käännetyn kontrollin toteutuksen mahdollistavat kehyksen laajennoskohtien *koukkukohdat (hook)* [27], joihin kehystä täydentävät sovelluskohtaiset ohjelmistorakenteet voivat rekisteröityä. Kehystä käyttävä sovellus voi alustaa kehyksen ja luovuttaa kontrollin kehykselle, joka luovuttaa kontrollin välillä takaisin kehykseen rekisteröityneille sovelluskohtaisille osille *takaisinkutsujen (callback)* avulla. Sovelluskohtaiset osat voivat yleensä lisäksi kutsua kehyksen sisältämiä palveluja. Tämä käänteinen kontrollin kulku tunnetaan myös nimellä *Hollywood-periaate ("Don't call us, we'll call you").*

Ohjelmistokehykset voidaan luokitella *kutsuviin (calling framework)* ja *kutsuttaviin (called framework)* kehyksiin sen mukaan, millainen kehystä käyttävän sovelluksen kontrollin kulku on [38]. Kutsuvat kehykset ovat aktiivisia kehyksiä, jotka kääntävät sovelluksen ja kehyksen välisen kontrollin. Kutsuttavat kehykset ovat passiivisia kehyksiä, joiden palveluja sovellus kutsuu luokkakirjastojen tavoin. Kuvassa 3.1 on esitetty kontrollin kulku kutsuvaa ja kutsuttavaa ohjelmistokehystä hyödyntävässä sovelluksessa.



Kuva 3.1: Kontrollin kulku kutsuvassa ja kutsuttavassa ohjelmistokehyksessä.

3.1.4 Erikoistaminen

Kehyksen sovitusta valmiiksi ohjelmistokokonaisuudeksi voidaan nimittää myös *kehyksen erikoistamiseksi (framework specialization)* [31] [49]. Oliopohjaisten kehysten erikoistamisessa käytetään hyväksi olio-ohjelmoinnin menetelmiä.

Usein käytetty menetelmä kehyksen erikoistamisessa on *periytyminen (inheritance)*. Kehyksen laajennoskohdat voidaan toteuttaa *kantaluokkien (base class)* avulla, joita perimällä kehyksen laajennoskohtien täydentäminen ja kehyksen erikoistaminen tapahtuu [22]. Erikoistaminen toteutetaan lisäämällä haluttu toiminnallisuus perittyjen aliluokkien kantaluokkien metodit *syrjäyttäviin (override)* metodeihin. Kehyksen sisältämät kantaluokat voivat myös olla *abstrakteja luokkia (abstract class)* [22], jolloin kantaluokat toimivat ainoastaan rajapintana kehyksen sisältämille palveluille. Aliluokista luodut oliot rekisteröidään kehykselle takaisinkutsuttaviksi ja valmiissa ohjelmistokokonaisuudessa kontrolli siirtyy kehykseltä perityille aliluokille *dynaamisen sidonnan (dynamic binding)* avulla kehyksen kutsuessa kantaluokissa määriteltyjä metodeja.

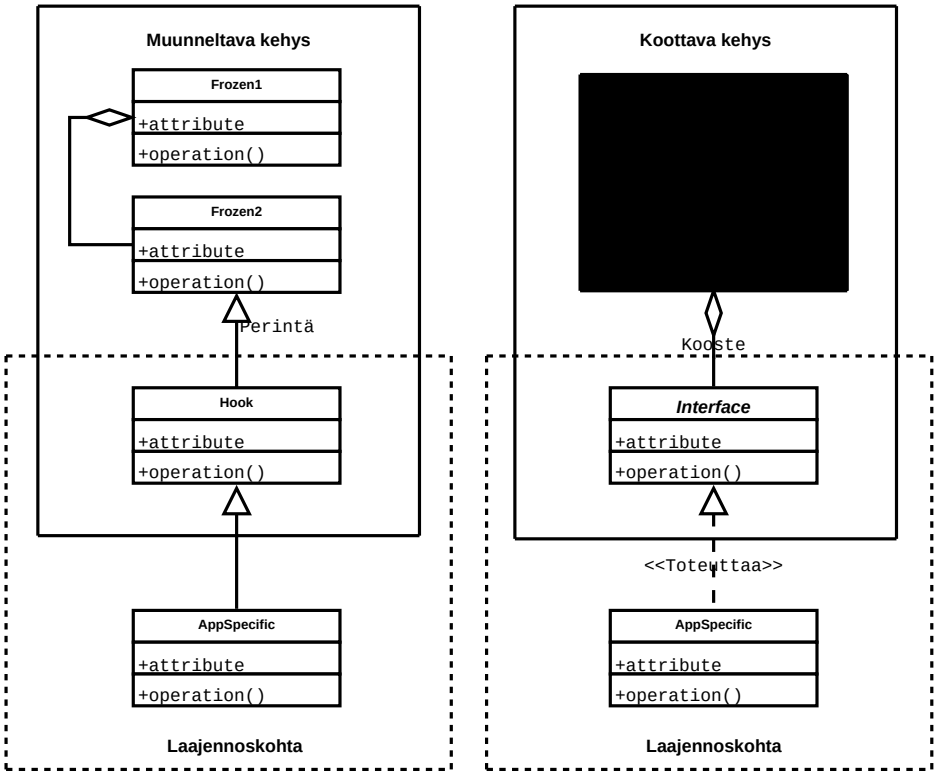
Kehyksen laajennoskohdassa voidaan myös määritellä pelkästään kohtaan rekisteröitävältä ohjelmistorakenteelta vaadittava rajapinta. Tällöin erikoistaminen tapahtuu luomalla kehyksen sisältämien, vaaditun rajapinnan toteuttavien ohjelmarakenteiden ilmentymiä ja liittämällä luodut ilmentymät osaksi kehystä. Kehyksen ohjelmarakenteisiin ei siis lisätä toiminnallisuutta vaan haluttu toiminnallisuus saadaan aikaiseksi luomalla erilaisia ilmentymiä ja muodostamalla erilaisia ilmentymien *kokoonpanoja (composition)* [22]. Valmiin ohjelmistokokonaisuuden ohjelmakoodi on ilmentymien luontia ja muita alustustoimenpiteitä lukuunottamatta kehyksessä.

3.1.5 Kehystyypit

Kehykset voidaan jakaa erikoistamistavan mukaan kahteen eri tyyppiin: *muunneltaviin (white-box)* ja *koottaviin (black-box)* kehyksiin [22]. Kuvassa 3.2 on esitetty muunneltavan ja koottavan kehyksen toimintaperiaate.

Muunneltavien kehysten erikoistaminen perustuu periyttämiseen ja kehyksen laajennoskohdat täydennetään kantaluokkien toimintaa laajentamalla. Muunneltavien kehykset ovat hyvin joustavia ja niiden toimintaa voidaan erikoistamalla muokata voimakkaasti. Käyttäjän on kuitenkin tunnettava tarkkaan kehyksen toteutuksen sisäinen rakenne ja laajennoskohdat toteuttavien kantaluokkien sisäinen toteutus [22]. Tämä hankaloittaa muunneltavan kehyksen käyttöä ja käytön oppimista.

Koottavien kehysten erikoistaminen perustuu kehyksen laajennoskohtien määrittelemien rajapintojen mukaisten ohjelmistorakenteiden ilmentymien luomiseen. Koottavien kehysten toimintaa ei voida muokata yhtä voimakkaasti kuin muunneltavien ke-



Kuva 3.2: Muunneltava ja koottava kehys.

hysten, mutta niiden käyttö on helpompaa [22]. Kehyksen käyttäjän ei tarvitse tuntea kehyksen sisäistä rakennetta, riittää kun tuntee kehyksen laajennoskohdat, laajennoskohtien täyttämisessä käytettävien ohjelmarakenteiden palvelurajapinnat ja kehyksen yleisen toimintaperiaatteen.

Muunneltavia kehyksiä käytetään yleensä kun kehyksen kohdealueella tarvittavaa toiminnallisuutta ei tunneta vielä tarpeeksi hyvin. Muunneltava kehys kehittyy usein koottavaksi kehykseksi kun kohdealueella tarvittavien palveluiden ja niiden välisten suhteiden tuntemus tarkentuu [22]. Usein kehykset sisältävät kuitenkin sekä muunneltavien että koottavien kehysten piirteitä.

3.1.6 Kehysten yhdistely

Esimerkiksi tuoterungon toteutuksessa ohjelmistokehyksiä joudutaan usein yhdistelemään [9, s. 244]. Kehysten yhdistämiseen liittyvät ongelmat voidaan välttää, jos yhdistettävät kehykset voidaan suunnitella jo etukäteen yhdistettäväksi. Usein joudutaan kuitenkin yhdistämään itsenäisiä, toisistaan erillään kehitettyjä kehyksiä, jotka voivat olla esimerkiksi ostettuja kaupallisia kehyksiä. Itsenäisten kehysten yhdistämisessä voi tulla ongelmia, jotka johtuvat kehysten ympäristölleen asettamista rajoitteista ja siitä,

että kehykset on suunniteltu erikoistettaviksi eivätkä yhdistettäväksi toisiin kehyksiin.

Ongelmia voi tulla erityisesti kahden tai useamman kutsuvan kehyksen yhdistämisessä [9, s. 244]. Yleisin esimerkki itsenäisten kehysten yhdistämisestä on käyttöliittymäkehyksen yhdistäminen sovelluskehykseen, jonka avulla varsinainen sovelluslogiikka toteutetaan. Käyttöliittymäkehys on yleensä kutsuva kehys, jonka *tapahtumasilmukka (event loop)* pyörii sovelluksen pääsäikeessä ja kutsuu sieltä *tapahtumankäsittelijöitä (event handler).* Kun tällainen käyttöliittymäkehys halutaan yhdistää sovelluskehykseen, jolla on oma kontrollisilmukkansa, on kontrollin jakamisen ongelma ratkaistava ennen kehysten yhdistämistä.

Usein käytetty ratkaisu kontrollin jakamiseen on tehdä yhdestä kehyksestä kutsuva, aktiivinen kehys ja lopuista kehyksistä passiivisia, kutsuttavia kehyksiä [9, s. 245]. Esimerkin käyttöliittymäkehykselle voitaisiin antaa sovelluksen pääkontrolli ja tehdä sovelluslogiikasta huolehtivasta kehyksestä passiivinen kehys, jota käyttöliittymäkehys kutsuu. Tämä ei ole kuitenkaan aina mahdollista, jos esimerkiksi kehysten toteutusta ei ole mahdollista muokata. Toinen vaihtoehto on sijoittaa kehykset omiin säikeisiinsä ja säilyttää kehykset kutsuvina kehyksinä [9, s. 245]. Tällöin pitää kuitenkin huolehtia säikeiden välisestä synkronoinnista.

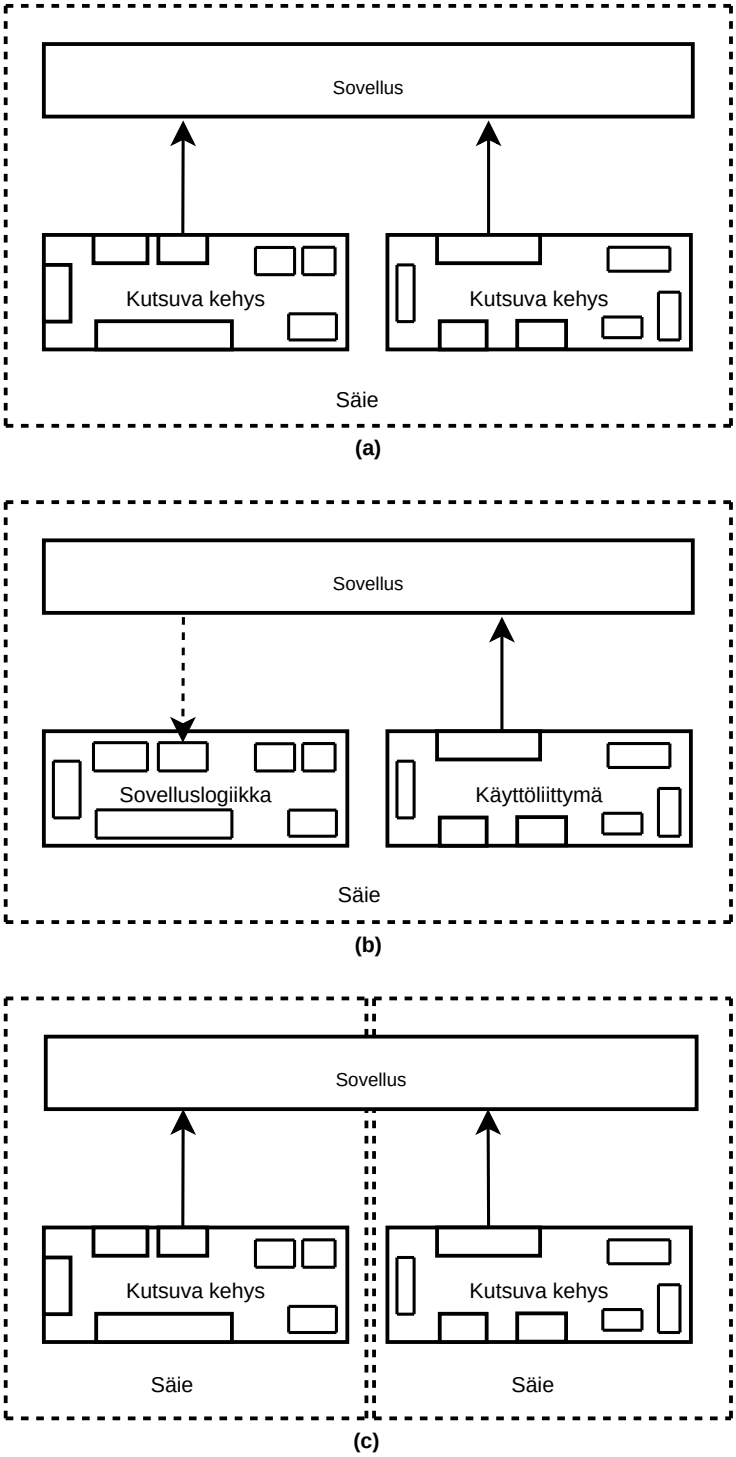
Kuvassa 3.3 on esitetty kehysten yhdistämistä ja kontrollin jakoa kehysten välillä. Kohdassa (a) on esitetty kahden kutsuvan kehyksen kontrollin jaon ristiriitainen lähtöasetelma. Kohdassa (b) kontrolli on jaettu tekemällä toisesta kehyksestä kutsuttava. Kohdassa (c) kontrolli on jaettu sijoittamalla molemmat kutsuvat kehykset omiin säikeisiinsä. Kohdasta (c) voidaan nähdä, että nyt sovelluskohtaisten osien toteutuksessa olisi otettava huomioon kahden säikeen samanaikainen pääsy sovelluskohtaisiin osiin.

3.2 Ohjelmistokehyksen suunnittelu

Ohjelmistokehyksen suunnittelu on haastavampaa kuin yksittäisen sovelluksen suunnittelu. Yksittäisen sovelluksen suunnitteluun tarkoitettujen oliosuunnittelumenetelmien lähtökohtana on tarkoin määritellyt vaatimukset täyttävä sovellus, jonka vaatimuksia voidaan tarkentaa käyttötapauksien avulla. Ohjelmistokehyksellä on kuitenkin periaatteessa rajaton määrä sovelluksia, joiden vaatimuksiin sen tulisi vastata, eivätkä kaikki kehyksen avulla kehitettävät sovellukset ole etukäteen tiedossa [49].

3.2.1 Käyttäjäroolit

Ohjelmistokehyksellä voidaan ajatella olevan eri rooleissa toimivia käyttäjiä. Kehyksen käyttäjiksi voidaan laskea itse kehystä kehittävät ohjelmistokehittäjät, kehykseen



Kuva 3.3: Kehysten kontrollien yhdistely.

lisättävien uusien ohjelmistorakenteiden ohjelmistokehittäjät, kehyksestä lopullisia ohjelmistokokonaisuuksia erikoistavat ohjelmistokehittäjät sekä erikoistettujen ohjelmistokokonaisuuksien *loppukäyttäjät (end user)* [20].

Eri rooleissa toimivat käyttäjät asettavat omat vaatimuksensa kehykselle, jotka on otettava huomioon kehystä suunniteltaessa [20]. Yksi käyttäjä voi toimia myös useassa eri roolissa. Esimerkiksi kehystä kehittävä ohjelmistokehittäjä voi kehittää myös lisärakenteita kehykseen, erikoistaa kehyksestä testausta varten ohjelmistokokonaisuuksia ja toimia testauksen aikana erikoistettujen ohjelmistokokonaisuuksien loppukäyttäjänä. Jos loppukäyttäjä voi määritellä asetuksia kehyksen ohjelmistorakenteille, hänen voidaan ajatella toimivan myös kehyksen erikoistajan roolissa.

3.2.2 Prosessi

Perinteiset ohjelmistokehitysmenetelmät soveltuvat huonosti ohjelmistokehysten suunnitteluun eikä kehysten suunnitteluun ole olemassa omia vakiintuneita menetelmiä [49, s. 206]. Ohjelmistokehyksen kehitys aloitetaankin usein keräämällä kehyksen kohdealueelle aikaisemmin toteutettujen sovelluksien yhteisiä ohjelmarakenteita yhteen. Näitä rakenteita yleistetään iteratiivisesti kunnes ne ovat tarpeeksi yleisellä tasolla, jotta ne vastaisivat kaikkiin kehyksen kohdealueen vaatimuksiin.

Tämän *takaisinmallinnukseen (reverse engineering)* perustuva menetelmä on pitkä ja huonosti hallittava prosessi [49, s. 208]. Lisäksi se ottaa huomioon vain aloitusvaiheessa kehykseen mukaan otettavien rakenteiden liittämiseen ja yhteistoimintaan liittyvät vaatimukset, eikä varaudu tarpeeksi hyvin kehykseen tulevaisuudessa lisättävien rakenteiden liittämiseen osaksi kehystä [20].

3.2.3 Dokumentointi

Ohjelmistokehyksen rakenteen ja käytön dokumentointi on tärkeä osa kehyksen suunnittelua. Kehyksen dokumentointi on suunnattava erityisesti kehykseen lisäosia kehittäville ja kehyksestä lopullisia ohjelmistokokonaisuuksia erikoistaville käyttäjille. Erityisesti kehyksen laajennoskohdat ja niiden käyttö on dokumentoitava hyvin, jotta käyttäjät oppisivat käyttämään kehystä oikein. Myös kehyksen yleinen toimintaperiaate ja käyttöohje on dokumentoitava. Kehyksen erikoistamisesta annetut esimerkkitapaukset ovat olennainen osa kehyksen dokumentointia [27].

Koottavien kehysten dokumentointi on helpompaa kuin muunneltavien kehysten. Koottavan kehyksen käyttäjän dokumentoinniksi riittää usein kehyksen yleisen toimintaperiaatteen kuvauksen lisäksi kehyksen laajennoskohtien ja niiden täydentämiseen käytettävien ohjelmarakenteiden palvelurajapintojen kuvaus. Muunneltavien kehysten

tapauksessa käyttäjän on kuitenkin tunnettava kehyksen rakenne ja laajennoskohtien toteutus tarkemmin [22]. Usein kehykset sisältävät sekä koottavien että muunneltavien kehysten piirteitä ja dokumentoinnin tarve kehyksen eri kohdille vaihtelee.

3.3 Ohjelmistokehysten etuja ja haasteita

Ohjelmistokehykset ovat usein suuria ja monimutkaisia järjestelmiä joiden käyttö voi olla vaikeaa [27]. Hyvin suunniteltu ja dokumentoitu kehys on kuitenkin tehokas apuväline ohjelmistojen kehityksessä. Ohjelmistokehykset sopivat erityisesti myös tuoterunkoarkkitehtuurien toteutukseen.

3.3.1 Etuja

Ohjelmistokehyksen avulla voidaan käyttää uudelleen hyväksi todettuja suunnitteluratkaisuja ja toteutuksia, mikä vähentää ohjelmistojen kehityskustannuksia ja parantaa ohjelmistojen laatua. Kehykset lisäävät ohjelmiston modulaarisuutta, uudelleenkäytettävyyttä ja laajennettavuutta [27]. Toisin kuin monet suunnitteluratkaisujen uudelleenkäyttömenetelmät, kehys on konkreettinen, ohjelmakoodina esitettävä menetelmä [51].

Kehyksen toteutus piilotetaan hyvin määriteltyjen rajapintojen alle, mikä mahdollistaa kehyksen toteutuksen muokkaamisen siten, että se ei vaikuta kehyksestä erikoistettuihin ohjelmistokokonaisuuksiin [27]. Kehys sisältää kohdealueella tarvittavat yleiset palvelut, joten kehyksen käyttäjän ei tarvitse kehittää ratkaisuja kaikkiin yksityiskohtiin, vaan käyttäjä voi keskittyä kohdealueen korkeamman tason ongelmiin. Kohdealueen yleisten palveluiden keskitetty ylläpito ja hallinta parantavat myös ohjelmistojen laatua.

Kehyksen laajennettavuus ja modularisuus helpottavat kehystä käyttävien sovellusten ylläpitoa ja päivitystä. Muutosten ja päivitysten vaikutukset voidaan lokalisoida kehyksen erikoistamisessa käytettyihin ohjelmarakenteisiin [27].

3.3.2 Haasteita

Ohjelmistokehyksen kehitys on haastavampaa kuin yksittäisen sovelluksen kehitys. Kehysten suunnitteluun, toteutukseen, dokumentointiin ja erikoistamiseen ei ole olemassa standardoituja menetelmiä siinä määrin kuin perinteiseen ohjelmistokehitykseen [27] [49].

Kehysten käyttöön voi olla monia erilaisia vaihtoehtoja, mikä voi tehdä kehyksen käytön opettelusta haastavaa [27]. Mitä monimutkaisempi kehys sitä enemmän vie ai-

kaa opetella sen käyttöä. Kehyksen käytön dokumentointi on erityisen tärkeää. Kehyksen käyttö voi pahimmassa tapauksessa hankaloittaa ja pitkittää ohjelmiston kehitysprosessia, jos käyttäjät eivät osaa käyttää kehystä riittävän hyvin. Lisäksi kehyksen väärinkäyttö voi rikkoa kehyksen avulla kehitettyjen ohjelmistokokonaisuuksien yhtenäisyyden ja ylläpidettävyyden.

Kehyksen testaus on hankalaa, koska siitä erikoistetuilla ohjelmistokokonaisuuksilla on periaatteessa rajoittamaton määrä erilaisia toimintaympäristöjä. Kehyksen sisältämät ohjelmarakenteet tulevat testatuiksi jokaisen erikoistuksen yhteydessä, mutta erikoistamattomien ohjelmarakenteiden kattava testaus on mahdotonta. Erikoistetun ohjelmistokokonaisuuden yhteydessä tehtävässä testauksessa voi olla hankalaa erottaa onko löydetty vika kehyksessä vai erikoistamiseen käytetyssä ohjelmakoodissa [27].

Schmidt ym. [26] toteavat myös, että teknisten seikkojen lisäksi myös kehystä kehittävän ja käyttävän organisaation on tuettava kehyksen käyttöä. Jos kehyksen käytölle ei ole todellista tilausta organisaation toiminta-alueella, kehyksen kohdealue ei ole riittävän haastava tai kehyksen uudelleenkäytettävien rakenteiden kehitystä ei tueta tarpeeksi, ohjelmistokehittäjät eivät käytä kehyksen uudelleenkäytettäviä rakenteita vaan kehittävät kaiken alusta asti uudelleen. Schmidt ym. kutsuvat tätä *”ei keksitty täällä”* -syndroomaksi *(”Not invented-here”).*

3.4 Komponenttipohjainen ohjelmistokehys

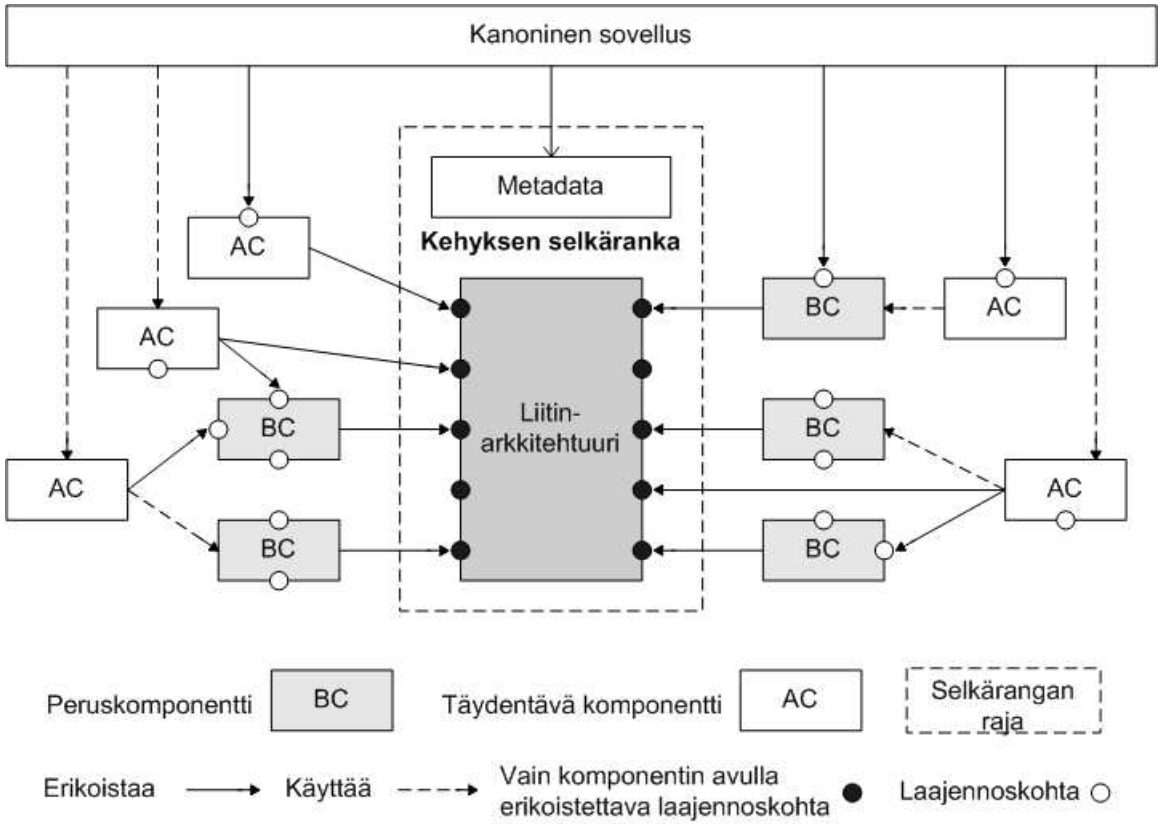
Oliokehyksissä rakennusyksikköinä toimivat luokat ja niistä luodut oliot. Parsons ym. [20] mukaan hienorakenteisten olioiden käyttö kehyksen rakennusyksikköinä rajoittaa kehyksen skaalautuvuutta ja laajennettavuutta. Olioiden kytkennät ympäristöönsä ovat löyhiä, mikä vaikeuttaa olioiden elinkaarien ja *metadatan* hallintaa kehyksen avulla. Parsons ym. esittävätkin, että oliokehys on koodikeskeinen rakenne, joka tarjoaa vain vähän joustavuutta kehykseen liitettyjen olioiden hallintaan ja joka soveltuu parhaiten hyvin rajatuille sekä vain vähän skaalautuvuutta vaativille kohdealueille.

Komponentit tarjoavat karkearakeisemman rakennusyksikön kehysten kehitykseen kuin oliot ja komponenttien käyttö kehyksen rakennusyksikköinä laajentaa kehyksen kohdealuetta [20]. Komponentti on itsenäinen ohjelmistorakenne, jonka toiminta kuvataan sille määriteltyjen rajapintojen avulla. Komponentille määritellään sen tarjoamien ja vaatimien palvelujen rajapinnat sekä tekninen rajapinta, joka määrittelee miten komponentti liitetään osaksi kehystä. Tekniseen rajapintaan kuuluu myös metadatan määrittely, jonka avulla komponentin asetuksia voidaan määritellä [20].

Komponenttipohjaiset kehykset ovat yleensä koottavia kehyksiä, joiden erikoistaminen tapahtuu kehykseen liitettävien komponenttien avulla [20]. Komponenttipohjai-

sen kehyksen sisältämä komponentti voi itsessään olla myös komponenttikehys, josta voidaan erikoistaa lopullisia komponentteja. Komponenttipohjainen kehys voidaankin nähdä esimerkkinä kehysten yhdistelystä. Kehykseen liitettävät komponentit voivat olla myös kolmannen osapuolen tekemiä niin sanottuja *COTS (Commercial off-the-shelf)* -komponentteja. Komponenttipohjainen ohjelmistokehys sisältää mekanismit komponenttien yhdistämiseksi osaksi kehystä sekä mekanismin komponenttien väliseen kommunikointiin.

Parsons ym. esittävät artikkelissaan [20] yleisen komponenttipohjaisen kehyksen arkkitehtuurimallin, jota voidaan hyödyntää komponenttipohjaisia kehyksiä suunniteltaessa. Parsonsin ym. mallissa kehykseen liitettävät komponentit voivat olla myös joko koottavia tai muunneltavia kehyksiä, joita muut komponentit tai kehystävä hyödyntävä sovellus voivat erikoistaa. Tämänkaltainen rakenne mahdollistaa kehyksen suuren muunneltavuuden siten, että kehyksen arkkitehtuuri ja laajennoskohdat pysyvät hallinnassa ja ne voidaan kuvata selkeästi kehyksen käyttäjälle [20]. Kuvassa 3.4 on esitetty Parsonsin ym. komponenttikehyksen arkkitehtuurimalli.



Kuva 3.4: Komponenttikehyksen arkkitehtuurimalli. [20]

Parsonsin ym. [20] malli sisältää kolme ydinelementtiä ja kolme avustavaa element-

tiä. Ydinelementtejä ovat *kehyksen selkäranka (framework backbone)*, *peruskomponentit* ja *täydentävät komponentit*. Selkäranka sisältää kaksi avustavaa elementtiä, *metadatan* ja *liitinarkkitehtuurin (connector architecture)*. Kolmas avustava elementti on kehystä hyödyntävä *kanoninen sovellus*.

Parsonsin ym. [20] mallin komponentit ovat löyhästi kytkettyjä rakenteita, joita voidaan kehittää itsenäisesti toisista komponenteista riippumattomina. Selkäranka on kehyksen toiminnallisuuden ydin, jonka avulla hallitaan itsenäisesti kehitettyjä komponentteja. Selkärangan tulisi toteuttaa kaikki komponenttien integroimiseen ja yhteentoimivuuteen vaadittavat mekanismit. Komponentit integroidaan kehykseen selkärangan liitinarkkitehtuurin laajennoskohtien avulla ja selkäranka toteuttaa kehykseen liitettyjen komponenttien väliseen kommunikointiin, tiedonvälitykseen ja synkronointiin liittyvät mekanismit. Peruskomponentit puolestaan toteuttavat kehyksen perustoiminnallisuuden ja liittyvät kehykseen selkärangan liitinarkkitehtuurin laajennoskohtien kautta.

Selkäranka on kehyksen muuttumaton elementti, joka toteuttaa kehyksen kohdealueen arkkitehtuurin ja sitä ei voida korvata toisella elementillä [20]. Kaikki muut mallin elementit voidaan vaihtaa toisiin elementteihin. Selkärankaan voidaan kuitenkin liittää peruskomponenttien lisäksi myös täydentäviä komponentteja, joiden avulla kehyksen toimintaa voidaan sopeuttaa kohdealueen vaatimuksiin. Täydentävät komponentit voivat käyttää sekä peruskomponenttien tarjoamia palveluja että erikoistaa peruskomponentteja. Kehystä käyttävä kanoninen sovellus voi puolestaan käyttää ja erikoistaa sekä peruskomponentteja että täydentäviä komponentteja.

Parsonsin ym. [20] arkkitehtuurimallia voidaan käyttää apuna edellisessä luvussa esitellyn tuoterunkomallin toteutusta suunniteltaessa. Parsonsin ym. mallin selkäranka, mahdollisesti yhdessä joidenkin peruskomponenttien kanssa, voitaisiin mieltää tuoterungon pakolliseksi komponentiksi eli ohjelmistoalustaksi, joka sisältyy jokaiseen tuoterungon avulla kehitettävään ohjelmistoon. Peruskomponenttien voitaisiin ajatella vastaavan tuoterungon vaihtoehtoisia komponentteja ja täydentävien komponenttien tuoterungon valinnaisia komponentteja.

4 Järjestelmän hajautus

Luvut 2 ja 3 käsittelivät tuoterunkoa sekä sen toteutusta ohjelmistokehysten avulla. Kun tuoterungon sovellusalue sijoittuu hajautettuun ympäristöön, tulee rungon kehityksessä vastaan uusia haasteita, joita tähänastinen tarkastelu ei ole tuonut esille. Tutkielman loppuosa esittelee vaatimuksia, joita hajautettu ympäristö tuoterungolle asettaa ja käsittelee tuoterungon toteutusta tällaisessa ympäristössä. Tässä luvussa esitellään hajautetun ympäristön ominaisuuksia, hajautetussa ympäristössä toimivalta järjestelmältä vaadittavia ominaisuuksia sekä järjestelmän hajautuksen toteutusta.

Tanenbaum ym. [42, luvut 2-8] määrittelevät seitsemän periaatetta, jotka on otettava huomioon hajautettuja järjestelmiä suunniteltaessa: *kommunikaatio, prosessit, nimeäminen, synkronointi, johdonmukaisuus ja toistaminen (consistency and replication), vikasietoisuus sekä turvallisuus*. Tanenbaumin ym. esittelemät periaatteet ovat kaikki tärkeitä osa-alueita hajautettujen järjestelmien kehityksessä ja antavat kattavan selvityksen ominaisuuksista, jotka on otettava huomioon hajautettuja järjestelmiä kehitettäessä. Tämän tutkielman tarkastelu ei kata kaikkia Tanenbaumin ym. esittelemien periaatteiden toteutusta, vaan keskittyy erityisesti järjestelmän rakenteen hajauttamisen kannalta oleellisimmiksi katsottaviin periaatteisiin. Muut periaatteet voidaan ajatella toteutettavan järjestelmään erillisten palvelujen avulla luvussa 5 esiteltävän *OMG (The Object Management Group) OMA (Object Management Architecture)* mallin mukaisesti.

Luku 4.1 esittelee ensin hajautetun järjestelmän määritelmän, joka auttaa ymmärtämään, mitä järjestelmän hajautuksella tässä yhteydessä tarkoitetaan. Luvuissa 4.2 ja 4.3 esitellään luettelomaisesti esitetyn määritelmän mukaisen hajautetun ympäristön ja järjestelmien ominaisuuksia. Lopuksi luvussa 4.4 käsitellään järjestelmän hajautuksen toteutusta ja esitellään hajautuksen toteutuksen osa-alueita.

4.1 Hajautetun järjestelmän määritelmä

Hajautetuille järjestelmille on olemassa useita erilaisia määritelmiä, joiden mukaiset järjestelmät muodostavat suuren joukon erilaisia järjestelmiä. Tässä yhteydessä ei yritetä tehdä kattavaa vertailua eri järjestelmistä vaan pyritään johtamaan erilaisten määritelmien pohjalta kuvaus, joka toimisi mahdollisimman hyvin tulevan tarkastelun apuna.

Tanenbaumin ym. [42] määritelmä kuvaa hajautetun järjestelmän joukoksi itsenäisiä tietokoneita, jotka muodostavat järjestelmän käyttäjälle vaikutelman yksittäisestä, yhtenäisestä järjestelmästä:

> *A distributed system is a collection of independent computers that appears to its users as a single coherent system.*

Määritelmän vaatimus itsenäisistä tietokoneista rajaa tarkastelun ulkopuolelle *tiukasti kytketyt (tightly coupled)* järjestelmät, kuten monen prosessorin *(multiprocessor)* ja koneen *(multicomputer) hajautetut käyttöjärjestelmät (distributed operating system)*. Yhtenäisen järjestelmän vaatimus puolestaan karsii pois *verkkokäyttöjärjestelmät (network operating systems)*.

Tanenbaumin ym. määritelmä ei kuitenkaan pidä sisällään vaatimusta järjestelmän eri osien yhteisestä tarkoituksesta tai tavoitteesta, toisin kuin Burnsin ym. [19] määritelmä joukosta itsenäisiä käsittelyelementtejä, jotka toimivat yhdessä yhteisen tavoitteen eteen:

> *A system of multiple autonomous processing elements, cooperating in a common purpose or achieve a common goal.*

Myös tässä määritelmässä itsenäisten elementtien voidaan katsoa karsivan määritelmän ulkopuolelle niin sanotut tiukasti kytketyt järjestelmät. Tämän lisäksi määritelmä karsii pois yleiskäyttöiset järjestelmät, kuten Internetin, joilla ei katsota olevan tiettyä yhteistä tavoitetta.

Tiukasti kytketyillä järjestelmillä on yhteinen *jaettu muisti (shared memory)*, joka voi olla toteutettu joko laitteiston avulla tai ohjelmallisesti. Jaetun muistin käytöstä seuraa, että järjestelmällä on myös yhteinen tila, joka on kaikkien järjestelmään kuuluvien osien tiedossa. *Löyhästi kytketyillä (loosely coupled)* järjestelmillä ei ole yhteistä tilaa, ja niiden toiminnan hallinta perustuu järjestelmän osien väliseen *viestinvälitykseen (message passing)* [42, s. 28]. Coulouriksen ym. [35] määritelmä hajautetulle järjestelmälle perustuu löyhästi kytkettyihin järjestelmiin:

> *One in which components located at networked computers communicate and coordinate their action only by passing messages.*

Näiden määritelmien pohjalta annetaan hajautetulle järjestelmälle tässä yhteydessä seuraava määritelmä: hajautettu järjestelmä koostuu joukosta komponentteja, jotka kommunikoivat keskenään viestinvälityksen avulla ja jotka yhdessä muodostavat käyttäjälle yhtenäisen, määrättyyn tavoitteeseen tähtäävän järjestelmän. Komponentilla

tarkoitetaan tässä yhteydessä lähinnä Burnsin ym. [19] määritelmän mukaisia itsenäisiä käsittelyelementtejä, erotuksena tämän tutkielman muissa luvuissa esiintyvästä varsinaisen ohjelmistokomponentin käsitteestä.

Tulevan tarkastelun tueksi tehdään edellä annettuun määritelmään vielä muutama tarkennus. Jokainen komponentti toimii omassa *suoritussäikeessään (thread of execution)*. Viestinvälitys tapahtuu komponenttien suoritussäikeiden välillä. Suoritussäikeitä voi sijaita sekä samalla prosessorilla että olla hajautettuna järjestelmän eri *solmuihin (node)* [21]. Hajautettu järjestelmä sisältää kaksi tai useampia solmuja, joissa jokaisessa voi olla useita komponenttien suoritussäikeitä. Samassa solmussa sijaitsevien suoritussäikeiden välinen viestinvälitys tapahtuu paikallisten kommunikointimenetelmien avulla ja eri solmujen välinen viestinvälitys tietoliikenneyhteyden välityksellä. Solmujen väliset tietoliikenneyhteydet voivat sisältää erilaisia fyysisiä tiedonsiirtoyhteyksiä ja protokollia.

4.2 Hajautetun järjestelmän piirteitä

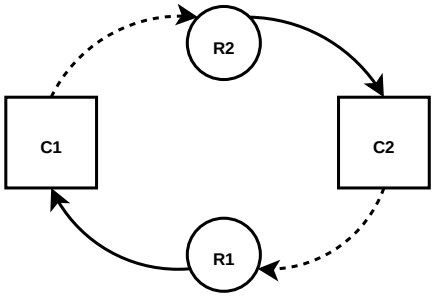
ISO:n *(International Organization for Standardization)* referenssimalli avoimelle hajautetulle prosessoinnille *(The Reference Model of Open Distributed Processing, RM-ODP)* [5] esittelee hajautetuille järjestelmille ominaisia piirteitä. Nämä piirteet ovat ominaisia edellä esitetyn määritelmän mukaisen järjestelmän luonteelle ja ne on otettava huomioon järjestelmän suunnittelussa ja toteutuksessa.

4.2.1 Samanaikaisuus

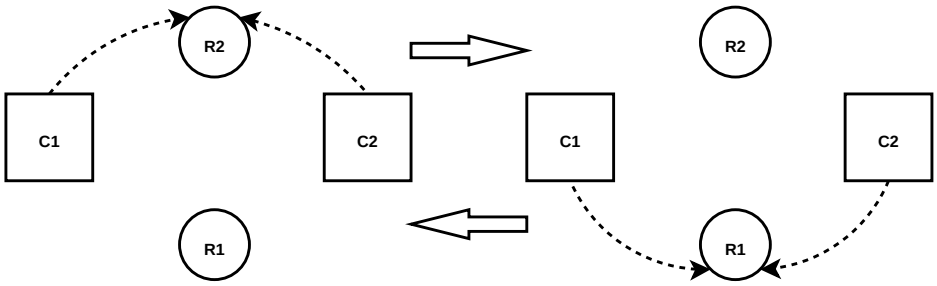
Hajautetut järjestelmän komponenttien suoritussäikeet voivat toimia *samanaikaisesti (concurrent)* muiden järjestelmän komponenttien suoritussäikeiden kanssa [5]. Järjestelmän on huolehdittava resursseihin pääsyn synkronoinnista samanaikaisesti toimivien säikeiden välillä. Tyypillisiä samanaikaisuudesta aiheutuvia ongelmatilanteita ovat *lukkiutumiset (deadlock)* ja *vauhkoontumiset (livelock)*.

Kuvassa 4.1 on esitetty järjestelmän lukkiutumistilanteen syntyä. Kuvan lähtötilanne on, että komponentilla C1 on hallussaan resurssi R1 ja komponentilla C2 resurssi R2. Jatkaakseen toimintaansa komponentti C1 tarvitsee resurssin R1 lisäksi resurssin R2, joten se jää odottamaan, että komponentti C2 vapauttaa resurssin R2. Samanaikaisesti komponentti C2 tarvitsee toimintansa jatkamiseen myös molemmat resurssit ja jää odottamaan, että C1 vapauttaa resurssin R1. Nyt molemmat komponentit odottavat toistensa varaamia resursseja ja järjestelmä on lukkiutunut.

Kuvassa 4.2 esitetään järjestelmän vauhkoontumiseen johtava tilanne. Komponentit



Kuva 4.1: Lukkiutuminen.



Kuva 4.2: Vauhkoontuminen.

yrittävät varata samanaikaisesti ensin resurssia R2. Molemmat komponentit havaitsevat, että resurssi on varattuna ja yrittävät seuraavaksi varata resurssia R1. Sama tilanne toistuu ja muodostuu silmukka, jossa resursseja vapautetaan ja varataan, mutta järjestelmän toiminta ei etene. Tätä kutsutaan järjestelmän vauhkoontumiseksi.

4.2.2 Yhteisen tilan ja kellon puute

Hajautetun järjestelmän komponentit voivat sijaita eri solmuissa, joita yhdistää tietoliikenneyhteys. Tästä seuraa, että järjestelmällä ei ole yhteistä *globaalia kelloa* eikä *globaalia tilaa* [5]. Yhteisen tilan ja yhteiseen kelloon perustuvan koordinoinnin sijaan hajautetun järjestelmän komponentit voivat toimia samanaikaisesti ja niiden toimintaa koordinoidaan viestinvälityksen avulla. Tämä tarkoittaa, että järjestelmän tilan muutoksien ei voida olettaa tapahtuvan tietyllä tarkkaan määritellyllä hetkellä.

4.2.3 Paikalliset viat

Hajautetun järjestelmän komponentit toimivat itsenäisesti eristyksessä toisistaan. Tämä tarkoittaa, että komponenttien toiminnan virheet tapahtuvat myös itsenäisesti. Muut järjestelmän osat eivät siis saa automaattisesti tietoa yksittäisessä komponentissa sattuneista virhetilanteista, eikä yksittäinen komponentti muiden järjestelmän osien

virheistä *(partial failures)* [5]. Järjestelmän solmuja yhdistävien tietoliikenneyhteyksien virhetilanteet voivat myös jakaa järjestelmän osiin, jotka eivät voi kommunikoida toisten osien kanssa.

4.2.4 Heterogeenisyys

Hajautetun järjestelmän toimintaympäristö on heterogeeninen [5]. Se voi sisältää eri tekniikoilla toteutettuja komponentteja ja komponentit voivat toimia erilaisissa ympäristöissä. Komponentit voidaan toteuttaa eri ohjelmointikielillä ja ne voivat hyödyntää eri käyttöjärjestelmien, ohjelmistokehysten tai kirjastojen tarjoamia palveluja. Komponenttien toimintaympäristö voi sisältää eri käyttöjärjestelmien lisäksi erilaisia laitteistokokoonpanoja sekä erilaisia tietoliikenneyhteyksiä ja protokollia. Eri tekniikoiden ja laitteistojen evoluutio lisää myös ympäristön heterogeenisyyttä.

4.3 Hajautetulta järjestelmältä vaadittavia ominaisuuksia

Hajautetulla järjestelmällä on oltava tiettyjä ominaisuuksia, jotta se voisi toimia hajautetussa ympäristössä ja täyttää edellä esitetyn määritelmän mukaisen järjestelmän vaatimukset.

4.3.1 Läpinäkyvyys

Hajautettu järjestelmä toimii heterogeenisessä ympäristössä. Jotta määritelmä käyttäjälle näkyvästä yhtenäisestä järjestelmästä täyttyisi, hajautetun järjestelmän on *peitettävä (mask)* ympäristön heterogeenisyys ja toteutuksen hajautus järjestelmän käyttäjältä [5]. Toisin sanoen järjestelmän on tehtävä toteutuksen *hajautus* käyttäjälle *läpinäkyväksi (distribution transparency)* [5].

Läpinäkyvyydellä on eri tasoja ja hajautetun järjestelmän on tarjottava näitä tasoja vastaavia läpinäkyvyyspalveluja. ISO RM-ODP [5] määrittää läpinäkyvyydelle kahdeksan eri tasoa: *saanti- (access), häiriö- (failure), sijainti- (location), muutto- (migration), sijoitettavuus- (relocation), monistus- (replication), pysyvyys- (persistence)* ja *tapahtumaläpinäkyvyys (transaction transparency).*

Saantiläpinäkyvyys tarkoittaa, että järjestelmä tarjoaa komponenteille yhtenäisen tavan kommunikoida keskenään peittämällä ympäristön heterogeenisyyden yhtenäisten kutsurajapintojen ja tiedonesitystapojen alle. Häiriöläpinäkyvyys toteutetaan peittämällä itsenäisissä komponenteissa tapahtuvat virheet muilta komponenteilta ja tekemällä komponenteista sellaisia, että ne toipuvat itse omista virheistään.

Sijaintiläpinäkyvyys piilottaa komponenttien fyysisen sijainnin ja mahdollistaa kommunikoinnin komponenttien kanssa tietämättä niiden varsinaista sijaintia. Järjestelmän komponenttien sijaintia on myös voitava muuttaa ilman, että se vaikuttaa tapaan, jolla niiden kanssa kommunikoidaan. Tätä kutsutaan muuttoläpinäkyvyydeksi. Sijoitettavuusläpinäkyvyydellä puolestaan tarkoitetaan, että tietyn rajapinnan toteutuksen sijaintia voidaan vaihtaa keskeyttämättä rajapintaa käyttävien komponenttien toimintaa.

Hajautetussa järjestelmässä usea komponentti voi tarjota samaa palvelua yhteisen rajapinnan kautta. Monistusläpinäkyvyys peittää saman rajapinnan toteuttavien komponenttien lukumäärän palvelun käyttäjältä. Pysyvyysläpinäkyvyys tarkoittaa, että järjestelmän komponentti ja sen tila voidaan tallentaa pysyvään muistiin ja aktivoida komponentti sen palveluja kutsuttaessa siten, että kutsujan näkökulmasta komponentti on koko ajan aktiivinen.

Järjestelmän sisäisiä tapahtumia kuten tietyn resurssin käyttöä on koordinoitava eli huolehdittava tapahtumien vuorotuksesta, monitoroinnista ja tapahtumaan liittyvien osapuolien käyttöönotosta. Tapahtumien koordinointi on hankalaa ja vaatii omat mekanisminsa järjestelmään. Tapahtumaläpinäkyvyys piilottaa tapahtumien koordinoinnin vaatimat toimenpiteet ja mekanismit järjestelmän käyttäjältä.

Hajautettua järjestelmää suunniteltaessa on päätettävä *läpinäkyvyyden aste (degree of transparency)*, jota järjestelmä tarjoaa [42, s. 7]. Järjestelmän ei ole aina järkevää yrittää toteuttaa kaikkia läpinäkyvyyden tasoja. Läpinäkyvyyden astetta päätettäessä on otettava huomioon järjestelmän käyttäjän tarvitsemat tiedot järjestelmän toiminnasta. Jos esimerkiksi käyttäjän antama käskyn suoritus on hidasta, koska suorituksessa käytettävä komponentti sijaitsee ruuhkaisen tietoliikenneyhteyden päässä, voi olla järkevää antaa käyttäjän tietoon suorituksen hitauden syy.

4.3.2 Avoimuus

Hajautetun järjestelmän on oltava *avoin (open)* [5]. Tanenbaum ym. [42] määrittelevät avoimen hajautetun järjestelmän järjestelmäksi, jonka tarjoamat palvelut määritetään standardin mukaisten sääntöjen avulla annettujen palvelun syntaksin ja toiminnan kuvausten perusteella:

> *An open distributed system is a system that offers services according to standard rules that describe the syntax and semantics of those services.*

Hajautetun järjestelmän komponenttien tarjoamat palvelut voidaan määrittää komponenttien noudattamien rajapintojen kautta ja palveluiden kutsumuodot eli syntaksi *rajapinnan kuvauskielien (interface definition language)* avulla. Syntaksin lisäksi on

annettava palvelun toiminnan kuvaus eli kuvaus siitä, mitä palvelu tekee. Palvelun toiminnan kuvaus annetaan yleensä luonnollisella kielellä [42, s. 8].

Rajapinta erottaa palvelun kuvauksen ja toteutuksen toisistaan. Hyvin määritellyt rajapinnat mahdollistavat järjestelmän komponenttien välisen yhteistyön ja järjestelmän toiminnallisuuden muuntelun. Hyvä rajapinnan määrittely on täydellinen ja neutraali [42, s. 8]. Täydellisyys tarkoittaa, että kaikki rajapinnan toteuttamisessa tarvittavat tiedot on määritelty. Jos kaikkia tietoja ei ole määritelty, rajapinnan toteuttajan on lisättävä rajapintaan omia ratkaisujaan rajapinnan toteuttamiseksi, mikä rikkoo saman rajapinnan toteutuksien yhtenäisyyden. Neutraali rajapinnan määrittely ei ota kantaa rajapinnan toteutukseen, mikä mahdollistaa erilaiset rajapinnan toteutukset.

Järjestelmän tekniset rajapinnat määrittelevät, miten komponentit liitetään osaksi järjestelmää ja mahdollistavat erilaisten, uusia palveluja tuottavien komponenttien liittämisen järjestelmään. Avoimissa järjestelmissä määrätyn teknisen rajapinnan toteuttava komponentti voidaan siirtää järjestelmän sisällä solmusta toiseen ja myös samaa teknistä rajapintaa noudattavaan toiseen järjestelmään [42, s. 9].

4.3.3 Skaalautuvuus

Hajautetun järjestelmän on oltava skaalautuva [5]. Neuman [52] määrittelee skaalautuvuudelle kolme eri ulottuvuutta: määrällinen, maantieteellinen ja hallinnollinen ulottuvuus. Määrällinen ulottuvuus käsittää järjestelmän sisältämien solmujen, komponenttien, komponenttien tarjoamien palveluiden ja järjestelmän käyttäjien lukumäärän. Maantieteellisellä ulottuvuudella tarkoitetaan järjestelmän komponenttien välistä etäisyyttä ja hallinnollisella ulottuvuudella järjestelmän eri osien hallintaan osallistuvien organisaatioiden lukumäärää. Neuman esittelee määrittelemiensä ulottuvuuksien avulla useita skaalautuvuuden vaikutuksia hajautettuihin järjestelmiin.

Määrällinen ja maantieteellinen skaalautuminen vaikuttaa järjestelmän luotettavuuteen ja suorituskykyyn [52]. Kun järjestelmän komponenttien ja käyttäjien lukumäärä kasvaa, kasvaa myös todennäköisyys, että joku järjestelmän solmuista tai solmun tarjoamista palveluista on poissa käytöstä. Järjestelmän solmujen etäisyyden kasvaessa kasvaa myös todennäköisyys, että kaikki solmut eivät pysty kommunikoimaan keskenään. Järjestelmän palvelujen lisääntynyt käyttäjämäärä puolestaan lisää palveluja tarjoavien komponenttien kuormitusta ja kasvaneet etäisyydet kommunikoinnin viivettä, mikä vaikuttaa palvelujen vasteaikoihin.

Hallinnollinen skaalautuminen lisää muutosten määrää järjestelmässä ja vaikeuttaa järjestelmän muutosten hallintaa [52]. Neumanin mukaan skaalautuminen vaikuttaa myös järjestelmän heterogeenisyyteen, koska järjestelmän skaalautuessa järjestelmän

solmujen ja komponenttien heterogeenisyys todennäköisesti lisääntyy.

4.4 Hajautuksen toteutus

Edellä muodostettiin yleinen käsitys hajautetuista järjestelmistä, niiden luonteesta ja ominaisuuksista. Tässä luvussa tarkastellaan annetun määritelmän mukaisen hajautetun järjestelmän hajautuksen toteutusta yleisellä tasolla. Lähemmin hajautuksen toteutusta ja esimerkkejä toteutuksista tarkastellaan väliohjelmistojen käsittelyn yhteydessä luvussa 5.

Hajautetussa järjestelmässä järjestelmän toiminnallisuus ositetaan itsenäisiin komponentteihin, jotka toimivat omissa suoritussäikeissään. Luku 4.4.1 esittelee tapoja, joiden mukaisesti järjestelmä voi jakaa suoritussäikeet komponenteille. Kun järjestelmän toiminnallisuus on hajautettu komponentteihin, järjestelmän on tarjottava menetelmät komponenttien suoritussäikeiden väliseen viestinvälitykseen eli komponenttien väliseen kommunikaatioon. Luvussa 4.4.2 käsitellään eri kommunikaatiotapoja Tanenbaumin ym. [42] kommunikaatiotapojen luokittelun avulla.

Järjestelmän arkkitehtuuri määrittelee, miten järjestelmän rakenne ositetaan ja miten järjestelmän eri osat liittyvät toisiinsa siten, että järjestelmän tavoite saavutetaan. Luvussa 4.4.3 käsitellään asiakas-palvelin arkkitehtuurin mukaista järjestelmän rakenteen hajautusta, jota tarkennetaan hajautetun olion käsitteellä luvussa 4.4.4. Lopuksi luvuissa 4.4.5 ja 4.4.6 esitetään lisäksi kaksi muuta järjestelmän hajautuksen arkkitehtuurimallia: viestijonoihin perustuva ja julkaise-tilaa malli.

4.4.1 Suoritussäikeet

Luvussa 4.1 annetussa hajautetun järjestelmän määritelmässä sanottiin jokaisen komponentin toimivan omassa suoritussäikeessään. Komponenttien sijoittamiseen omiin suoritussäikeisiinsä on olemassa erilaisia menetelmiä. Tässä yhteydessä ei käsitellä itse säikeiden toteutusta käyttöjärjestelmissä eikä niiden käytön etuja tai haittoja. Säikeiden toteutustapa riippuu käyttöjärjestelmästä ja niiden käyttöön ohjelmistokehityksessä on olemassa erilaisia kirjastoja. Lähteissä [47] [48] ja [33] annetaan hyvä, säikeiden ohjelmointiin painottuva, selvitys eri käyttöjärjestelmien säiemalleista ja käsitellään erityisesti säikeiden ohjelmointia käyttöjärjestelmäriippumattoman *ACE* ohjelmistokehyksen avulla. Säikeiden toteutusta UNIX käyttöjärjestelmissä käsitellään esimerkiksi lähteessä [53] ja säikeiden ohjelmointia POSIX kirjaston avulla lähteessä [10].

Komponenttien suoritussäikeiden toteutukseen voidaan ajatella olevan kaksi lähestymistapaa. Jos hajautetun järjestelmän ohjelmoinnissa ei käytetä *säikeistystä (mul-*

tithreading), jokainen komponentti on sijoitettava omaan prosessiinsa. Komponentin suoritussäikeenä toimii tällöin prosessin *pääsäie (main thread of execution)* ja komponentin isäntä voidaan ajatella olevan se järjestelmän solmu, jossa prosessi sijaitsee. Säikeistystä käytettäessä sen sijaan yksi prosessi voi sisältää useita säikeitä, jotka toimivat komponenttien suoritussäikeinä. Tässä tapauksessa komponentin isäntä on se prosessi, johon komponentin suoritussäie kuuluu.

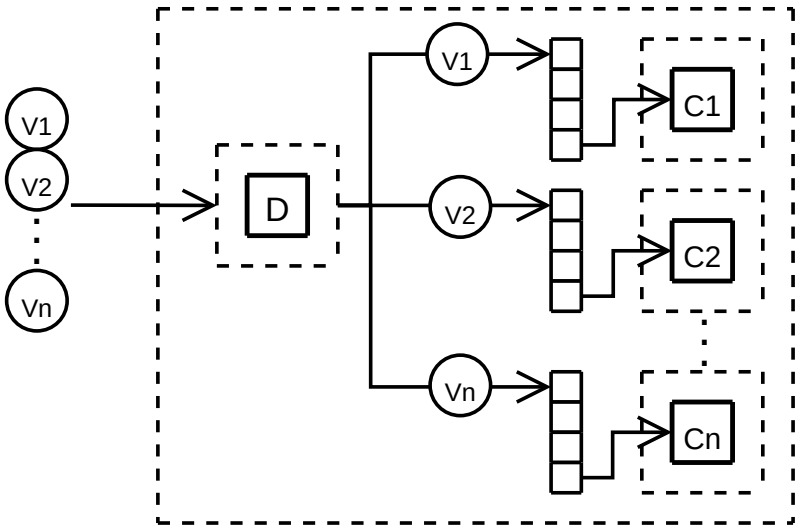
Hajautulle järjestelmälle luvussa 4.1 annetun määritelmän mukaan jokaisella komponentilla tulisi siis olla oma suoritussäie. Määritelmällä tehdään ero komponenttien ja sellaisten ohjelmarakenteiden välillä, jotka toimivat samaan säikeen sisällä. Tämä ei kuitenkaan tarkoita, että komponentille pitäisi olla koko ajan varattuna oma suoritussäie. Suoritussäie voidaan esimerkiksi osoittaa komponentille ainoastaan silloin kun komponentti sitä tarvitsee eli silloin kun sillä on jotain prosessoitavaa.

Komponentin isäntä vastaa suoritussäikeen tarjoamisesta komponentille. Suoritussäikeiden jakoa voidaan tarkastella komponenttien *päätepisteiden (endpoint)* [42, s. 149] avulla. Päätepiste on se piste, johon komponentille osoitetut viestit lähetetään. Isäntä voi esimerkiksi osoittaa jokaiselle komponentille oman suoritussäikeen ja päätepisteen, jonka kautta komponentin viestit kulkevat ilman isännän puuttumista viestinvälitykseen. Tälläiseen mallin toteutus voidaan tässä yhteydessä katsoa olevan triviaalia. Toinen mielenkiintoisempi vaihtoehto on, että kaikki komponenteille osoitetut viestit lähetetään komponenttien isännälle määriteltyyn päätepisteeseen. Yksi suoritussäie hallinnoi tätä päätepistettä ja välittää vastaanottamansa viestit niiden oikeiden vastaanottajien käsiteltäväksi.

Komponenttien isännälle määritelty komponenttien yhteinen päätepiste mahdollistaa erilaisten mallien käytön komponenttien suoritussäikeiden jaossa. *Tilaukseen perustuvassa* säikeiden luontimallissa *(on-demand spawning)* isäntä luo komponentille viestin käsittelyyn uuden suoritussäikeen (prosessin tai säikeen) joka kerta kun se vastaanottaa komponentille osoitetun viestin [47, s. 113]. *Innokkaaseen* säikeiden luontiin perustuvassa mallissa *(eager spawning)* isäntä luo joukon prosesseja tai säikeitä valmiiksi *varantoon (process pool, thread pool)* ja osoittaa viestiä vastaanottaessaan jonkin strategian mukaan tästä varannosta suoritussäikeen komponentille [47, s. 112].

Tilaukseen perustuvassa ja innokkaaseen säikeiden luontiin perustuvassa mallissa komponenteilla ei ole koko ajan käytössään omaa suoritussäiettä. Sen sijaan komponentille voi olla osoitettu samanaikaisesti useita suoritussäikeitä ja se voi käsitellä samanaikaisesti useita viestejä. Tämä tarkoittaa, että komponentin yhteinen tila on oltava suojattu säikeiden yhtäaikaiselta pääsyltä eli komponentin yhteiseen tilaan vaikuttavat operaatiot on synkronoitava, mikä voi monimutkaistaa komponentin toteutusta huomattavasti.

Usein käytetty ratkaisu suoritussäikeiden jaossa perustuu niin sanotun *vuorottajan (dispatcher)* käyttöön [42, s. 143]. Tässä mallissa isäntä on osoittanut jokaiselle hallinnoimalleen komponentille oman suoritussäikeen. Isäntä sisältää lisäksi yhden säikeen, joka vastaanottaa kaikki komponenteille osoitetut viestit ja välittää ne oikeille vastaanottajille. Mallissa komponentille on koko ajan siis osoitettu vain yksi suoritussäie, joten komponentin tilaa ei tarvitse suojata säikeiden samanaikaiselta käsittelyltä.



Kuva 4.3: Vuorottaja.

Kuvassa 4.3 esitetään vuorottajan käyttöä komponenttien suoritussäikeiden jakamisessa. Kuvassa komponenttien isäntänä toimii yksi prosessi, joka on osoittanut jokaiselle komponentille oman suoritussäikeen. Prosessin pääsäie eli vuorottaja (D) vastaanottaa kaikki komponenteille (C) osoitetut viestit (V) ja välittää ne komponenttien suoritussäikeiden käsiteltäväksi. Prosessien ja säikeiden rajoja on kuvattu katkoviivalla. Komponentilla voi olla samanaikaisesti käsiteltävänä vain yksi viesti. Komponentille aiemman viestin käsittelyn aikana välitettävät viestit voidaan tallentaa komponentin käsittelyjonoon, josta se voi myöhemmin ottaa viestit käsittelyyn.

4.4.2 Komponenttien välinen kommunikointi

Hajautun järjestelmän eri solmuissa sijaitsevilla komponenteilla ei ole jaettua muistia, minkä vuoksi komponenttien suoritussäikeiden tai prosessien välinen kommunikaatio *(interprocess communication)* on toteutettava tietoliikenneyhteyden kautta tapahtuvan viestinvälityksen avulla [42, s. 57]. Useita komponentteja sisältävän järjestelmän kommunikaation toteuttaminen tietoliikenneprotokollien tarjoamien matalan abstraktiotason mekanismien avulla on hyvin työlästä, minkä vuoksi hajautetun järjestelmän

on tarjottava korkeamman tason ratkaisu komponenttien väliseen kommunikointiin [42, s. 57].

Tanenbaum ym. [42, s. 99–103] luokittelevat hajautetun järjestelmän komponenttien välisen viestinvälityksen neljän eri tekijän perusteella. Viestinvälitys voidaan jakaa *pysyvään (persistent)* ja *tilapäiseen (transient)* kommunikaatioon, minkä lisäksi viestinvälitys voi olla joko *synkronista* tai *asynkronista*.

Pysyvässä kommunikaatiossa järjestelmä säilyttää lähetetyn viestin kunnes viesti on toimitettu vastaanottajalle [42, s. 100]. Järjestelmän on pystyttävä siis tallentamaan viesti viestinvälityksestä vastaavissa järjestelmän solmuissa, jos viestiä ei pystytä heti toimittamaan eteenpäin. Viestin lähettäjän ei ole välttämätöntä jäädä odottamaan viestin perille pääsyä, eikä viestin vastaanottajan suoritussäikeen ole välttämättä oltava aktiivinen viestiä lähetettäessä.

Tilapäisessä kommunikaatiossa järjestelmän viestinvälityksestä vastaavat solmut yrittävät toimittaa lähetetyn viestin seuraavalle solmulle ja jos viestin toimitus seuraavalle solmulle epäonnistuu, järjestelmä hylkää viestin [42, s. 101]. Tämä tarkoittaa, että kaikkien viestinvälitykseen osallistuvien solmujen sekä viestin vastaanottajan suoritussäikeiden on oltava aktiivisia viestiä lähetettäessä.

Synkronisessa viestinvälityksessä viestin lähettäjä ei jatka viestin lähetettyään omaa toimintaansa ennen kuin viesti on tallennettu viestin vastaanottajan solmussa tai toimitettu viestin vastaanottajalle [42, s. 101]. Asynkronisessa viestinvälityksessä viestin lähettäjä lähettää viestin ja jatkaa toimintaansa heti lähetyksen jälkeen [42, s. 101]. Lähetetty viesti tallennetaan järjestelmään ja viestin lähettäjä voi jatkaa toimintaansa samanaikaisesti kun viestiä toimitetaan perille.

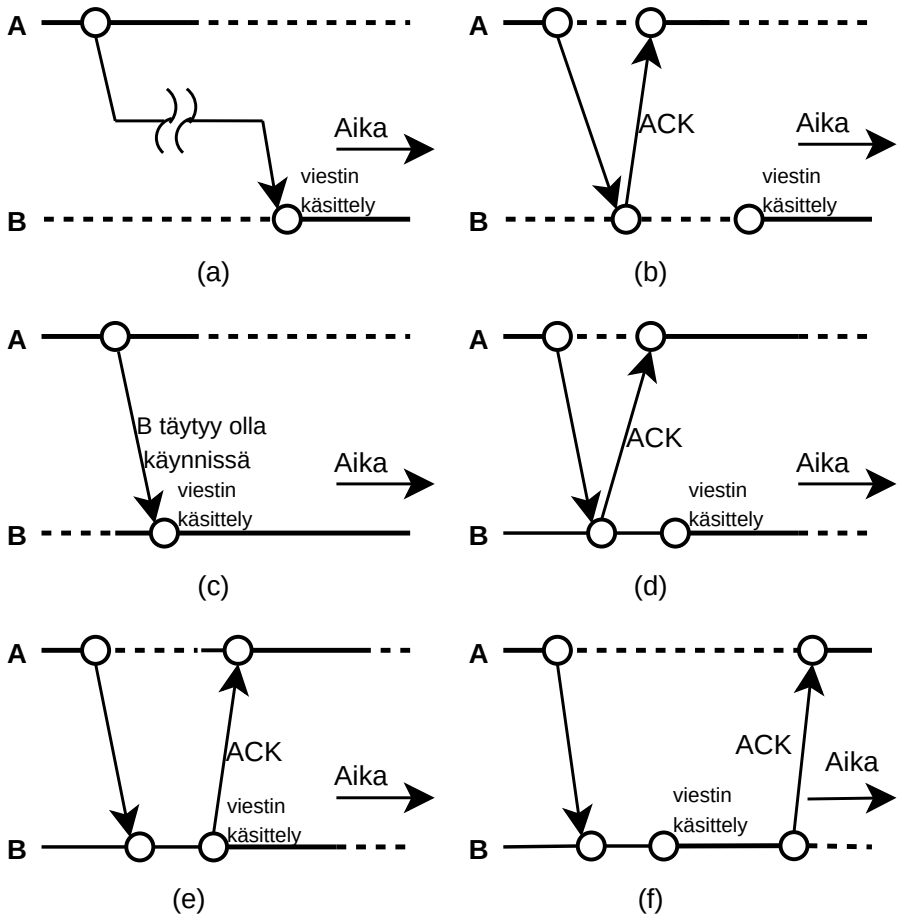
Tanenbaum ym. [42, s. 103] määrittelevät viestinvälityksen synkronisaation ja pysyvyyden perusteella kuusi eri luokkaa kommunikaatiolle: *pysyvä asynkroninen, pysyvä synkroninen, tilapäinen asynkroninen, viestin vastaanottoon perustuva tilapäinen synkroninen, viestin toimitukseen perustuva tilapäinen synkroninen ja vasteeseen perustuva tilapäinen synkroninen kommunikaatio.*

Kuvassa 4.4 kaksi komponenttia A ja B kommunikoivat keskenään viestinvälityksen avulla siten, että A toimii viestin lähettäjänä ja B viestin vastaanottajana. Komponentit A ja B toimivat omissa säikeissään ja järjestelmä voi joissain tapauksissa tallentaa lähetettyjä viestejä. Kohdissa (a) ja (b) esitetään pysyvää kommunikaatiota ja kohdissa (c)–(f) tilapäistä kommunikaatiota.

Kohdan (a) pysyvässä asynronisessa kommunikaatiossa A lähettää viestin B:lle, joka ei ole aktiivinen lähetyshetkellä. A jatkaa toimintaansa heti viestin lähetettyään ja viesti tallennetaan järjestelmään. B aloittaa viestin käsittelyn aktivoitumisensa jälkeen. Kohdassa (b) esitetään esimerkki pysyvästä synkronisesta kommunikoinnista. A lähet-

tää viestin ja jää odottamaan kunnes viesti on tallennettu järjestelmään. Saatuaan kuittauksen viestin tallennuksesta A jatkaa toimintaansa.

Kohdassa (c) kuvataan tilapäistä asynronista kommunikointia. A lähettää viestin B:lle ja jatkaa toimintaansa. Viesti voidaan lähettää ainoastaan, jos B on aktiivinen. Kohta (d) on esimerkki viestin vastaanottoon perustuvasta tilapäisestä synkronisesta kommunikoinnista. A lähettää viestin B:lle ja jää odottamaan vastausta. B on aktiivinen, mutta ei ehdi heti viestin vastaanottohetkellä käsitellä viestiä. B lähettää A:lle kuitenkin kuittauksen vastaanotetusta viestistä ja A pääsee jatkamaan toimintaansa, vaikka B käsitteleekin viestin vasta myöhemmin. Kohdan (e) viestin toimitukseen perustuvassa tilapäisessä synkronisessa kommunikaatiossa on sama lähtötilanne kuin edellä, mutta nyt B lähettää kuittauksen A:lle vasta aloittaessaan viestin käsittelyn. Kohdan (f) vasteeseen perustuvassa pysyvässä kommunikaatiossa B lähettää kuittauksen A:lle vasta käsiteltyään koko viestin.



Kuva 4.4: Hajautetun järjestelmän kommunikaatiotapoja. [42]

4.4.3 Asiakas-palvelin arkkitehtuuri

Asiakas-palvelin arkkitehtuuri (client-server architecture) oli tärkeä askel kohti tässäkin tutkielmassa annetun määritelmän mukaisia hajautettuja järjestelmiä. Uusien laitteistojen (PC) ja verkkostandardien (Ethernet) kehittyminen mahdollisti suurien *keskuskoneissa (mainframe)* ajettavien sovelluksien hajauttamisen usealle samassa lähiverkossa sijaitsevalle PC-koneelle [25]. Asiakas-palvelin arkkitehtuuri puolestaan tarjosi menetelmän keskuskoneiden monoliittisten sovellusten rakenteiden hajauttamiseksi osiin, jotka voidaan sijoittaa eri koneille.

Asiakas-palvelin arkkitehtuurissa palvelin on järjestelmän osa, joka toteuttaa tietyn palvelun. Asiakas on järjestelmän osa, joka käyttää palvelimen tarjoamaa palvelua. Asiakas käyttää palvelimen tarjoamaa palvelua lähettämällä palvelimelle *pyynnön (request)* ja palvelin vastaa pyyntöön lähettämällä asiakkaan pyyntöön *vastauksen (reply)* [42, s. 43]. Hajautetussa järjestelmässä asiakas ja palvelin voivat sijaita joko samassa järjestelmän solmussa tai eri solmuissa.

Järjestelmän toiminnan hajautus asiakkaisiin ja palvelimiin voidaan ajatella tapahtuvan vertikaalisesti tai horisontaalisesti [42, s. 50–53]. Vertikaalisella hajautuksella tarkoitetaan toiminnallisuuden hajautusta järjestelmän sisältämien loogisten kerrosten perusteella. Järjestelmän käyttäjälle tarjoamat palvelut voidaan jakaa kolmeen loogiseen kerrokseen: datan esittämiseen, datan käsittelyyn ja datan varastointiin liittyviä palveluja sisältävät kerrokset [42, s. 46–50].

Kaksitasoisessa arkkitehtuurissa (two-tiered architecture) eri kerroksiin sijoittuvat palvelut jaetaan kahteen tasoon asiakkaan ja palvelimen kesken [42, s. 51]. Asiakas voi vastata esimerkiksi datan esittämisestä huolehtivasta käyttöliittymästä sekä osittaisesta datan käsittelystä ja palvelin datan käsittelystä sekä varastoinnista. *Monitasoisessa arkkitehtuurissa (multitiered architecture)* kerrosten palvelut jaetaan kolmeen tai useampaan tasoon [42, s. 51]. Palvelin voidaan esimerkiksi jakaa kahteen eri tasoon, jotka vastaavat datan käsittelystä ja datan varastoinnista.

Järjestelmän tietyn tason palveluiden toteutus voidaan myös hajauttaa järjestelmän eri osiin, joista jokainen vastaa osittain palveluiden toteutuksesta. Tätä kutsutaan horisontaaliseksi hajautukseksi [42, s. 52]. Horisontaalinen hajautuksen avulla voidaan lisätä järjestelmän skaalautuvuutta ja parantaa järjestelmän suorituskykyä.

Järjestelmän osittaminen asiakkaisiin ja palvelimiin ei ole aina yksiselitteistä. Asiakkaat voivat usein toimia myös palvelimen roolissa ja palvelimet asiakkaan roolissa. Erityisesti *vertaiskäsittelyyn (peer-to-peer computing)* perustuvassa arkkitehtuurissa järjestelmän osia ei jaeta asiakkaisiin ja palvelimiin, vaan järjestelmä koostuu keskenään vertaisista osista, joista jokainen voi toimia samanaikaisesti sekä asiakkaan että palve-

limen roolissa [50].

Vertaiskäsittelyn avulla voidaan välttää puhtaan asiakas-palvelin arkkitehtuurin palvelimien ympärille muodustuvia prosessoinnin ja verkkoliikenteen pullonkaulakohtia [50]. Vertaiskäsittelyyn perustuva järjestelmä ei myöskään sisällä *yksittäistä vikaantumispistettä (single point of failure)*, joka voisi keskeyttää koko järjestelmän toiminnan [50]. Vertaiskäsittelyssä puhtaan asiakas-palvelin arkkitehtuurin roolijako kuitenkin häviää ja järjestelmän on toteutettava yhteinen koordinaatioprotokolla, jonka avulla hallitaan järjestelmän osien välistä yhteistyötä. Tällaisen hajautetun koordinaatioprotokollan kehitykseen on monia eri malleja, joiden käsittely tässä yhteydessä sivuutetaan. Erilaisia malleja on käsitelty esimerkiksi lähteessä [50].

4.4.4 Hajautetut oliot

Asiakas-palvelin arkkitehtuuri hajautti massiiviset keskuskonesovellukset osiin, mikä oli suuri askel kohti hajautettuja järjestelmiä. Usein tulos oli kuitenkin vain yhden massiivisen sovelluksen jakaantuminen kahdeksi massiiviseksi sovellukseksi (asiakas ja palvelin) [25, s. 15]. *Hajautetut oliot (distributed object)* mahdollistavat sekä asiakkaan että palvelimen jakamisen pienempiin itsenäisiin osiin, joita yhdistelemällä voidaan muodostaa haluttuja kokonaisuuksia [25, s. 15].

Kuten perinteinen olio-ohjelmoinnista tuttu olio, myös hajautettu olio on kokonaisuus, joka kapseloi yhteen datan tai resurssin ja operaatiot eli metodit, jotka käsittelevät kyseistä dataa tai resurssia [25, s. 21–22]. Perinteisen olion tavoin hajautetun olion metodien toteutus erotetaan metodien esittelystä rajapinnan avulla, joka esittelee olion toteuttamat metodit. Saman rajapinnan voi toteuttaa eri tavoilla usea eri olio (monimuotoisuus) ja yksi olio voi toteuttaa monta eri rajapintaa (moniperintä).

Erotuksena perinteisestä oliosta, joka on olemassa vain yksittäisen ohjelmainstanssin sisällä, hajautettu olio on itsenäinen kokonaisuus, jota ei ole sidottu tiettyyn ohjelmainstanssiin [25, s. 21–22]. Koska hajautettu olio toimii hajautetussa ympäristössä, sen toteutus ei myöskään ole sidottu tiettyyn ohjelmointikieleen, käyttöjärjestelmään, laitteistoon tai tietoliikenneprotokollaan. Luvussa 4.1 annetun määritelmän mukainen komponentti voitaisiin siis itse asiassa mieltää hajautetuksi olioksi.

Hajautettuihin olioihin perustuvassa järjestelmän hajautuksessa olion *tilaa (state)* ei yleensä hajauteta, vaan ainoastaan olion toteuttamat rajapinnat annetaan muiden järjestelmän osien tietoon, jotta ne osaisivat kutsua olion metodeja [42, s. 86]. Tällaisia olioita voidaan kutsua myös *etäolioiksi (remote object)* [42, s. 87].

4.4.5 Viestijonoarkkitehtuuri

Asiakas-palvelin roolijakoa noudattavien järjestelmien pyyntö-vaste pareihin ja hajautettujen olioiden metodikutsuihin perustuva kommunikointi on luonteeltaan luvussa 4.4.2 esitetyn Tanenbaumin ym. luokittelun mukaista tilapäistä kommunikaatiota ja yleensä myös synkronista. *Viestijonoihin perustuvat järjestelmät (message-queuing system)* mahdollistavat hajautetun järjestelmän komponenttien välisen asynkronisen pysyvän kommunikaation [42, s. 108].

Viestijonojen avulla hajautetun järjestelmän komponenttien välisiä viestejä voidaan säilöä välitettäessä viestejä lähettäjältä vastaanottajalle [42, s. 108]. Jokaisella komponentilla on oma viestijono, johon muut komponentit voivat lähettää viestejä. Viestijono ja komponentti ovat erotettu toisistaan siten, että komponentin jonoon voidaan lähettää viestejä, vaikka komponentin suoritussäie ei olisi aktiivinen. Välitettävät viestit voivat sisältää mitä tahansa dataa, mutta viestin on osoitettava viestin vastaanottaja [42, s. 108]. Viestin vastaanottaja osoitetaan määrittelemällä vastaanottajan viestijonon uniikki tunniste.

Hajautetun järjestelmän sisältämät viestijonot on hajautettu järjestelmän eri solmuihin [42, s. 111]. Viestin lähettävä komponentti laittaa viestin oman solmunsa paikalliseen jonoon ja määrittää viestiin jonon, johon viesti tulisi toimittaa. Järjestelmä tarjoaa viestejä lähettäville ja vastaanottaville komponenteille tarvittavat jonot ja huolehtii viestien välityksestä jonojen välillä.

Viestijonoihin perustuvan järjestelmän komponenttien on pystyttävä tulkitsemaan toisten komponenttien lähettämiä viestejä [42, s. 113]. Jotta tämä olisi mahdollista, lähetettävän viestin on oltava viestin vastaanottajan ymmärtämässä muodossa. Tämä voidaan toteuttaa kahdella eri tavalla. Järjestelmän sisällä välitettäville viesteille voidaan määrittää yhteinen muoto, jonka mukaan kaikki komponentit muodostavat lähetettävät viestit. Toinen vaihtoehto on muuntaa lähetetyt viestit ennen vastaanottajalle toimittamista järjestelmän sisällä vastaanottajan ymmärtämään muotoon.

4.4.6 Julkaise-tilaa arkkitehtuuri

Julkaise-tilaa (publish-subscribe) arkkitehtuurimallissa järjestelmän komponentit julkaisevat tuottamaansa tietoa ja muut järjestelmän komponentit voivat tilata toistensa tuottamaa tietoa. Järjestelmän komponenttien väliset kytkennät määräytyvät pelkästään tuotetun ja tilatun tiedon perusteella, eikä komponenttien välille tarvitse muodostaa erillisiä kytkentöjä [21]. Tiedon tilaajan ei tarvitse tuntea tiedon lähdettä eikä julkaisijan tiedon vastaanottajia.

Julkaise-tilaa mallia tai mallin mukaista *edustaja (observer)* [44] suunnittelumallia

on käytetty usein perinteisissä ei-hajautetuissa ohjelmistoissa, kuten käyttöliittymätyökaluissa, tapahtumankäsittelijöiden yhdistämiseen tapahtumalähteisiin. Myös hajautetussa järjestelmässä mallia voidaan käyttää tapahtumatietojen välitykseen [17]. Välitettävä tieto voi olla myös puhdasta dataa, joka identifioidaan määrittelemällä julkaistun datan *aihe (topic)* [21].

Julkaise-tilaa mallissa järjestelmän komponenttien toteutuksessa ei tarvitse kiinnittää huomiota yhteyksien muodostamiseen muihin järjestelmän komponenttehin [21]. Järjestelmä huolehtii julkaistun tiedon välityksestä tietoa tilanneille komponenteille. Viestijonoihin perustuvissa järjestelmissä julkaise-tilaa mallia voidaan käyttää *pisteestä pisteeseen (point-to-point)* perustuvan viestinvälityksen sijasta kun halutaan tehdä komponenttien välisistä kytkennöistä löyhiä [8].

5 Väliohjelmistot järjestelmän hajautuksessa

Hajautetun ympäristön heterogeenisyyttä ja järjestelmän hajautuksen läpinäkyvyyttä voidaan hallita järjestelmään lisättävän ylimääräisen ohjelmistokerroksen eli *väliohjelmiston (middleware)* avulla [42, s. 37]. Väliohjelmistokerros toteuttaa hajautetun järjestelmän komponenttien väliset kommunikointimekanismit ja tarjoaa useita hajautettujen järjestelmien kehityksessä tarvittavia palveluja.

Tässä luvussa annetaan yleinen esittely väliohjelmistoista keskittyen edellisen luvun tavoin erityisesti sellaisiin väliohjelmistojen tarjoamiin toteutusmekanismeihin, joiden katsotaan olevan tärkeitä hajautetun tuoterungon kannalta. Luku 5.1 esittelee ensin väliohjelmiston määritelmän, jonka jälkeen luvussa 5.2 käsitellään väliohjelmistojen käyttöä hajautetun järjestelmän komponenttien välisessä kommunikoinnissa. Luvussa 5.3 esitellään väliohjelmistojen hajautettujen järjestelmien kehitykseen tarjoamia palveluja ja lopuksi luvussa 5.4 tarkastellaan esimerkkejä palveluiden toteutuksista.

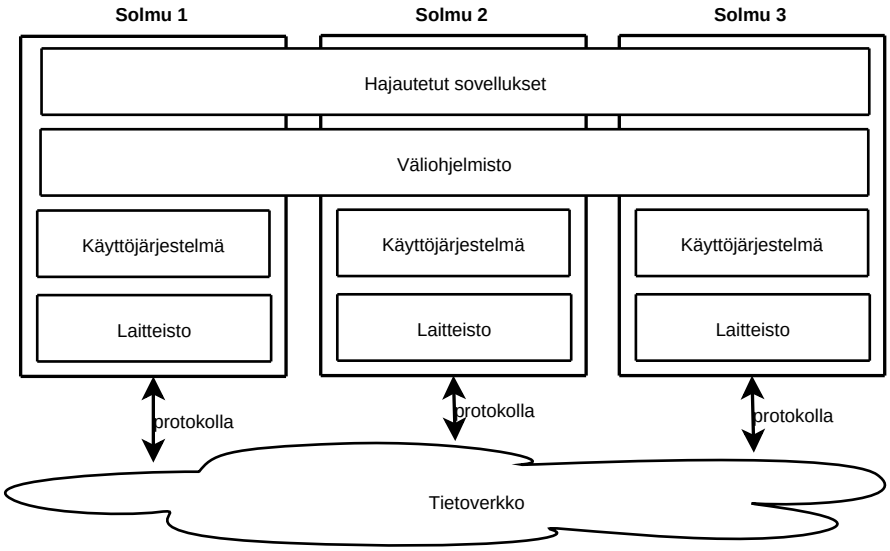
5.1 Väliohjelmiston määritelmä

Väliohjelmiston avulla voidaan hallita hajautettujen järjestelmien toimintaympäristön heterogeenisyyttä. Kuvassa 5.1 on esitetty hajautetun järjestelmän toimintaympäristöä loogisen kerrosmallin avulla. Väliohjelmisto on ohjelmistokerros, joka sijoittuu loogisesti hajautettujen sovellusten muodostaman kerroksen ja käyttöjärjestelmien muodostaman kerroksen väliin [42, s. 37].

Kuvan 5.1 jokaisessa järjestelmän solmussa käyttöjärjestelmä huolehtii paikallisista resursseista ja tarjoaa alempien kerroksien palvelujen avulla keinot kommunikointiin muiden järjestelmän solmujen kanssa. Väliohjelmisto peittää allaan olevien kerroksien tarjoamien palveluiden heterogeenisyyden hajautetuilta sovelluksilta ja tarjoaa jokaisessa solmussa yhtenäisiä palveluja järjestelmän hajautuksen hallintaan [42, s. 36–37].

5.2 Väliohjelmiston kommunikaatiomekanismit

Yksi väliohjelmistojen tärkeimmistä tehtävistä on tarjota hajautetun järjestelmän komponenteille järjestelmän solmujen välisen matalan tason viestinvälityksen peittävä menetelmä kommunikoida keskenään [42, s. 36]. Kommunikaation mahdollistamiseksi jär-



Kuva 5.1: Väliohjelmisto. [42]

jestelmän komponentit muodostavat ohjelmarakenteet on tunnettava ja määritettävä tapa, jolla ohjelmarakenteiden välillä voidaan välittää viestejä.

5.2.1 Etäproseduurikutsut

Ensimmäiset sijaintiläpinäkyvyyden toteuttavat hajautettujen järjestelmien ja väliohjelmistojen kommunikointimekanismit perustuivat Birrellin ym. [23] vuonna 1983 esittämään *etäproseduurikutsujen (Remote Procedure Call, RPC)* toteutukseen [42, s. 68]. Birrellin ym. etäproseduurikutsujen toteutus sisältää monia tärkeitä periaatteita, joiden pohjalta on myöhemmin kehitetty muita väliohjelmistojen kommunikointimekanismeja ja joiden avulla voidaan tarkastella muiden mekanismien toteutuksia. Kuten myöhemmin tehtävässä tarkastelussa huomataan, monet viimeaikaisistakin väliohjelmistojen kommunikointimekanismeista perustuvat Birnellin ym. esittämän etäproseedurikutsujen toteutuksen pohjalta kehitettyjen menetelmien käyttöön.

Etäproseduurikutsujen ajatuksena on mahdollistaa toisessa koneessa sijaitsevien ohjelmaproseduurien kutsuminen paikallisten proseduurien tapaan [23]. Kun etäproseduurikutsu suoritetaan, kutsuvan ympäristön toiminta keskeytetään, proseduurin parametrit välitetään proseduurin suorittavaan ympäristöön tietoliikenneyhteyden yli ja proseduuri suoritetaan tässä ympäristössä. Kun proseduuri on suoritettu, proseduurin palauttamat tulokset palautetaan kutsun suorittaneeseen ympäristöön, jonka toiminta jatkuu kuten paikallisen proseduurikutsun yhteydessäkin.

Birrellin ym. esittämä etäproseduurikutsujen toteutus perustuu niin sanottujen *tyn-*

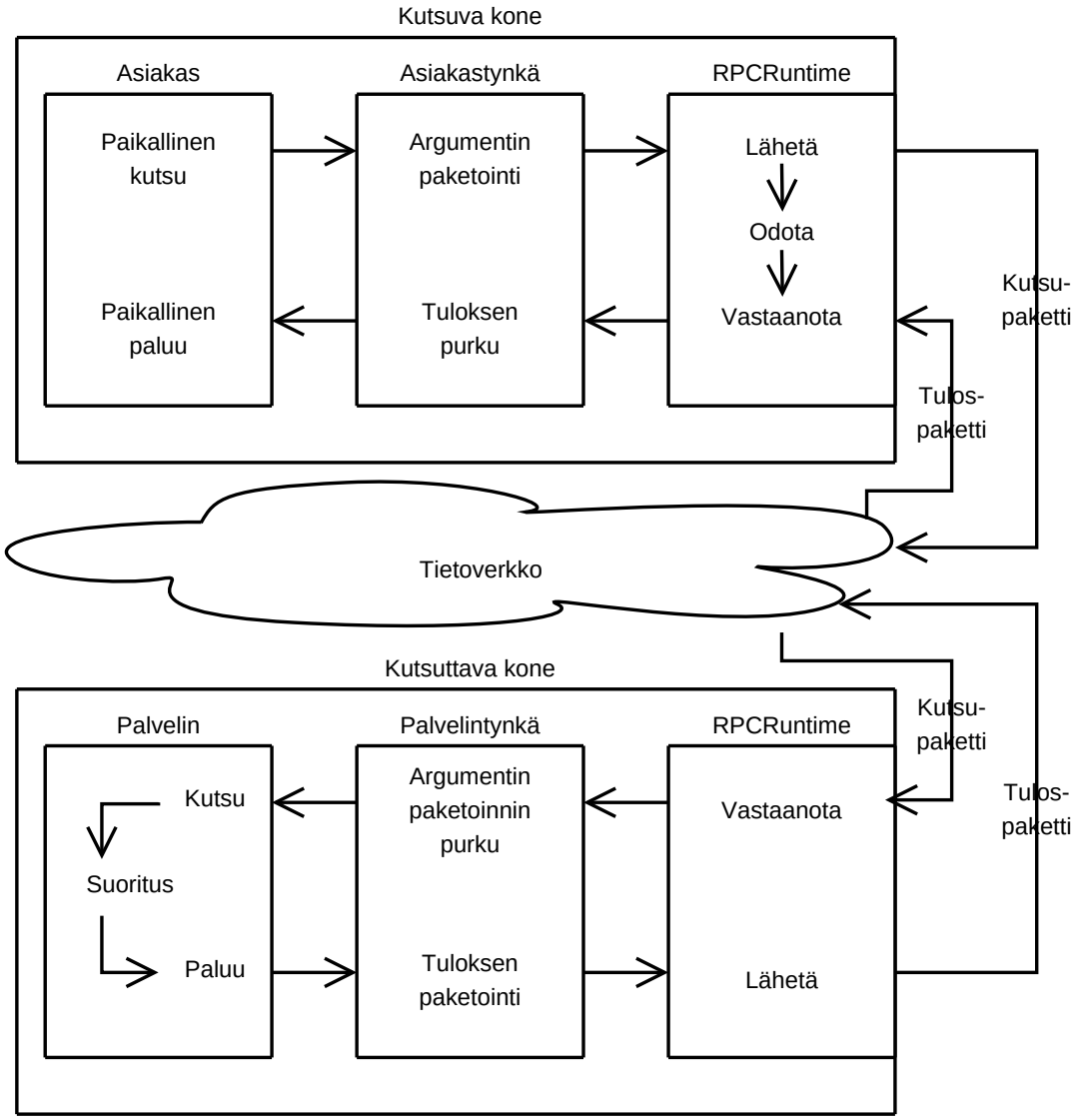
kien (stub) käyttöön ja sisältää viisi eri ohjelmakokonaisuutta: asiakas (*user*), asiakastynkä (*user-stub*), RPC kommunikaatiopaketti (*RPCRuntime*), palvelintynkä (*server-stub*) ja palvelin (*server*) [23]. Etäproseduurikutsun tekevällä eli kutsuvalla koneella toimivat asiakas, asiakastynkä ja yksi kommunikaatiopaketin ilmentymä. Kutsuttavalla koneella sijaitsee palvelin, palvelintynkä ja yksi kommunikaatiopaketin ilmentymä. Palvelin on ohjelma, joka sisältää kutsuttavien proseduurien toteutukset ja asiakas ohjelma, joka kutsuu palvelimen sisältämiä proseduureja.

Birrellin ym. määrittelemä tynkä on tärkeä käsite, joka toistuu myös muissa kommunikointimekanismeissa. Birnellin ym. esittämien tynkien muodostaminen perustui *Mesa*-ohjelmointikielen *rajapintamoduulien (interface module)* käyttöön, joita asiakas ja palvelin voivat käyttää käännösaikaisessa tyyppitarkastuksessa [23]. Mesa-ohjelmointikieli määrittelee kahdenlaisia moduuleja: rajapintamoduuli (*DEFINITIONS*) ja *toteutusmoduuli (PROGRAM)* [30]. Rajapintamoduuli sisältää vakioita ja toteutusmoduulin tarjoamien proseduurien esittelyt. Rajapintamoduulien avulla erotetaan proseduurien *kääntäminen (compilation)* niiden *sidonnasta (binding)* [23].

Etäproseduureja hyödyntävän järjestelmän toteuttajan on kirjoitettava rajapintamoduuli, joka sisältää järjestelmässä käytettävien etäproseduurien esittelyt [23]. Asiakas- ja palvelinohjelmaa kirjoitettaessa rajapintamoduulia voidaan käyttää käännösaikaiseen tyyppitarkastukseen. Erillinen ohjelma muodostaa automaattisesti rajapintamoduulin avulla asiakas- ja palvelintyngät, joiden avulla asiakas ja palvelin sitoutuvat rajapintamoduulissa määriteltyihin proseduureihin. Palvelintynkä sidotaan palvelimeen, joka toteuttaa rajapintamoduulissa määritellyt proseduurit eli *tarjoaa (export)* rajapinnan [23]. Asiakastynkä sidotaan asiakkaaseen, joka kutsuu rajapintamoduulissa määriteltyjä proseduureja eli *käyttää (import)* rajapintaa [23].

Kuvassa 5.2 on esitetty etäproseduurikutsun kulkua hajautetussa järjestelmässä. Kun asiakas haluaa tehdä etäkutsun, se tekee normaalin paikallisen kutsun proseduurille, joka on määritelty asiakastyngässä. Asiakastynkä paketoi kutsun viestiksi, joka identifioi kutsuttavan proseduurin ja sisältää kutsun parametrit. Tämän jälkeen asiakastynkä kutsuu RPC kommunikaatiopakettia, joka välittää viestin kutsuttavan koneen kommunikaatiopaketille. Kutsuttavan koneen kommunikaatiopaketti puolestaan välittää viestin palvelintyngälle, joka purkaa viestin ja tekee paikallisen proseduurikutsun palvelimen sisältämälle proseduurille. Kun palvelin on suorittanut proseduurin, palvelintynkä paketoi proseduurin palauttamat tulokset viestiksi ja antaa viestin kommunikaatiopaketin välitettäväksi takaisin kutsuvan koneen kommunikaatiopaketille. Kutsuvan koneen kommunikaatiopaketti välittää viestin asiakastyngälle, joka purkaa viestin ja palauttaa tulokset asiakkaalle.

Birrellin ym. [23] etäproseduurikutsun toteutus on esimerkki luvussa 4 esitellyn Ta-



Kuva 5.2: Etäproseduurikutsu. [23]

nenbaumin ym. [42] luokittelun mukaisesta vasteeseen perustuvasta tilapäisestä synkronisesta kommunikaatiosta. Toteutuksessa ei erotella paikallisen koneen suoritussäikeiden välisiä etäproseduurikutsuja koneiden välisistä etäproseduurikutsuista, mikä heikentää järjestelmän suorituskykyä [42, s. 77]. Usein hajautetussa järjestelmässä kutsun suorittavan asiakkaan on myös turha jäädä odottamaan kutsun käsittelyä [42, s. 79]. Näitä puutteita on paikattu kehittämällä toteutuksia, jotka mahdollistavat asynkroniset etäproseduurikutsut ja ottavat huomioon paikallisen koneen suoritussäikeiden väliset etäproseduurikutsut [42, s. 77–80].

5.2.2 Viestien esitystapa ja järjestäminen

Hajautetun järjestelmän solmut voivat sisältää erilaisia laitteistoarkkitehtuureja, joilla on erilaisia esitystapoja datatyypeille kuten kokonaisluvuille tai kirjaimille. Tämän vuoksi hajautetulla järjestelmällä on oltava yhteinen laitteistoriippumaton esitystapa käytettäville datatyypeille ja solmujen välillä välitettävien viestien sisältämä data täytyy *järjestää (marshal)* tämän yhteisen esitystavan mukaiseksi [42, s. 73].

Esimerkiksi etäproseduurikutsujen parametrit ja tulokset on esitettävä yhteisen esitystavan mukaisesti, jotta asiakas ja palvelintyngät osaavat koota ja purkaa viestit oikein. Esimerkkinä yhteisestä esitystavasta voidaan pitää *ONC*:n *(Open Network Computing)* etäproseduurikutsuprotokollan [58] käyttämää *XDR (External Data Representation)* -standardia [13]. XDR-standardi määrittelee kuvauskielen, jonka avulla datatyypit voidaan kuvata laitteistoriippumattomasti.

Eri laitteistoarkkitehtuurit käyttävät myös erilaista *bitti- (bit order) tai tavujärjestystä (byte order)* datan esityksessä. Bittijärjestys voi olla joko *little-endian*, jossa *vähiten merkitsevä bitti (least significant bit, LSB)* esitetään ensin tai *big-endian*, jossa *eniten merkitsevä bitti* esitetään ensin [11]. Tästä seuraa, että vaikka hajautetulla järjestelmällä olisi yhteinen esitystapa datatyypeille, voidaan lähetetty data tulkita erilaisesti eri bittijärjestystä käyttävissä laitteistoarkkitehtuureissa. Järjestelmän sisällä on siis sovittava, missä järjestyksessä bitit esitetään.

Esimerkiksi XDR käyttää tavujen (kahdeksan bittiä) koodauksessa big-endian menetelmää ja olettaa, että tavut ovat siirrettäviä eri laitteistojen välillä [13]. Tämä tarkoittaa, että laitteistojen on koodattava lähetettävät tavut eri tiedonsiirtomedioille median käyttämässä muodossa [13]. Esimerkiksi IP protokollaa [54] käytettäessä tavut on koodattava big-endian menetelmää käyttäen *(network byte order)*.

XDR:n tekemä siirrettävien tavujen oletus aiheuttaa tehottomuutta tilanteessa, jossa kaksi samaa tavujärjestystä käyttävää laitteistoarkkitehtuuria kommunikoi keskenään laitteistojen kanssa eri tavujärjestystä käyttävän tiedonsiirtomedian välityksellä.

Tällaisessa tapauksessa molemmat osapuolet joutuvat koodaamaan lähettämänsä datan eri tavujärjestykseen, mitä ne itse käyttävät, sekä vastaavasti purkamaan koodauksen vastaanotettaessa dataa.

OMG:n *CORBA (Common Object Request Broker Architecture) GIOP (General Inter-ORB Protocol)* -protokollan käyttämä *CDR (Common Data Representation)* [15] -määritys puolestaan käyttää viestikohtaista koodausta, jossa jokainen viestipaketti sisältää tiedon siitä, onko viestin koodauksessa käytetty little-endian vai big-endian menetelmää. Tämän menetelmän avulla kaksi samaa menetelmää käyttävää laitteistoarkkitehtuuria voi kommunikoida keskenään ilman ylimääräisiä järjestyksen muunnoksia.

5.2.3 Rajapintojen kuvaus

Birrellin ym. [23] etäproseedurikutsujen toteutuksessa proseduurien kuvaus erotettiin niiden toteutuksesta Mesa-rajapintamoduulien avulla. Toteutuksen kuvauksen erotus varsinaisesta toteutuksesta rajapinnan avulla on tärkeä periaate järjestelmien hajautuksessa. Myös hajautettuihin olioihin perustuvat *oliomallit (object model)*, kuten OMG:n oliomalli *(Core Object Model)* [14] ja siitä johdettu CORBAn oliomalli [15] tai ISO RM-ODP:n määrittelemä oliomalli [5], perustuvat olion rajapinnan ja toteutuksen erotukseen.

Rajapintojen kuvaukseen tarvitaan menetelmä, jonka avulla rajapinnat voidaan kuvata siten, että erilaiset hajautetussa ympäristössä toimivat rajapinnan käyttäjät ja rajapinnan toteuttajat osaavat tulkita rajapinnan kuvauksen oikein [15]. Rajapintaa käyttävät ohjelmakokonaisuudet ja rajapinnan toteuttavat ohjelmakokonaisuudet voivat olla kirjoitettu eri ohjelmointikielillä ja ne toimivat hajautetussa ympäristössä, joka voi sisältää erilaisia laitteistoarkkitehtuureja, käyttöjärjestelmiä ja tietoliikenneprotokollia.

Hajautetun järjestelmän rajapintoja voidaan kuvata *rajapintojen kuvauskielien (Interface Definition Language, IDL)* avulla. ONC:n etäproseduuriprotokolla määrittelee rajapintojen kuvaukseen RPC-kielen, joka laajentaa XDR-standardia ohjelman, proseduurin ja version käsitteillä [58]. OMG on puolestaan määritellyt rajapintojen kuvaukseen oman kielensä, joka on myös ISO:n hyväksymä standardi [15]. OMG IDL on suunnattu CORBAn hajautettujen olioiden rajapintojen kuvaukseen. Kuvassa 5.3 on esitetty yksinkertainen esimerkki rajapinnan kuvaamisesta OMG IDL -kielellä.

Rajapinnankuvauskieli sisältää rajapinnan kuvaukseen tarvittavat rakenteet, mutta se ei ole ohjelmointikieli, eikä sen avulla voi määritellä ohjelman toiminnallisuutta [58] [15]. Rajapinnankuvauskielen rakenteet on pystyttävä muuntamaan rajapintaa käyttävien ja rajapinnan toteuttavien ohjelmakokonaisuuksien toteutuksessa käytettä-

```
module FooBar {
  interface Foo {
    void foo(in int inputParam, out SomeObject outputParam);
  };

  interface Bar {
    int bar(in string inputParam);
  };
};
```

Kuva 5.3: Rajapintojen kuvaus OMG IDL -kielellä.

vien ohjelmointikielten rakenteisiin. Esimerkiksi OMG on määritellyt oman kuvauskielensä rakenteita vastaavat rakenteet *(language mapping)* useille eri ohjelmointikielille [15].

5.2.4 Etämetodikutsut ja ORB

Birrellin ym. [23] etäproseduurikutsujen toteutuksessa käytettyä RPC kommunikaatiopakettia vastaava mekanismi hajautettujen olioiden yhteydessä voidaan ajatella olevan olioiden välisiä viestejä ja dataa välittävä *ORB (Object Request Broker)*. ORB on *olioväylä (object bus)*, joka mahdollistaa etäolioiden metodien kutsumisen joko staattisesti tai dynaamisesti [25, s. 18].

ORB tarjoaa monia palveluja järjestelmän hajautukseen, joista tärkeimmät ovat rajapintojen määritysmekanismi, etäolioiden paikannus ja mahdollinen aktivoiminen sekä olion ja olion palveluja käyttävän asiakkaan välisen kommunikoinnin toteutus [1]. ORB toteuttaa olioiden välisen kommunikoinnin sijaintiläpinäkyvästi eli siten, että etäolion palveluja kutsuva asiakas ei tiedä kutsumansa olion todellista sijaintia.

Staattisissa *etämetodikutsuissa (Remote Method Invocation, RMI)* sitoutuminen rajapinnan toteutukseen tapahtuu samankaltaisen mekanismin avulla kuin Birrellin ym. [23] etäproseduurikutsuissa. Asiakas sitoutuu käyttämäänsä rajapintaan etäproseduurien asiakastynkää vastaavan *edustajan (proxy)* [42, s. 86] kautta. Palvelin puolestaan sitoutuu toteuttamaansa rajapintaan etäproseduurien palvelintynkää vastaavan *rungon (skeleton)* [42, s. 86] avulla. Edustaja sisältää tietyn olion tarjoaman rajapinnan ja vastaa asiakkaan etämetodikutsujen järjestämisestä viesteiksi sekä kutsun tuloksien purkamisesta. Runko sisältää saman rajapinnan kuin edustaja ja vastaa asiakkaan viestien purkamisesta rajapinnan toteuttavalle palvelimelle sekä palvelimen palauttamien

tuloksien järjestämisestä viesteiksi. Staattisissa kutsuissa kutsuttavien olioiden rajapinnat ovat tiedossa käännettäessä kutsun tekevää ohjelmakoodia. Dynaamisessa kutsussa kutsuttava olio ja metodi voidaan valita ORBin avulla ajonaikaisesti määrittelemällä kutsuttava olio, metodi ja metodin parametrit.

Erotuksena etäproseduurikutsuista, hajautettuihin olioihin perustuvassa järjestelmässä voidaan hyödyntää järjestelmänlaajuisia *olioviitteitä (object reference)* [42, s. 88]. Tämä tarkoittaa, että hajautetun järjestelmän solmujen välillä voidaan välittää viitteitä toisissa solmuissa sijaitseviin olioihin. Olioviitteitä voidaan välittää esimerkiksi metodien parametreina tai metodien palauttamina tuloksina. Ennen kuin asiakas voi kutsua olioviitteen viittaaman olion metodeja, sen täytyy sitoutua viitattuun olioon. Sidottaessa asiakasta kutsuttavaan olioon järjestelmän on selvitettävä viitattavan olion toteutuksen sisältämän solmun sijainti järjestelmässä. ORB huolehtii sidonnasta siten, että etäolion palveluja kutsuva asiakas ei tiedä kutsumansa olion todellista sijaintia.

ORB voi toteuttaa sijaintiläpinäkyvyyden lisäksi myös muita hajautetun järjestelmän läpinäkyvyyspalveluja. ORB voi peittää olion palveluja kutsuvalta asiakkaalta olion sijainnin lisäksi olion toteutuksen yksityiskohdat, kuten olion ohjelmoinnissa käytetyn ohjelmointikielen [1]. Lisäksi ORB voi peittää olion isäntäsolmun ominaisuudet, kuten solmun laitteiston ja käyttöjärjestelmän [1]. ORB voidaan toteuttaa eri tavoilla ja eri toteutustapojen sisältämät läpinäkyvyyspalvelut voivat vaihdella. ORB voi olla esimerkiksi kehitettävään sovellukseen linkitettävä kirjasto, erillinen prosessi tai osa käyttöjärjestelmän ydintä. Esimerkkiä ORBin toteutuksesta tarkastellaan lähemmin luvussa 5.4.

5.3 Väliohjelmistojen palvelut

Väliohjelmistot tarjoavat useita hajautetun järjestelmän kehitystä tukevia palveluja. Schantz ym. [28] määrittelevät hajautettujen olioiden käyttöön perustuville väliohjelmistoille *(distributed object computing (DOC) middleware)* kerrosmallin, joka antaa hyvän pohjan väliohjelmistojen tarjoamien palvelujen tarkasteluun. Malli jakaa väliohjelmistot neljään kerrokseen: *heterogeenisyyden hallinnan palveluja (host infrastructure middleware), hajautukseen liittyviä palveluja (distribution middleware), yleisiä väliohjelmistopalveluja (common middleware services) ja toimialuekohtaisia väliohjelmistopalveluja (domain-specific middleware services)* tarjoava kerros. Kuten kerrosmalleissa yleensä, mallissa ylempi kerros käyttää allaan olevan kerroksen palveluja omien palveluidensa toteutukseen.

Heterogeenisyyden hallinnan palveluja tarjoava kerros sijoittuu laitteiston ja käyttöjärjestelmän sekä erilaisten protokollien muodostaman *alustan (platform)* päälle ja

muodostaa alimman kerroksen väliohjelmistojen kerrosmallissa. Kerros kapseloi käyttöjärjestelmien kommunikaation ja samanaikaisuuden hallintaan tarjoamat mekanismit sekä tarjoaa uudelleenkäytettäviä ja siirrettäviä ohjelmistorakenteita ylemmille kerroksille [28]. Tämän kerroksen COTS toteutuksiksi Schantz ym. [28] luokittelevat *Sun Java virtuaalikoneen*, *Microsoft .NET* ympäristössä käytettävän *CLR*:n *(Common Language Runtime)* sekä *ACE*n *(The ADAPTIVE Communication Environment)*.

Hajautukseen liittyviä palveluja tarjoava kerros sijoittuu heterogeenisyyden hallinnan palveluja tarjoavan kerroksen päälle. Kerros toteuttaa valmiita malleja hajautettujen järjestelmien ohjelmointiin ja tarjoaa sijaintiläpinäkyvän menetelmän hajautettujen olioiden palveluiden kutsumiseen [28]. Kerros tarjoaa mekanismit, joiden avulla hajautettujen olioiden palveluja voidaan kutsua kuten paikallisia palveluja. Esimerkkeinä kerroksen COTS toteutuksista Schantz ym. [28] mainitsevat OMG CORBA -toteutukset, *Sun Java RMI*:n, *Microsoft DCOM*:n *(Distributed Component Object Model)* sekä *SOAP*:n *(Simple Object Access Protocol)*.

Hajautukseen liittyviä palveluja tarjoavan kerroksen päällä Schantzin ym. [28] mallissa sijaitsee yleisiä väliohjelmistopalveluja tarjoava kerros. Kerros laajentaa hajautukseen liittyvien palveluiden kerrosta toteuttamalla kohdealueesta riippumattomia, hajautetun järjestelmän vaatimuksia täyttäviä palveluja, jotka helpottavat sovelluskehittäjien keskittymistä oman kohdealueensa vaatimusten toteuttamiseen itse hajautuksen asettamien vaatimusten sijasta [28]. Schantz ym. antavat esimerkeiksi tämän kerroksen COTS toteutuksista OMG CORBA *Common Object Services*, *Sun Enterprise Java Beans (EJB)* ja *Microsoft .NET Web Services* teknologiat. Schantzin ym. luokittelun jälkeen kehittyneistä teknologioista tähän kerrokseen voitaisiin todennäköisesti lisätä myös *CORBA Component Model (CCM)*.

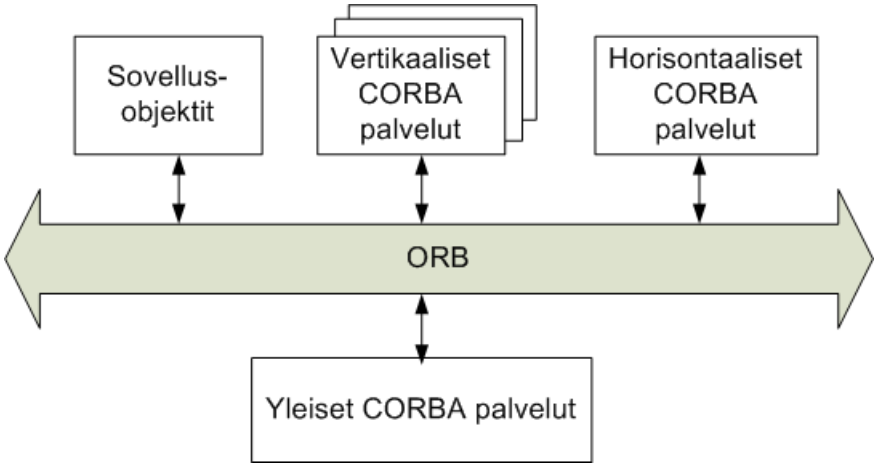
Ylimmän väliohjelmistokerroksen, jonka päälle varsinaiset sovellukset sijoittuvat, muodostaa toimialuekohtaiset väliohjelmistopalvelut. Kerrokseen kuuluvat eri toimialueille kuten televiestintään, pankkitoimintaan tai terveydenhuoltoon suunnatut palvelut [28]. Schantzin ym. [28] mukaan valmiiden kaupallisten toimialuekohtaisten väliohjelmistopalvelujen kehityksen edellytyksenä on kehittyneiden alempien kerrosten palveluiden saatavuus, mikä on viivästyttänyt kerroksen palvelujen toteutusta. Eri toimialueille on kuitenkin olemassa joitakin valmiita kaupallisia väliohjelmistopalveluja.

5.4 Esimerkkejä väliohjelmistoista

Tässä luvussa esitellään lyhyesti muutama esimerkki edellisen luvun Schantzin ym. [28] kerrosmallin väliohjelmistojen palveluiden toteutuksista. Schantzin ym. mallin alimman kerroksen toteutuksen esimerkiksi on valittu luvussa 5.4.1 esiteltävä ACE, joka

tarjoaa käännettävän rajapinnan heterogeenisyyden hallintaan. Hajautukseen liittyviä palveluja ja yleisiä väliohjelmistopalveluja tarjoavista väliohjelmistoista esitellään luvuissa 5.4.2 ja 5.4.3 *OMG (The Object Management Group)* järjestön CORBA ja CCM-määritykset.

OMG:n määritykset perustuvat OMA (Object Management Architecture) referenssiarkkitehtuuriin [14], joka kokoaa hajautettujen sovellusten yhteisen toiminnallisuuden standardin mukaisiksi elementeiksi. OMG standardoi elementtien rajapinnat OMG IDL rajapinnankuvauskielen avulla sekä määrittelee elementeiltä vaadittavan toiminnallisuuden. Ohjelmistovalmistajat toteuttavat elementtejä OMG:n standardin mukaisesti ja myyvät omia toteutuksiaan eteenpäin. Kuvassa 5.4 esitetty OMA arkkitehtuuri koostuu neljästä elementistä, joita yhdistää yhteinen ORBin muodostama ohjelmistoväylä. Yleiset palvelut tarjoavat hajautuksessa tarvittavia yleisiä palveluja, horisontaaliset palvelut yleisiä kohdealueesta riippumattomia palveluja ja vertikaaliset palvelut määrätylle kohdealueelle suunnattuja palveluja. Sovellusobjekteilla tarkoitetaan sovelluskohtaisia ohjelmarakenteita.



Kuva 5.4: OMA.

5.4.1 ACE

ACE (The ADAPTIVE Communication Environment) [2] on esimerkki käännettävän rajapinnan tarjoamasta väliohjelmistototoetuksesta. ACE on oliopohjainen ohjelmistokehys, joka toteuttaa hajautettujen järjestelmien hajautuksen, samanaikaisuuden hallinnan ja komponenttien välisen kommunikoinnin suunnittelumalleja [56]. ACE on toteutettu C++-ohjelmointikielellä ja sen tarjoamat *C++-kääreluokat (wrapper class)* ja komponentit toimivat monien eri käyttöjärjestelmien päällä, kuten Windowsin eri

versioiden, useimpien UNIX versioiden ja useiden reaaliaikakäyttöjärjestelmien päällä [56].

ACE on suunniteltu kerrosarkkitehtuurin mukaisesti [57]. Arkkitehtuurin alin kerros rakentuu käyttöjärjestelmien tarjoamien matalan tason C-kielisten ohjelmointirajapintojen päälle ja tarjoaa ylemmille kerroksille yhtenäisen ohjelmointirajapinnan peittämällä käyttöjärjestelmien ohjelmointirajapintojen heterogeenisyyden [57]. Kerros kapseloi käyttöjärjestelmien tarjoamat mekanismit ja tarjoaa käyttöjärjestelmäriippumattoman ohjelmointirajapinnan samanaikaisuuden ja synkronoinnin hallintaan, prosessien välisen kommunikoinnin ja jaetun muistin hallintaan, tapahtumien kanavointiin, eksplisiittiseen dynaamiseen linkitykseen sekä tiedostojärjestelmien käyttöön.

Käyttöjärjestelmien ohjelmointirajapintojen heterogeenisyyden peittävän kerroksen tarjoamaa C-kielistä ohjelmointirajapintaa voi käyttää suoraan sovellusten ohjelmointiin, mutta sen käyttö on työlästä ja hankalaa [57]. ACEn heterogeenisyyden peittävän kerroksen päälle sijoittuu C++-kääreluokista koostuva kerros, joka helpottaa sovellusten ohjelmointia tarjoamalla tyyppiturvallisia C++-rajapintoja sovellusten ohjelmointiin. Näiden rajapintojen tarkoitus on helpottaa ACEn käyttöä ja käytön oppimista. Kääreluokkia voidaan käyttää sovelluksissa perimällä ja koostamalla luokkia sekä luomalla luokkien ilmentymiä. ACEn kehittäjien mukaan C++-kielen vahva tyypitys auttaa havaitsemaan käännösaikaisesti verkko-ohjelmoinnissa käytettävien C-kielisten ohjelmointirajapintojen, kuten *vastakkeiden (socket)* käytössä usein esiintyviä tyyppeihin liittyviä virheitä.

C++-kääreluokkien muodostaman kerroksen päälle sijoittuu ACEn kerrosarkkitehtuurin ylin kerros, joka yhdistää ja tehostaa kääreluokkien toiminnallisuutta [57]. Kerros koostuu hajautetun järjestelmän komponenttien välisen kommunikaation toteuttamiseen suunnatuista komponenttikehyksistä. Komponenttikehysten avulla voidaan hallita hajautetun järjestelmän komponenttien välistä yhteistyötä ja järjestelmän tarjoamia palveluja. Kerros sisältää myös ORB-sovitinkomponentteja, joiden avulla ACE voidaan yhdistää CORBA-toteutuksien kanssa.

ACEn kerrosarkkitehtuuri määrittelee varsinaisen ohjelmistokehyksen ulkopuolella lisäksi komponenttikirjaston, joka sisältää itsenäisiä hajautettuja palveluja tarjoavia komponentteja [57]. Komponentit toteuttavat hajautettujen järjestelmien kehityksessä tarvittavia palveluja kuten nimeäminen, tapahtumien reititys ja tietojenkeruu. Valmiit komponentit toimivat myös esimerkkeinä varsinaisen ohjelmistokehyksen sisältämien rakenteiden käytöstä.

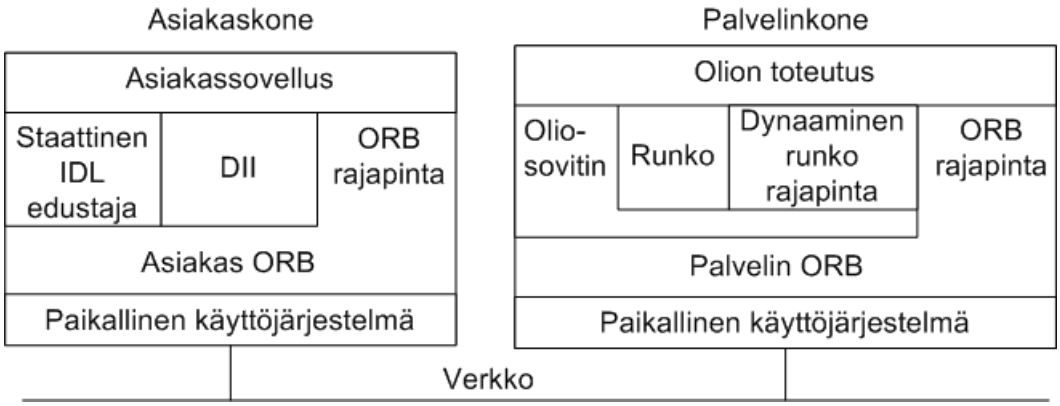
ACEn kehittäjät ovat myös toteuttaneet korkeamman tason väliohjelmistopalveluja ACE-ohjelmistokehyksen päälle. Esimerkiksi *TAO (The ACE ORB)* [3] on ACE-ohjelmistokehystä hyödyntävä reaaliaikainen CORBA-toteutus ja *CIAO (The Component-*

Integrated ACE ORB) [4] TAO:n päälle kehitettävä *CORBAn kevyen komponenttimallin (Lightweight CORBA Component Model (CCM))* [16] toteutus.

5.4.2 OMG CORBA

CORBA (Common Object Request Broker) [15] on OMG OMA -arkkitehtuuriin perustuva hajautettujen järjestelmien kehitystä tukeva määritys. CORBAn ytimen muodostaa ORB, joka mahdollistaa hajautetun järjestelmän sovellusten välisen kommunikaation asiakas-palvelin arkkitehtuurin mukaisesti. CORBA-määritykseen sisältyy myös kokoelma hajautettujen olioiden kehitystä ja käyttöä tukevia palveluja. Palvelut toteuttavat esimerkiksi olioiden nimeämiseen (*naming*), elinkaaren hallintaan (*life cycle*), transaktioihin, turvallisuuteen (*security*), pysyvään tilaan (*persistence*), samanaikaisuuden hallintaan (*concurrency*) sekä tapahtumailmoituksiin (*event* ja *notification service*) liittyviä mekanismeja. Tässä esittelyssä keskitytään kuitenkin CORBAn arkkitehtuurin sekä oliomallin tarkasteluun ja palveluiden tarkempi käsittely sivuutetaan.

CORBAn oliomalli perustuu hajautettuihin olioihin, joiden tarjoamia palveluja asiakkaat voivat kutsua. Olioiden rajapinnat kuvataan OMG IDL -rajapinnankuvauskielen avulla [15]. CORBA-järjestelmä koostuu hajautettujen olioiden toteutuksia hallinnoivista oliopalvelimista ja olioiden tarjoamia palveluita käyttävistä asiakkaista kuvan 5.5 mukaisesti.



Kuva 5.5: CORBA. [42]

Kuten kuvasta 5.5 nähdään, kaikkien CORBA-asiakkaiden ja palvelimien pohjalla toimii ORB. ORB vastaa kaikista mekanismeista, joita tarvitaan kutsuttavan olion toteutuksen etsimisessä, olion toteutuksen valmistelussa palvelupyyntöön sekä palvelupyynnön vaatiman kommunikaation toteutuksessa [15]. Asiakkaan näkemä olion rajapinta on riippumaton olion sijainnista, olion toteutustavasta ja kaikista muista rajapinnassa kuvaamattomista tekijöistä [15].

Asiakas suorittaa olion palvelukutsun olioviitteen avulla joko palvelun toteuttavan olion rajapinnankuvauksesta muodostetun staattisen *edustajan (proxy)* tai dynaamisen *DII (Dynamic Invocation Interface)* -rajapinnan kautta [15]. Edustaja kokoaa (*marshal*) palvelukutsun viestiksi, jonka ORB välittää olion toteuksen sisältävälle oliopalvelimelle. Edustaja myös purkaa (*unmarshal*) kutsun tulokset sisältävän viestin ja palauttaa tulokset asiakkaalle. DII mahdollistaa palvelukutsun muodostamisen ajonaikaisesti tarjoamalla geneerisen `invoke`-operaation, jolle annetaan parametriksi viite kutsuttavaan olioon, kutsuttavan metodin tunniste sekä lista syöte- ja tulosparametreista. Olion toteutus vastaanottaa palvelukutsun kutsun sisältävän viestin purkamisesta vastaavan staattisen tai dynaamisen *rungon (skeleton)* välityksellä [15].

Ajonaikaisten palvelukutsujen muodostamisen mahdollistamiseksi olioiden rajapinnat voidaan tallentaa *rajapinnankuvauskantaan (interface repository)*, josta ORB saa asiakkaan dynaamisten kutsujen suorituksessa tarvittavat tiedot [15]. Rajapinnankuvauskanta toteutetaan usein erillisenä prosessina, joka tarjoaa standardoidun rajapinnan rajapintojen kuvauksien tallentamiseen ja noutamiseen [42]. Rajapintoja voidaan noutaa tietokannasta IDL-kääntäjän rajapinnalle osoittaman tunnisteen (*repository identifier*) avulla. Rajapintatietokantaan voidaan myös tallentaa rajapintoihin liittyviä lisätietoja, kuten debug-tietoja, edustaja- tai runkokirjastoja ja olioiden käsittelyrutiineja [15].

Toteutustietokanta (implementation repository) sisältää tietoja, joiden avulla ORB voi paikantaa ja aktivoida olioiden toteutuksia [15]. Olioiden toteutusten asennus sekä olioiden aktivointiin ja suoritukseen liittyvien menettelytapojen hallinta suoritetaan toteutustietokannan avulla. Esimerkiksi olion aktivoinnista huolehtiva oliosovitin voi noutaa aktivoinnissa tarvittavia tietoja toteutustietokannasta. Tietokantaan voidaan tallentaa myös olioiden toteutuksiin liittyviä lisätietoja, kuten debug-, hallinnointi-, resurssointi- ja turvallisuustietoja [15].

Hajautetulle oliolle osoitettujen palvelukutsujen purkamisesta ja tulosten järjestämisestä vastaava runko liitetään ORBiin *oliosovittimen (object adapter)* avulla. Oliosovitin on olion toteutuksen pääasiallinen keino päästä käsiksi ORBin tarjoamiin palveluihin [15]. Oliosovitin vaatii toimiakseen ORBin tarjoaman yksityisen rajapinnan ja se tarjoaa yksityisen rajapinnan siihen liitetylle rungolle sekä julkisen rajapinnan olion toteutukselle. Oliosovitin vastaa olioviitteiden muodostamisesta ja tulkinnasta, metodikutsujen suorituksesta, vuorovaikutusten turvallisuudesta, olioiden aktivoinnista ja deaktivoinnista, olioviitteiden muuntamisesta viitteitä vastaaviin olioiden toteutuksiin sekä toteutusten rekisteröinnistä [15]. Oliosovitin voi toteuttaa vastuualueensa itse tai siirtää vastuun ORBille. Eri ORBien tarjoamien palveluiden taso vaihtelee, minkä vuoksi oliosovittimen toteutus riippuu sen alla toimivasta ORBista.

Oliosovittimia voi olla erilaisia, eikä CORBA-määritys rajoita mahdollisten oliosovittimien määrää. Koska olioiden toteutukset ovat riippuvaisia oliosovittimen tarjoamasta rajapinnasta, määrityksessä kuitenkin todetaan olevan suotavaa, että erilaisten oliosovittimien määrä olisi mahdollisimman pieni [15]. Oliosovittimien määrän minimoimiseksi CORBA määrittelee POA (Portable Object Adapter) sovittimen, jonka on tarkoitus toimia useiden ORBien päällä ja jonka muuntaminen eri valmistajien ORBien kanssa yhteensopivaksi vaatii mahdollisimman vähän muutoksia.

Yhteentoimivuudella (interoperability) tarkoitetaan heterogeenisessä ympäristössä toimivien erilaisten ORBien kykyä kommunikoida keskenään. ORBien yhteentoimivuus toteutetaan *ORBien välisillä protokollilla (inter-ORB protocol) ja silloilla (inter-ORB bridge)* [15].

CORBA määrittelee kaksi ORBien välistä protokollaa: *GIOP (General Inter-ORB Protocol) ja IIOP (Internet Inter-ORB Protocol)* [15] -protokollat. GIOP on abstrakti protokolla, joka määrittelee ORBien välisessä kommunikaatiossa käytettävät viestit ja datan esitystavan (*Commmon Data Representation, CDR*). GIOP vaatii alleen kuljetusprotokollan, jonka on oltava muun muassa luotettava ja yhteydellinen. GIOP-protokollaa voidaan käyttää kaikkien ehdot täyttävien kuljetusprotokollien päällä toteuttamalla sen mukainen, haluttua kuljetusprotokollaa tukeva konkreettinen protokolla. IIOP on TCP/IP-protokollan päällä toimiva konkreettinen GIOP-protokollan mukainen protokolla. IIOP määrittelee, miten GIOP-viestejä vaihdetaan TCP/IP-yhteyden välityksellä. GIOP-protokollan lisäksi CORBA varautuu *ympäristökohtaisten protokollien (Environment-Specific Inter-ORB Protocol, ESIOP)* käyttöön [15]. Ympäristökohtaisten protokollien avulla ORBit voivat kommunikoida esimerkiksi jonkin toisen hajautetun järjestelmän kommunikaatioprotokollan mukaisesti. Tuorein CORBA-määritys sisältää yhden ympäristökohtaisen protokollan (DCE ESIOP).

ORBit voivat sijoita erilaisilla *toimialueilla (domain)*, joiden sisällä voi olla määritetty omia käytäntöjä esimerkiksi olioviitteille, turvallisuudelle ja transaktioille. Jos ORBit sijaitsevat samalla toimialueella, ne voivat kommunikoida suoraan keskenään. Kun palvelukutsuja välitetään eri toimialueilla sijaitsevien ORBien välillä, on kutsujen sisältämä informaatio muutettava toimialueen käytäntöjen mukaiseksi. Nämä muunnokset suoritetaan ORBien välisten siltojen avulla [15]. Siltoja voidaan hyödyntää myös yhteentoimivuuden saavuttamiseksi CORBAn ulkopuolisten järjestelmien kanssa.

Hajautettuihin olioihin perustuvassa järjestelmän hajautuksessa keskeisessä roolissa ovat olioviitteet, joiden avulla asiakkaat voivat kutsua etäolioiden palveluja. Olioviitteitä voidaan välittää toisten olioiden kutsujen parametreina ja asiakas saa olioviitteitä käyttöönsä yleensä kutsujen tulosparametreina [15]. Olioviitteen on sisällettävä olion yksiselitteisesti yksilöivät tiedot. ORB huolehtii olioviitteiden muodostamisesta edus-

tajalta tai rungolta saamiensa tietojen avulla. Jotta ORB osaisi tulkita toisen ORBin välittämän olioviitteen oikein, on olioviitteille oltava yhteinen esitystapa. *IOR (Interoperable Object Reference)* on CORBAn olioviitteen esitystapa, joka määrittelee ORBien välillä välitettävien olioviitteiden sisältämät tiedot [15]. IOR-olioviitteen tärkeimmän osan muodostaa profiili (*tagged profile*), joka sisältää kaikki olion kutsumisessa tarvittavat tiedot. Profiilin tiedoissa määritellään käytettävä ORBien välinen protokolla ja protokollan tarvitsemat tiedot, kuten IIOP-protokollassa palvelimen IP-osoite ja portti. Lisäksi profiili sisältää olion palvelimella identifioivan tunnisteen sekä valinnaisia kutsun suoritusta tukevia tietoja. IOR voi sisältää myös useita profiileja, jos oliopalvelin tukee useita eri protokollia.

5.4.3 OMG CCM

CORBAn oliomalli ja hajautukseen liittyvät palvelut mahdollistivat sovellusten yhdistämisen monimutkaisiksi hajautetuiksi järjestelmiksi. CORBAn oliomalli ei kuitenkaan tarjoa riittävästi tukea hajautettujen olioiden kehitykseen, mikä pakottaa hajautettujen olioiden kehittäjät käyttämään omia tapauskohtaisia ratkaisujaan olioiden toteutuksessa [29]. Tapauskohtaiset ratkaisut johtavat tiukasti kytkettyihin olioiden toteutuksiin ja vaikeuttavat olioiden suunnittelua, uudelleenkäyttöä, käyttöönottoa, ylläpitoa ja laajentamista. Wang ym. [29] erittelevät viisi puutetta perinteisessä CORBAn oliomallissa, jotka hankaloittavat hajautettujen olioiden kehitystä:

- Olioiden käyttöönotto. Perinteinen oliomalli ei määrittele, miten oliot otetaan käyttöön palvelinprosesseissa. Olion käyttöönotto tarkoittaa olion toteutuksen jakelua, toteutuksen asennusta suoritusympäristöön sekä toteutuksen aktivointia ja liittämistä ORBiin.

- Rajoitettu tuki yleisille CORBA-palvelimen ohjelmointimalleille. CORBA sisältää paljon palvelimien ohjelmoinnissa tarvittavia ominaisuuksia ja niiden käytön omaksuminen on haastavaa. Esimerkiksi POA-oliosovitin tarjoaa laajan ohjelmointirajapinnan, jota on sovellettava tapauskohtaisesti olioita ohjelmoitaessa.

- Rajoittunut olioiden toiminnallisuuden laajentaminen. Olioiden toiminnallisuutta voidaan laajentaa ainoastaan perinnän avulla. Uuden rajapinnan lisäämiseksi täytyy määritellä OMG IDL -kielen avulla uusi rajapinta, joka periytyy kaikista vaadituista rajapinnoista, toteuttaa uusi rajapinta ja ottaa uusi toteutus käyttöön kaikissa palvelimissa. Rajapintojen moniperintä on kuitenkin haavoittuvaa, koska CORBA ei tue rajapintojen ylikuormitusta.

- Yleisten CORBA palveluiden saatavuus ei ole tiedossa ennen sovellusten suoritusta. Sovellusten on etsittävä ajonaikaisesti saatavilla olevia palveluja.

- Ei standardoitua olioiden elinkaarien hallintaa. Yleiset palvelut sisältää elinkaaren hallitsemispalvelun, mutta sen käyttö ei ole pakollista. Elinkaaren hallitsemispalvelun käyttö edellyttää olioilta myös erillisten rajapintojen toteutusta.

CORBA Component Model (CCM) [18] -määritys laadittiin paikkaamaan edellä lueteltuja puutteita CORBAn oliomallissa. CCM laajentaa CORBAn oliomallia ominaisuuksilla ja palveluilla, jotka mahdollistavat usein käytettävien CORBAn yleisten palveluiden yhdistävien komponenttien toteuttamisen, hallinnan, konfiguroinnin ja käyttöönoton standardoidussa ympäristössä [29].

CCM-komponentille voidaan määritellä erilaisia rajapintoja *porttien* avulla. Portit kuvataan OMG IDL -rajapinnankuvauskielen avulla ja ne voidaan jakaa neljään eri luokkaan: *aspektit (facet), liittimet (receptacle), tapahtumalähteet- ja nielut (event source/sink) ja attribuutit* [18]. Aspektien avulla määritellään komponentin tarjoamat rajapinnat ja niiden avulla komponentista voidaan esittää erilaisia näkökulmia komponentin käyttäjälle. Liittimet määrittelevät komponentin vaatimat rajapinnat ja niiden avulla komponentti liitetään muihin ohjelmarakenteisiin. Tapahtumalähteet- ja nielut määrittelevät komponentin tuottamat ja kuluttamat tapahtumailmoitukset. Attribuuttien avulla määritellään komponenttien asetukset.

CCM-komponenttiin viitataan olioviitteellä, joka tukee komponentin *ekvivalenttirajapintaa (equivalent interface)* [18]. Ekvivalenttirajapinta julkaisee komponentin ulkoisen rajapinnan asiakkaille ja mahdollistaa komponentin aspektien navigoinnin. Komponentin elinkaarta puolestaan hallitaan *kotirajapinnan (home interface)* kautta [18]. Jokaisella komponentilla on oma koti, jonka avulla voidaan hallita komponentin elinkaarta eli esimerkiksi luoda ja tuhota komponentin ilmentymiä. Asiakas saa komponentin käyttöönsä paikantamalla ensin komponentin kotirajapinnan ja luomalla sen kautta komponentin ilmentymän. Komponentin kotirajapinnan paikantamisessa voidaan hyödyntää keskitettyä tietokantaa, johon päästään käsiksi *HomeFinder-rajapinnan* avulla.

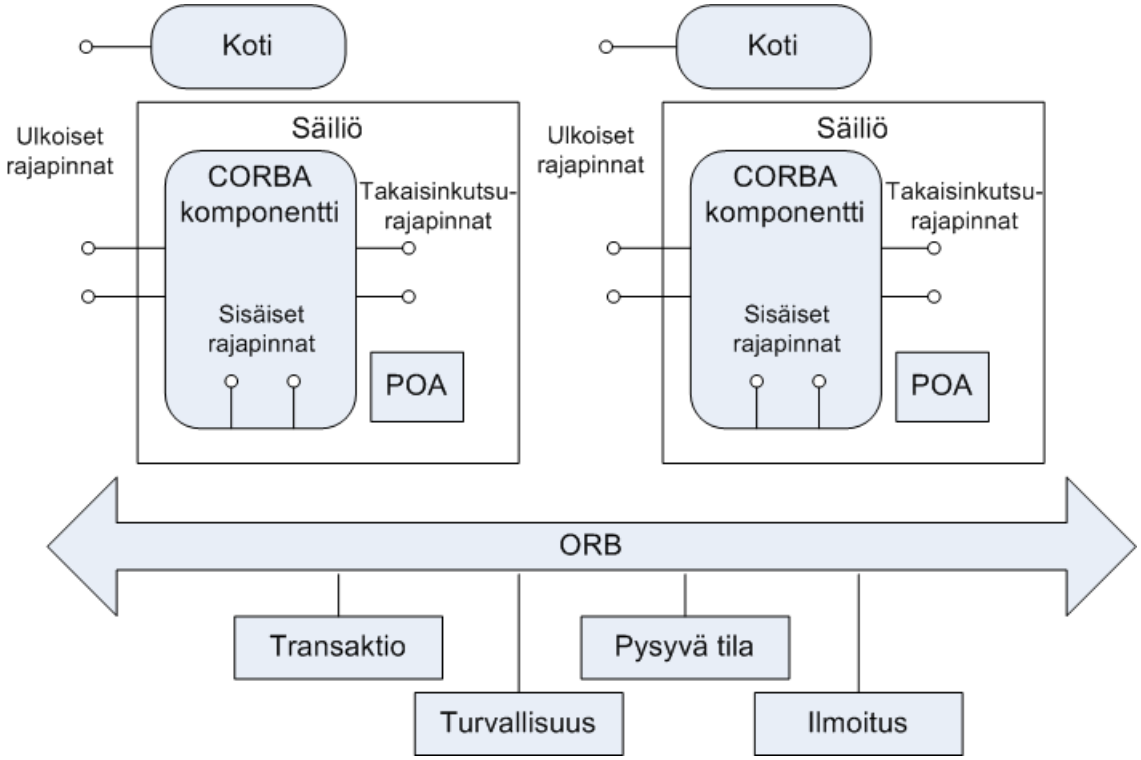
Komponenttien asetusten ja komponenttien liittimien sekä tapahtumalähteiden- ja nielujen kytkentöjen laatimisessa komponenttipalvelimilla voidaan hyödyntää CCM:n määrittelemää komponentin konfigurointirajapintaa (*StandardConfigurator*) [18]. Komponentin home-rajapinnalle voidaan antaa myös viite konfigurointiolioon, joka toteuttaa komponentin asetusten määrityksen. Konfigurointiolio voi tutkia CCM-komponenttien omininaisuuksia komponenttien itsehavannointirajapintojen avulla ja muodostaa komponentin kytkennät muihin komponentteihin sekä ORB-palveluihin [29].

CORBA Component Implementation Framework (CIF) automatisoi komponenttien toteutusta ja tarjoaa komponenttien elinkaaren sekä tilan hallinnan toteutukseen tarkoitetun ohjelmointirajapinnan [29]. Komponentin toteutus sekä komponentin ja sen kodin pysyvä tila voidaan määritellä *CIDL (Component Implementation Definition Language)* -kielen avulla [18]. CIF sisältää CIDL-kääntäjän, joka kääntää CIDL-kielellä annetuista määrityksistä komponentin ydintoiminnot automatisoivia ohjelmointirunkoja. Automatisoitaviin ydintoimintoihin kuuluvat esimerkiksi rajapintojen navigointi, identiteettikyselyt, aktivointi ja tilan hallinta. CIDL-kielellä annetuista määrityksistä käännettyjä rakenteita kutsutaan *toimeenpanijoiksi (executor)* ja ne voidaan pakata *kokoonpanotiedostoiksi (assembly file)*, jotka asennetaan kohdealustaa- sekä ohjelmointikieltä tukevaan *komponenttipalvelimeen* [18]. CIDL-määrityksen pohjalta luodaan myös komponentin *kuvaukset (descriptor)*, jotka määrittelevät komponentin ominaisuudet, kuten tiedot komponentin tarjoamista ja vaatimista rajapinnoista, säikeistyksestä sekä transaktioista [29].

CCM-komponentteja hyödyntävien sovellusten kehityksen helpottamiseksi komponentit kapseloidaan *säiliöihin (container)*, jotka tarjoavat kapseloimilleen komponenteille niiden toimintaa tukevan ajonaikaisen ympäristön. Kuvassa 5.6 esitetty *CCM-säiliöohjelmointimalli (container programming model)* tarjoaa CCM-komponentteja hyödyntävien sovellusten ohjelmointia tukevia ohjelmointirajapintoja [18]. Kuvan ulkoisilla rajapinnoilla tarkoitetaan rajapintoja, joiden kautta komponentin asiakkaat käyttävät komponenttia. Sisäisten rajapintojen kautta komponentti kutsuu säiliön tarjoamia palveluja ja takaisinkutsurajapintojen avulla säiliö kutsuu komponenttia välittäessään esimerkiksi yleisten palveluiden tuottamia tapahtumailmoituksia komponentille.

Asiakkaiden pyynnöt välitetään komponenttien toteutuksille CORBA POA -oliosovittimen kautta. Komponentin kehittäjän ei kuitenkaan tarvitse muodostaa POA-hierarkiaa itse vaan CCM-toteutus muodostaa POA-hierarkian ja paikantaa yleiset palvelut automaattisesti komponentin kuvauksen perusteella [29]. Säiliö välittää komponentin asiakaspyyntöjä yleisille CORBA palveluille, toteuttaa säiliön ja ORBin suorittamien komponentin takaisinkutsujen vaatimat sovitinkerrokset ja hallitsee komponenttiviitteiden muodostamisessa tarvittavia POA-käytäntöjä [29]. Säiliöt hallitsevat myös komponentin toteutuksesta luotavien ilmentymien elinkaarta ottamalla vastuun komponentin aktivoinnista ja deaktivoinnista.

Jotta komponentteja voitaisiin sijoittaa erilaisiin alustoihin hajautetussa ympäristössä ja kuvata komponentin riippuvuudet, täytyy komponenttien kehittäjille tarjota komponentin paketointia ja jakelua tukevia menetelmiä. CCM-komponentti ja sen riippuvuudet voidaan esittää *Open Software Description (OSD)* XML DTD -määrittelyn mukaisella XML-dokumentilla [29], joka kuvaa komponentin kokoonpanotiedoston si-



Kuva 5.6: CCM-säiliöohjelmointimalli. [29]

sällön ja riippuvuudet. Komponentti voi olla esimerkiksi riippuvainen toisista komponenteista, joiden on sijaittava komponentin kanssa samassa muistiavaruudessa.

6 Tuoterunko hajautetussa ympäristössä

Tässä luvussa käsitellään tuoterungon muodostamista hajautettuun ympäristöön. Luku esittelee yleisen hajautetun tuoterungon arkkitehtuurimallin ja tarkastelee mallin avulla hajautetun tuoterungon perustamista. Luvussa 7 arkkitehtuurimallia hyödynnetään tutkielman esimerkkituoterungon muodostamisessa.

Luvussa 6.1 tehdään ensin rinnastus hajautetun tuoterungon ja väliohjelmistojen välille. Luvussa 6.2 esitellään tuoterungon arkkitehtuurimalli ja luvussa 6.3 käsitellään mallin mukaisen tuoterungon perustamista.

6.1 Hajautettu tuoterunko ja väliohjelmistot

Hajautettu tuoterunko on hajautetussa ympäristössä toimiva tuoterunko, jonka sisältämiä variaatiopisteitä voidaan täydentää hajautetun järjestelmän eri solmuissa sijaitsevien komponenttien avulla. Hajautetun tuoterungon avulla muodostetun tuoteperheen jäsenet ovat hajautetussa ympäristössä toimivia hajautettuja sovelluksia. Hajautetun tuoterungon muodostamiseen pätevät samat luvussa 2 esitetyt periaatteet kuin perinteisen tuoterungon muodostamiseenkin, minkä lisäksi hajautettu ympäristö asettaa omat lisävaatimuksensa tuoterungon muodostamiselle.

Hajautetun tuoterungon ympäristön asettamien vaatimuksien täyttämisessä voidaan hyödyntää väliohjelmistoja. Luvussa 5 esitettiin, kuinka väliohjelmiston avulla voidaan hallita hajautetun ympäristön heterogeenisyyttä ja toteuttaa hajautetun järjestelmän läpinäkyvyysvaatimuksia. Väliohjelmistot ovat alustoja, jotka piilottavat sovelluskerroksen alla sijaitsevien kerroksien heterogeenisyyden sovelluksilta ja mahdollistavat erillisten sovelluksien yhdistämisen hajautetuksi järjestelmäksi.

Väliohjelmistojen tarjoamia palveluita voidaan siis käyttää hajautetun tuoterungon muodostamisen apuna, mutta väliohjelmistoja ei voida kuitenkaan suoraan rinnastaa hajautettuun tuoterunkoon. Tuoterungolla on aina määrätty sovellusalue ja tuoterunko tarjoaa sen avulla muodostettaville sovelluksille yhteisen arkkitehtuurin ja perusrakenteen. Väliohjelmistot puolestaan tarjoavat sovelluksille ohjelmistoväylän (ORB), jonka kautta sovellukset voivat kommunikoida keskenään. Väliohjelmistojen päälle toteutetut palvelut, kuten luvussa 5 esitellyn Schantzin ym. [28] kerrosmallin toimialuekohtaisia palveluja tarjoava väliohjelmistokerros tuovat väliohjelmistot lähemmäs hajautetun tuoterungon käsitettä. Ratkaiseva ero hajautetun tuoterungon ja väliohjelmiston välille

syntyy kuitenkin siitä, että väliohjelmistot eivät tarjoa sovelluksille tuoterunkoarkkitehtuurin kaltaista yhteistä arkkitehtuuria.

6.2 Hajautetun tuoterungon arkkitehtuurimalli

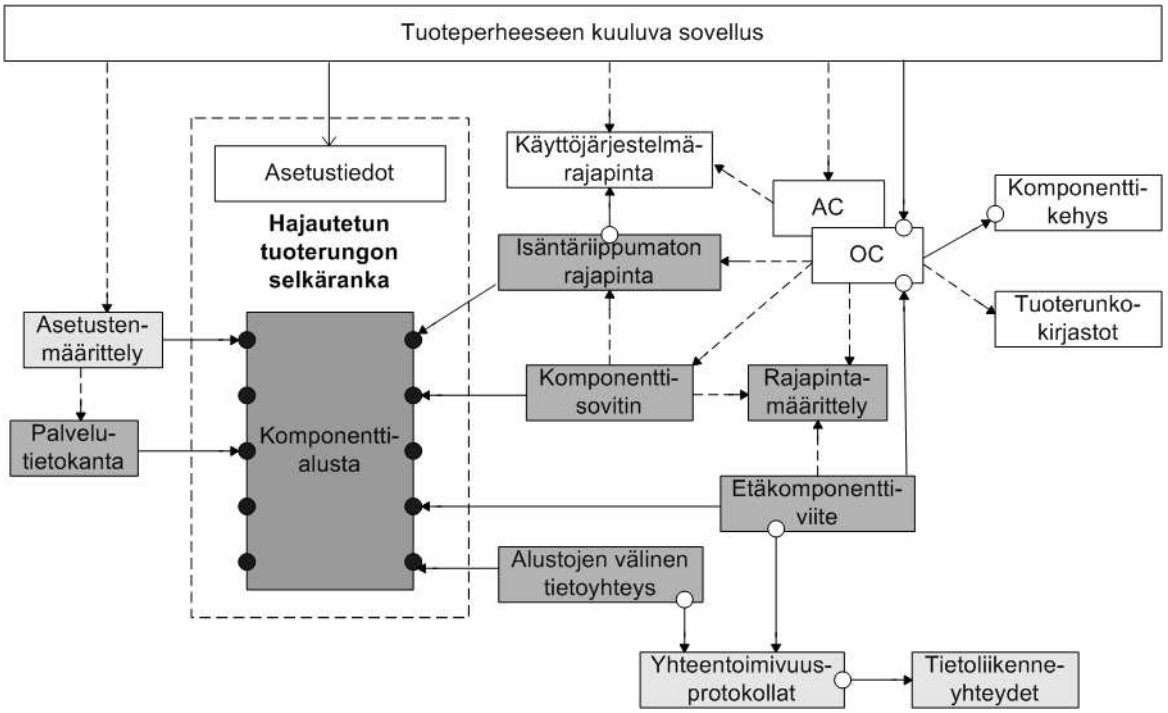
Jotta tuoterungon avulla voitaisiin muodostaa hajautettuja sovelluksia, täytyy tuoterungon täyttää hajautetun ympäristön asettamat vaatimukset. Hajautetun ympäristön asettamat vaatimukset on otettava huomioon tuoterunkoarkkitehtuurissa ja tuoterungon komponenttien toteutuksessa. Tässä luvussa esitellään yleinen hajautetun tuoterungon arkkitehtuurimalli, jota voidaan käyttää referenssimallina hajautettua tuoterunkoa suunniteltaessa. Malli keskittyy tuoterungon hajautuksen käsittelyyn ja sen tarkoitus on lähinnä esittää hajautetulta tuoterungolta vaadittavat peruselementit ja niiden ominaisuudet, puuttumatta elementtien tarkkaan toteutukseen.

Luku 6.2.1 esitelee ensin arkkitehtuurimallin ja luvuissa 6.2.2, 6.2.3, 6.2.4, 6.2.5 ja 6.2.6 käsitellään tarkemmin mallin sisältämiä osa-alueita.

6.2.1 Arkkitehtuurimalli

Luvussa 3.4 esiteltiin Parsonsin ym. [20] komponenttipohjaisen ohjelmistokehyksen arkkitehtuurimalli ja rinnastettiin malli tuoterungon toteutukseen. Parsonsin ym. mukaan heidän mallinsa ydinominaisuuksiin ei kuulu heterogeenisessä ja hajautetussa ympäristössä toimivien kehysten suunnittelun tukeminen [20]. Malli ei kuitenkaan vaadi suuria rakenteellisia muutoksia, jotta sitä voitaisiin soveltaa hajautettuun ympäristöön. Hajautetun ympäristön asettamat erityisvaatimukset voidaan ottaa huomioon Parsonsin ym. mallin selkärangan ja peruskomponenttien avulla. Tässä luvussa Parsonsin ym. mallin pohjalta muodostetaan yleinen hajautetun tuoterungon arkkitehtuurimalli, joka on esitelty kuvassa 6.1.

Kuvassa 6.1 on käytetty pääosin samoja merkintöjä kuin kuvassa 3.4. Kuvasta 3.4 poiketen kuvassa 6.1 on kuitenkin määritelty kolme erityyppistä tuoterungon osa-alueita kuvaavaa elementtiä. Kuvan 6.1 tummanharmaa laatikko esittää tuoterungon hajautuksen toteutuksen kannalta pakollista elementtiä ja vaaleanharmaa laatikko hajautuksen toteutuksessa käytettävää vaihtoehtoisten komponenttien joukkoa, josta vähintään yhden komponentin on kuuluttava mukaan tuoterungon avulla muodostettavaan ohjelmistoon. Vaaleat laatikot kuvaavat tuoterungon käyttöä ja rungon komponenttien kehitystä tukevia elementtejä sekä tuoterungon valinnaisia (OC) ja sovelluskohtaisia (AC) komponentteja. Lisäksi vaalealla laatikolla on kuvattu tuoterungon toimintaympäristöön kuuluvia kokonaisuuksia.



Kuva 6.1: Hajautetun tuoterungon arkkitehtuurimalli.

Malli on pyritty muodostamaan siten, että se tekee mahdollisimman vähän oletuksia ympäristöstään ja lopullisesta tuoterunkoarkkitehtuurista. Malli ei siis kiinnitä lopullista tuoterunkoarkkitehtuuria vaan pyrkii ainoastaan esittelemään ominaisuuksia, joita hajautetussa ympäristössä toimivalta tuoterungolta vaaditaan. Parsonsin ym. [20] mallin mukaisesti mallin ytimen muodostaa selkäranka, jonka laajennoskohtia täydentämällä sovellukset muodostetaan. Pakolliset ja vaihtoehtoiset elementit toteuttavat hajautuksen vaatimat mekanismit ja valinnaiset komponentit toteuttavat tuoterungon lopullisella sovellusalueella tarvittavat palvelut. Valinnaiset komponentit hyödyntävät hajautetun ympäristön hallinnassa pakollisten elementtien tarjoamia palveluita. Sekä selkärangan että komponenttien erikoistamisessa voidaan käyttää myös oman solmun ulkopuolisia etäkomponentteja.

Mallin selkärangan muodostavat *komponenttialusta* sekä komponenttialustan asetukset määrittelevä metadata (*asetustiedot*). Tuoterungon hajautuksen kannalta pakollisiin elementteihin kuuluvat *isäntäriippumaton rajapinta*, *komponenttisovitin*, *rajapintamäärittely*, *etäkomponenttiviite*, *alustojen välinen tietoyhteys* sekä *palvelutietokanta*. *Asetusten määrittely*, *yhteentoimivuusprotokolla* ja *tietoliikenneyhteys* muodostavat kukin oman vaihtoehtoisten komponenttien joukkonsa. Muodostettavan ohjelmiston toimintaympäristön *käyttöjärjestelmärajapinta* erikoistaa isäntäriippumattoman rajapinnan käyttämään käyttöjärjestelmän palveluja.

6.2.2 Selkäranka

Selkäranka on mallin ydinelementti, jonka muodostavat tuoterunkoarkkitehtuurin mukainen komponenttialusta ja alustan toimintaa ohjaavat asetustiedot. Komponenttialusta voidaan mieltää luvussa 2.1.2 esitetyn kaltaiseksi, kaikkiin tuoterungon avulla muodostettaviin ohjelmistoihin sisältyväksi ohjelmistoalustaksi. Komponenttialustan toteutus on koottava sovelluskehys, jonka sisältämät laajennoskohdat muodostavat tuoterungon variaatiopisteet. Jokaisessa laajennoskohdassa on määritelty erikoistamisessa vaadittava rajapinta. Malli ei ota kantaa komponenttialustan rakenteeseen eikä laajennoskohtien rajapintojen sisältöön.

Komponenttialustan toimintaa voidaan ohjata asetustietojen avulla, joissa määritellään alustaan liitettyjen komponenttien tiedot. Asetustietojen on sisällettävä tiedot erityyppisten komponenttien aktivoinnissa ja kommunikoinnissa käytettävistä komponenttisovittimista. Asetustiedot voivat sisältää myös komponenttisovittimen komponenttien aktivoinnissa tarvitsemia tietoja. Komponenttialustaan liitettävä komponentti voi olla myös toisen komponenttialustan hallinnoima etäkomponentti, joka määritellään etäkomponenttiviitteen avulla. Komponenttialusta huolehtii etäkomponenttiviitteiden välittämisestä muille komponenttialustoille.

6.2.3 Alustojen välinen yhteensopivuus

Komponenttialusta kommunikoi muiden komponenttialustojen kanssa alustojen välisen tietoyhteyden kautta. Alustojen välisen tietoyhteyden toiminta erikoistetaan yhteentoimivuusprotokollan avulla. Tietoyhteys voi tukea useita eri yhteentoimivuusprotokollia. Yhteentoimivuusprotokolla määrittelee komponenttialustojen välisen yhteyskäytännön sekä alustojen välillä välitettävien viestien ja datan esitysmuodon. Protokolla määrää myös komponenttialustojen välillä välitettävien etäkomponenttiviitteiden muodon. Yhteentoimivuusprotokolla vaatii toimiakseen tietoliikenneyhteyden, jonka päällä se toimii. Yhteentoimivuusprotokolla voi tukea useita erilaisia tietoliikenneyhteyksiä.

Ainoa edellytys kahden komponenttialustan väliselle yhteensopivuudelle on, että ne noudattavat kommunikoinnissaan samaa yhteentoimivuusprotokollaa. Komponenttialusta voidaan liittää myös tuoterungon ulkopuolisiin järjestelmiin toteuttamalla kyseisen järjestelmän kommunikaatiotapaa noudattava yhteentoimivuusprotokolla.

6.2.4 Komponenttien integrointi

Hajautetut komponentit integroidaan komponenttialustaan komponenttisovittimien kautta. Komponenttisovitin, yhdessä rajapintamäärittelyn kanssa, erottaa komponenttia-

lustan ja komponentin toisistaan siten, että alustan toteutuksessa ei tarvitse ottaa huomioon alustaan liitettävien komponenttien toteutustapaa ja toisaalta siten, että komponenttien toteutustapa ei ole riippuvainen komponenttialustan toteutustavasta.

Rajapintamäärittely erottaa komponentin rajapinnan komponentin toteutuksesta. Rajapintamäärittelyn avulla julkistetaan komponentin tarjoamat, vaatimat ja konfigurointirajapinnat. Tuoterungon on määritettävä tapa, jolla komponentin rajapinnat määritetään sekä toteutettava rajapintamäärittelyjen muodostamisessa ja käytössä tarvittavat mekanismit. Myös etäkomponenttiviitteet perustuvat rajapintamäärittelyyn. Etäkomponenttiviite toteuttaa määrittelemänsä rajapinnan välittämällä rajapintakutsut rajapinnan varsinaisesti toteuttavalle komponentille, joka voi sijaita jossakin toisessa järjestelmän solmussa.

Kuten luvuissa 4.3.1 ja 4.4.2 esitettiin, hajautetut komponentit voidaan jakaa kahteen luokkaan niiden aktivoimistavan mukaan. Pysyvä komponentti on komponentti, joka voidaan tallentaa pysyvään muistiin ja aktivoida sieltä tarvittaessa. Myös pysyvän komponentin tila voidaan tallentaa pysyvään muistiin ja palauttaa aktivoinnin yhteydessä. Tilapäinen komponentti on puolestaan komponentti, joka on olemassa vain ohjelman suoritusaikana prosessin muistiavaruudessa. Komponenttisovittimet huolehtivat sekä pysyvien että tilapäisten komponenttien aktivoinnista sekä tiedonvälityksestä komponenttialustalta komponenteille ja komponenteilta takaisin alustalle. Käytännössä komponentin aktivointi voi vaatia esimerkiksi uuden prosessin käynnistämistä tai jaetun kirjaston ajonaikaista linkitystä prosessin muistiavaruuteen ja komponentin ilmentymän luomista kirjastosta.

Komponentin aktivointiin sisältyy myös luvussa 4.4.1 käsitelty suoritussäikeen osoitus komponentille. Komponenttisovitin voi toteuttaa suoritussäikeen osoituksen komponentille esimerkiksi jollain luvussa 4.4.1 esitetyllä tavalla. Tiedonvälitys alustan ja komponenttien välillä puolestaan edellyttää, että komponenttisovitin toteuttaa tarvittavat kommunikointimekanismit alustan ja komponenttien suoritussäikeiden välillä.

6.2.5 Alustariippumattomuus

Isäntäriippumaton rajapinta tarjoaa tuoterungon avulla muodostetun ohjelmiston isäntäalustoista, eli laitteistoista ja käyttöjärjestelmistä, riippumattoman rajapinnan tuoterungon muille elementeille. Isäntäriippumattoman rajapinnan toteutuksen erikoistamisessa eri käyttöjärjestelmien päällä toimivaksi hyödynnetään käyttöjärjestelmien tarjoamia *ohjelmointirajapintoja (application programming interface, API).*

Isäntäriippumattoman rajapinnan tulisi erottaa muut tuoterungon elementit alustariippuvaisista rajapinnoista ja sen toteutus määrittelee alustat, joiden päällä tuoterun-

gon avulla muodostetut ohjelmistot voivat toimia. Kaikki tuoterungon elementit, jotka hyödyntävät toteutuksessaan ainoastaan isäntäriippumatonta rajapintaa ovat siirrettäviä isäntäriippumattoman rajapinnan tukemien alustojen välillä. Kuten kuvasta 6.1 nähdään, tuoterunko voi myös sisältää komponentteja, jotka käyttävät suoraan esimerkiksi käyttöjärjestelmärajapinnan tarjoamia palveluja ja joiden siirrettävyys on siten rajoitettua.

6.2.6 Palvelutietokanta

Palvelutietokanta sisältää tiedot järjestelmään tai järjestelmän osan muodostavaan toimialueeseen kuuluvista komponenteista. Tietokanta linkittää komponenttien rajapintojen kuvaukset komponenttien toteutuksiin. Linkitys tapahtuu etäkomponenttiviitteiden avulla.

Jokaisella järjestelmän komponenttialustalla tulee olla yhteys tietokantaan, johon se rekisteröi omien komponenttiensa tiedot sekä saa tietoja muiden järjestelmään kuuluvien komponenttialustojen hallinnoimista komponenteista. Tietokanta voidaan toteuttaa etäpalvelimella toimivana keskitettynä palveluna, johon kaikki järjestelmän komponenttialustat rekisteröivät tiedot omista komponenteistaan ja saavat tietoja muiden komponenttialustojen hallinnoimista komponenteista. Tietokanta voi olla myös hajautettu esimerkiksi siten, että jokainen komponentti pitää yllä tietokantaa omista komponenteistaan ja komponenttialustat synkronoivat tietoja hallinnoimistaan komponenteista yhteisen protokollan mukaisesti.

Asetusten määrittely kuvaa yleisesti tuoterungon käyttöä tukevia työkaluja. Työkalujen avulla voidaan helpottaa tuoterunkoa hyödyntävän ohjelmiston muodostamista ja ohjelmiston asetusten määritystä. Työkalujen avulla tuoterungon käyttäjä voi olla yhteydessä rungon palvelutietokantaan ja lisätä palvelutietokannan tietojen avulla uusia komponentteja ohjelmistoon sekä määrittää komponenttien välisiä yhteyksiä.

6.3 Hajautetun tuoterungon perustaminen

Hajautetun tuoterungon perustamiseen pätevät samat luvussa 2.2 esitetyt periaatteet kuin ei-hajautetun tuoterungonkin perustamiseenkin. Lisäksi hajautetun tuoterungon perustamisessa on otettava huomioon hajautetun ympäristön asettamat vaatimukset tuoterungolle. Tämä luku käsittelee tuoterungon hajautuksen huomioimista tuoterungon perustamisessa ja hajautuksen hallinnan toteutusta.

Luvussa 6.3.1 tarkastellaan, miten tuoterungon hajautus tulee huomioida rungon perustamisprosessin eri vaiheissa. Luku 6.3.2 käsittelee tuoterungon hajautuksessa tar-

vittavien elementtien toteutusta. Lopuksi luvuissa 6.3.3 ja 6.3.4 arvioidaan väliohjelmistojen hyödyntämistä tuoterungon toteutuksessa ja hajautuksen vaikutusta tuoterungon perustamistapaan.

6.3.1 Hajautuksen huomioiminen perustamisprosessissa

Hajautetun tuoterungon perustamisprosessin aloittavassa sovellusalueanalyysissa- ja määrittelyssä on otettava huomioon tulevan tuoteperheen jäsenten kommunikaatioarkkitehtuurin yhtäläisyydet ja eroavaisuudet. Erilaiset vaatimukset kommunikaatioarkkitehtuurin suhteen voivat aiheuttaa hyvinkin suuria eroja tuoteperheen jäsenien välille.

Luvussa 6.2 esitetyn arkkitehtuurimallin selkäranka ja alustojen välinen tietoyhteys eivät kiinnittäneet mallia tiettyyn kommunikaatioarkkitehtuuriin ja mahdollistivat myös useamman kommunikaatioarkkitehtuurin rinnakkaisen käytön. Käytännössä useampaa rinnakkaista kommunikaatioarkkitehtuuria hyödyntävän tuoterunkoarkkitehtuurin kehittäminen voi kuitenkin olla hyvin haasteellista ja tuoterunko on perustettava tietyn kommunikaatioarkkitehtuurin varaan. On myös mahdollista, että ohjelmistojen erilaisia vaatimuksia kommunikaatioarkkitehtuurin suhteen ei voida järkevästi täyttää rungon variaatiopisteiden avulla, jolloin sovellusalue ei ole tarpeeksi yhtenäinen tuoterungon perustamiselle. Esimerkiksi tilapäisen synkronisen kommunikaation ja pysyvän asynkronisen kommunikaation vaatimukset saattavat joissain tapauksissa olla liian etäällä toisistaan, jotta vaatimuksia kannattaisi yrittää kattaa rungon variaatiopisteiden avulla.

Sovellusalueanalyysissa- ja määrittelyssä on huomioitava myös hajautetun tuoterungon ympäristön heterogeenisyys. Heterogeenisyyden analysointi voi helpottaa huomattavasti tulevaa kehitysprosessia selvittämällä tuoterungon ympäristössä rungolta todellisuudessa vaadittavat ominaisuudet ympäristön heterogeenisyyden hallinnan suhteen. Jos esimerkiksi tuoterungon hajautettu toimintaympäristö sisältää ainoastaan homogeenisia isäntäalustoja ja kaikki komponentit toteutetaan samalla ohjelmointikielellä, tuoterungon toteutus helpottuu huomattavasti. Hajautetun tuoterungon ei ole aina järkevää yrittää kattaa mahdollisimman laajaa ja heterogeenistä ympäristöä, koska se voi vaatia huomattavan panostuksen sellaiseen työhön, jota rungon sovellusalueella ei todellisuudessa vaadita.

Kommunikaatioarkkitehtuurilla on keskeinen rooli tuoterungon arkkitehtuurisuunnittelussa. Arkkitehtuurisuunnittelussa on suunniteltava sellainen arkkitehtuuri, joka mahdollistaa kaikkien tuoteperheen ohjelmistojen vaatimien kommunikaatiotapojen toteuttamisen. Kuten edellä todettiin, useita rinnakkaisia kommunikaatioarkkitehtuureja sisältävän tuoterunkoarkkitehtuurin kehittäminen voi olla hyvin haastavaa ja arkki-

tehtuurissa voidaankin joutua tekemään kompromisseja, jos ohjelmistoilla on toisistaan etäällä olevia vaatimuksia kommunikaatiotapojen suhteen. Yksi vaihtoehto on perustaa arkkitehtuuri tietyn kommunikaatiotavan varaan ja toteuttaa kommunikaatioon liittyvät variaatiopisteet valitun kommunikaatiotavan päälle. Arkkitehtuuri voi esimerkiksi perustua tilapäiseen synkroniseen kommunikaatioon, jonka päälle voidaan toteuttaa asynkronisen ja pysyvän kommunikaation variaatiopisteet. Tällaisten kompromissien kannattavuutta on mietittävä jo sovellusalueanalyysi- ja määrittelyvaiheessa.

6.3.2 Komponenttien suunnittelu ja toteutus

Hajautetun tuoterungon selkärangan ja hajautuksesta huolehtivien elementtien suunnittelu ja toteutus muodostavat tärkeän osan luvun 6.2 mukaisen hajautetun tuoterungon komponenttien suunnittelu- ja toteutusvaiheesta. Selkärangan tulisi toteuttaa tuoterunkoarkkitehtuurin mukainen yhteinen perusrakenne tuoterungon avulla muodostettaville ohjelmistoille ja pakollisten elementtien sekä vaihtoehtoisten komponenttien tulisi peittää tuoterungon hajautettu toimintaympäristö siten, että ympäristöä ei tarvitse huomioida täydentävien komponenttien toteutuksessa.

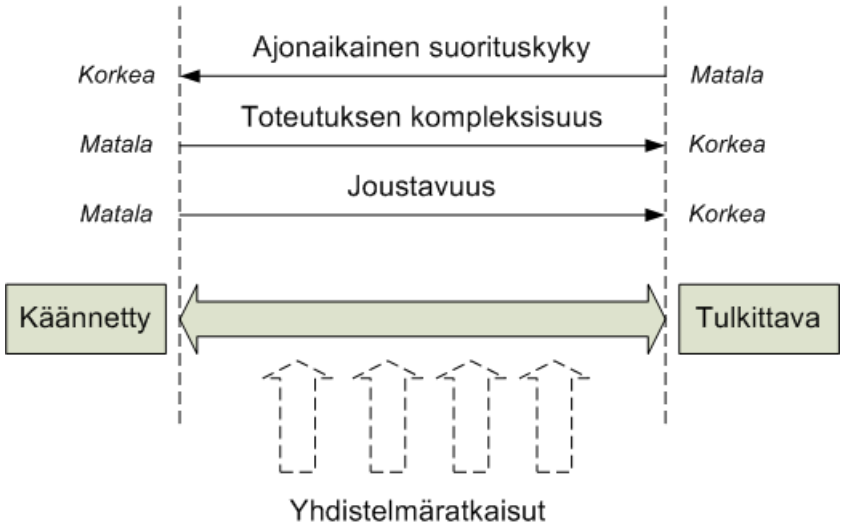
Tuoterungon toimintaympäristön heterogeenisyyden aste vaikuttaa merkittävästi tuoterungon hajautuksesta huolehtivien elementtien suunnittelun ja toteutuksen vaatimaan työmäärään. Työmäärään vaikuttavat erityisesti tuoterungon toimintaympäristön sisältämät isäntäalustat ja tuoterungon valinnaisten komponenttien toteutustapa.

Homogeenisista isäntäalustoista koostuva ympäristö mahdollistaa isäntäriippumattoman rajapinnan toteutuksen poisjättämisen tuoterungosta ja helpottaa alustojen välisten yhteentoimivuusprotokollien kehitystä. Valinnaisten komponenttien kehityksessä käytettävät ohjelmointikielet ja komponenttien toteutusmuoto on otettava huomioon valinnaisten komponenttien kehitystä tukevia pakollisia elementtejä suunniteltaessa. Rajapintamäärittelyjen muodostaminen helpottuu, jos komponenttien toteutuksessa käytetään aina samaa ohjelmointikieltä ja rajapintamäärittelyt voidaan muodostaa tämän ohjelmointikielen avulla.

Tärkeä selkärangan ja pakollisten elementtien toteutukseen vaikuttava päätös on valinnaisten komponenttien toteutusmuodon valitseminen käännettyjen ja tulkittavien ohjelmarakenteiden välillä. Parsons ym. [20] esittävät käännetyn ja tulkittavan toteutusmuodon vaikutusta kehyksensä toteutukseen kuvan 6.2 kaavion avulla. Kaaviossa vasemmalla on esitetty täysin käännetty toteutusmuoto ja oikealla täysin tulkittava toteutusmuoto. Täysin käännetyllä muodolla tarkoitetaan komponenttien mallintamista binaarisina, esimerkiksi jaettujen kirjastojen avulla. Täysin tulkittava muoto puolestaan mallintaa komponentit lähdekoodina, jota kehyksen sisältämä virtuaalikone osaa

tulkita. Yhdistelmäratkaisuilla tarkoitetaan toteutuksia, joissa on hyödennetty sekä käännettyjä että tulkittavia ohjelmarakenteita.

Kuvan 6.2 kaaviosta nähdään, kuinka tulkittava muoto lisää joustavuutta, mutta sen toteutus vaatii monimutkaisia rakenteita ja se heikentää sovelluksien ajonaikaista suorituskykyä. Käännetty muoto on vastaavasti yksinkertaisempi toteuttaa ja sen ajonaikainen suorituskyky on parempi, mutta se ei tarjoa samaa joustavuutta kuin tulkittava muoto. Parsonsin ym. mielestä oikean suhteen löytämisellä käännettyjen ja tulkattavien ohjelmarakenteiden välillä on ratkaisevan tärkeä rooli toteutuksen onnistumisen kannalta.



Kuva 6.2: Käännetty ja tulkittava toteutusmuoto. [20]

6.3.3 Väliohjelmistojen hyödyntäminen

Luvussa 6.1 todettiin, että väliohjelmistoja voidaan käyttää apuna hajautetun tuoterungon toteutuksessa. Väliohjelmistoista on nykyään olemassa monia COTS-toteutuksia, joita lueteltiin esimerkiksi luvussa 5.3. Erityisesti luvussa 5.3 esitellyn Schantzin ym. [28] luokittelun mukainen hajautukseen liittyviä palveluja tarjoava väliohjelmistokerros ja siihen sisältyvät väliohjelmistojen komponenttimallit voivat tarjota hyvän perustan hajautetun tuoterungon kehitykselle. Parhaassa tapauksessa väliohjelmisto tarjoaa kaikki tuoterungon hajautuksen vaatimat elementit, jolloin tuoterungon toteutuksessa voidaan keskittyä selkärangan ja valinnaisten komponenttien toteutukseen.

Vaikka luvun 5 Schantzin ym. [28] luokittelu esittää väliohjelmistojen tarjoamat palvelut kerrosmallina, väliohjelmistot ovat usein kokonaisratkaisuja, joiden tarjoamia palveluja ei voida hyödyntää itsenäisinä palveluina. Tämä tarkoittaa, että tietyn välioh-

jelmiston tarjoaman palvelun käyttäminen kiinnittää myös palvelun alla olevien kerroksien toteutuksen. Esimerkiksi tietyn väliohjelmiston tarjoaman komponenttimallin hyödyntäminen tuoterungon komponenttien toteutuksessa voi vaikuttaa kaikkiin luvun 6.2 arkkitehtuurimallin sisältämiin osa-alueisiin.

Tuoterungon tulee ensisijaisesti tarjota rungon sovellusalueella tarvittava yhteinen arkkitehtuuri ja perusrakenne tuoteperheen jäsenille. Hajautetussa tuoterungossa sovellusalueeseen sisältyy aina hajautetun ympäristön vaatimukset, mutta ne eivät saa syrjäyttää muita sovellusalueen vaatimuksia. Tämän vuoksi väliohjelmistoja hyödyntävän tuoterungon perustamisprosessinkin tulisi alkaa sovellusalueanalyysi- ja määrittelyvaiheella, jossa selvitetään sovellusalueella tuoterungolta vaadittavat ominaisuudet. Väliohjelmiston hyödyntäminen hajautetun tuoterungon toteutuksessa johtaa iteratiiviseen tuoterungon perustamisprosessiin, jossa verrataan väliohjelmiston ominaisuuksia tuoterungolle asetettuihin vaatimuksiin. Väliohjelmistoa valittaessa on huomioitava erityisesti väliohjelmiston tukemat alustat sekä kommunikaatioarkkitehtuurit ja niiden toteutustavat.

6.3.4 Perustamistapa

Hajautetun tuoterungon hajautuksen vaatimat ominaisuudet vaikuttavat voimakkaasti tuoterunkoarkkitehtuuriin ja asettavat omat rajoituksensa myös tuoterungon perustamistavalle. Luvussa 2.2.1 esitettiin tuoterungon perustamistavan valintaan vaikuttavia tekijöitä, kuten tuoterungon kehitykseen käytettävissä olevat resurssit, jo aloitetut tulevaan tuoteperheeseen kuuluvat ohjelmistoprojektit sekä olemassa olevat ohjelmistot, jotka halutaan liittää tuoteperheeseen. Näiden tekijöiden pohjalta esiteltiin neljä eri Boschin [9] luokittelun mukaista tuoterungon perustamistapaa.

Boschin [9] esittämät kumouksellisen lähestymistavan mukaiset tuoterungon perustamistavat ovat tuoterungon toteutuksen kannalta ideaalisia ratkaisuja, koska runko voidaan toteuttaa ilman olemassa olevien ohjelmistojen asettamia rajoitteita. Kumouksselliset perustamistavat voivat kuitenkin olla taloudellisesti mahdottomia vaihtoehtoja organisaatiolle, jolla on rajoitetut resurssit ja kattava pääoma aikaisemmin kehitetyistä ohjelmistoista. Myös jo aloitettuja tulevaan tuoteperheeseen kuuluvien ohjelmistojen kehitysprojekteja voi olla mahdotonta pitkittää muuttamalla lähestymistapaa kesken projektin.

Olemassa olevien ohjelmistojen hyödyntäminen tuoterungon perustamisessa Boschin [9] evolutiivisen lähestymistavan mukaisesti edellyttää, että hyödynnettävän ohjelmiston arkkitehtuuri perustuu modulaarisiin ohjelmarakenteisiin, joilla on selkeästi määritellyt rajapinnat. Tällöin olemassa olevien ohjelmistojen arkkitehtuurien poh-

jalta voidaan muodostaa yhteinen tuoterunkoarkkitehtuuri sekä olemassa olevista ohjelmarakenteista kehittää yleiskäyttöisiä komponentteja ja muodostaa komponenteille tarvittavat rajapintamäärittelyt. Jos komponenteista halutaan siirrettäviä, on ohjelmarakenteet lisäksi muutettava käyttämään tuoterungon isäntäriippumatonta rajapintaa.

Hajautuksesta huolehtivien elementtien toteutus muodostaa työmäärältään huomattavan osan hajautetun tuoterungon toteutuksesta. Hajautuksen toteuttamiseen käytettävä työmäärä tulisi kuitenkin pyrkiä minimoimaan, jotta voidaan keskittyä tuoterungon sovellusalueen muun toiminnallisuuden toteuttamiseen. Tämä kannustaa väliohjelmistojen hyödyntämiseen tuoterungon hajautuksen hallinnassa.

Merkittävä väliohjelmistojen käyttöä puoltava tekijä on myös yhteensopivuus muiden teknologioiden kanssa. Kehittyneiden väliohjelmistojen COTS-toteutuksien ansiosta ohjelmistoja tilaavat asiakkaat voivat yhä useammin vaatia ohjelmistotoimittajilta COTS-toteutuksien käyttöä tilaamiensa ohjelmistojen toteutuksessa. COTS-toteutuksien avulla asiakas voi välttää riippuvuutta yhdestä ohjelmistotoimittajasta tilaamansa tuotteen ylläpidossa sekä jatkokehityksessä ja kehittää tilaamaansa tuotetta myös muiden toimittajien avulla. Nykyään jopa perinteisesti suljetuista ratkaisuistaan tunnetussa sotateollisuudessa on nähtävissä merkkejä siirtymisestä niin sanotuista *MOTS (Military off-the-shelf)* -ratkaisuista kaupallisten, vapaasti saatavien COTS-toteutusten käyttöön. Hyvänä esimerkkinä tästä on Yhdysvaltojen laivaston avoimien arkkitehtuurien käyttöä laivaston sodankäyntijärjestelmissä puoltava ohjelma [7] [6], joka painottaa COTS-toteutuksien käyttöä sekä järjestelmien laitteistojen että ohjelmistojen toteutuksessa.

7 XMPL tuoterunko

Tässä luvussa käsitellään erään organisaation hajautetun tuoterungon muodostamista, keskittyen tuoterungon hajautuksen toteutukseen. Tuoterungon hajautuksen suunnittelun apuna käytetään luvussa 6 esitettyä hajautetun tuoterungon arkkitehtuurimallia. Tuoterungosta käytetään tässä yhteydessä nimitystä XMPL tuoterunko. Luvussa 7.1 määritellään ensin tuoterungon sovellusalue ja luvussa 7.2 esitetään tuoterungon perustamistapa. Luvussa 7.3 esitellään tuoterungon toteutusta ja lopuksi luvussa 7.4 esitetään lyhyt arvio tuoterungosta.

7.1 Sovellusalue

XMPL tuoterungon avulla on tarkoitus muodostaa ohjelmistoja, joiden avulla voidaan kerätä dataa erilaisista datalähteistä sekä analysoida ja simuloida kerättyä dataa. Ohjelmistojen avulla voidaan muodostaa liityntöjä erilaisiin laitteistoihin ja sensoreihin, suorittaa testisekvenssejä laitteistoille sekä analysoida ja simuloida laitteistojen ja sensorien tuottamaa dataa. Datasta ja analyysin tuloksista voidaan esittää erilaisia yhteenvetoja käyttäjille.

Tuoterungon avulla muodostettavat ohjelmistot sijoittuvat järjestelmään, joka koostuu erilaisilla tietoliikenneyhteyksillä toisiinsa kytketyistä solmuista sekä solmuihin liitetyistä erilaisista testattavista laitteistoista ja sensoreista. Järjestelmän solmut voivat sisältää erilaisia laitteistoarkkitehtuureja ja solmujen käyttöjärjestelmänä voi toimia joko Windows tai Linux -käyttöjärjestelmien eri versiot.

Datalähteitä voi olla useita ja niiden tuottaman datan määrä voi vaihdella yksittäisistä tapahtumailmoituksista hyvinkin suuriin datamääriin. Ohjelmistojen on pystyttävä keräämään dataa järjestelmän eri solmuihin liitetyistä datalähteistä sekä hajauttamaan datan analysointi ja tulosten esittäminen järjestelmän eri solmuihin. Myös ohjelmiston käyttöliittymät on pystyttävä hajauttamaan ja datasta sekä analysoinnin tuloksista tehtäviä yhteenvetoja esittämään eri muodoissa järjestelmän eri solmuissa.

Ohjelmiston eri elementtien välisen kommunikaation on oltava asynkronista ja tarvittaessa pysyvää. Järjestelmään, jossa ohjelmistot toimivat, voidaan lisätä uusia solmuja ja siitä voidaan poistaa solmuja ohjelmiston suorituksen aikana. Ohjelmistojen on pystyttävä sopeutumaan järjestelmän kokoonpanomuutoksiin. Ohjelmiston käyttäjä määrittelee sovelluksen eri solmuissa sijaitsevien elementtien väliset yhteydet ja käyt-

täjän on pystyttävä muuttamaan näitä yhteyksiä ajonaikaisesti. Ohjelmistojen suoritusajat voivat olla hyvin pitkiä ja ohjelmistojen toimintaa on pystyttävä päivittämään ajonaikaisesti.

7.2 Perustaminen

Tuoterungon perustava organisaatio oli aiemmin kehittänyt useita tuoterungon sovellusalueella toimivia ohjelmistoja. Aiemmin kehitetyt ohjelmistot perustuivat keskenään samankaltaiseen arkkitehtuuriin ja niiden kehityksessä oli hyödynnetty uudelleenkäytettäviä moduuleja. Ohjelmistojen arkkitehtuurissa oli huomioitu ohjelmistojen hajautus ja toteutettu hajautusta tukevia ohjelmarakenteita. Sekä aiemmin kehitettyjen ohjelmistojen arkkitehtuuria että ohjelmarakenteita haluttiin hyödyntää tuoterungon perustamisessa Boschin [9] evolutiivisen lähestymistavan mukaisesti.

Aiemmin kehitettyjen ohjelmistojen arkkitehtuurit perustuivat ohjelmistojen tehokkaaseen modularisointiin uudelleenkäytettäväksi moduuleiksi sekä moduulien välistä asynkronista kommunikaatiota tukevaan yksinkertaiseen viestinvälitysarkkitehtuuriin. Tuoterunkoarkkitehtuuri päätettiin muodostaa hyvin pitkälle yhdenmukaiseksi aiemman arkkitehtuurin kanssa. Tuoteperheeseen liitettävien ohjelmistojen perusarkkitehtuuri sekä sen toteuttava tuoterungon selkäranka haluttiin pitää aiempien ohjelmistojen tapaan yksinkertaisena ja keskittyä uudelleenkäytettävien moduulien kehityksen tukemiseen. Moduulien välisessä kommunikaatiossa käytetty ja moduulien rajapintojen määrityksen perustana toiminut alustariippumaton viestien esitystapa päätettiin säilyttää myös tuoterungon komponenttien välisten viestien esitystapana ja komponenttien rajapintamäärittelyjen perustana.

Aiemmin kehitettyjen ohjelmistojen ohjelmointikieli oli C++, joka haluttiin säilyttää tuoterungon toteutuksessa käytettävänä ohjelmointikielenä. Väliohjelmistojen käyttö tuoterungon hajautuksen toteutuksessa hylättiin, koska aiemmin kehitetyt ohjelmistot sisälsivät jo paljon hajautuksen toteutuksessa vaadittavia elementtejä, joita haluttiin hyödyntää myös tuoterungon perustamisessa. Hajautuksesta huolehtivat elementit haluttiin kuitenkin erottaa tuoterungossa siten, että niiden korvaaminen väliohjelmiston avulla olisi myöhemmin mahdollista. Aiemmin kehitetyt ohjelmistot toimivat ainoastaan Windows käyttöjärjestelmässä ja niiden toteutuksessa oli hyödynnetty sekä Windowsin käyttöjärjestelmärajapintaa että Microsoftin MFC-kirjastoa. Käyttöjärjestelmäriippumattomuuden toteuttamiseksi tuoterungossa päätettiin ottaa käyttöön isäntäriippumaton rajapinta.

Olemassa olevien ohjelmistojen ohjelmarakenteiden muuntaminen käyttämään tuoterungon isäntäriippumatonta rajapintaa katsottiin olevan pääosin kannattamatonta,

minkä vuoksi tuoterunkoarkkitehtuurin keskeisimmät rakenteet päätettiin toteuttaa uudelleen olemassa olevien rakenteiden vaatimusten pohjalta. Olemassa olevat ohjelmistot päätettiin liittää tuoteperheeseen ei-siirrettävinä ohjelmistoina ja sisällyttää tuoterunkoon yhteensopivuusprotokolla, joka mahdollistaa kommunikoinnin näiden ohjelmistojen kanssa. Myös käyttöjärjestelmäriippuvaiset uudelleenkäytettävät moduulit päätettiin sisällyttää tuoterunkoon ei-siirrettävinä komponentteina.

7.3 Toteutus

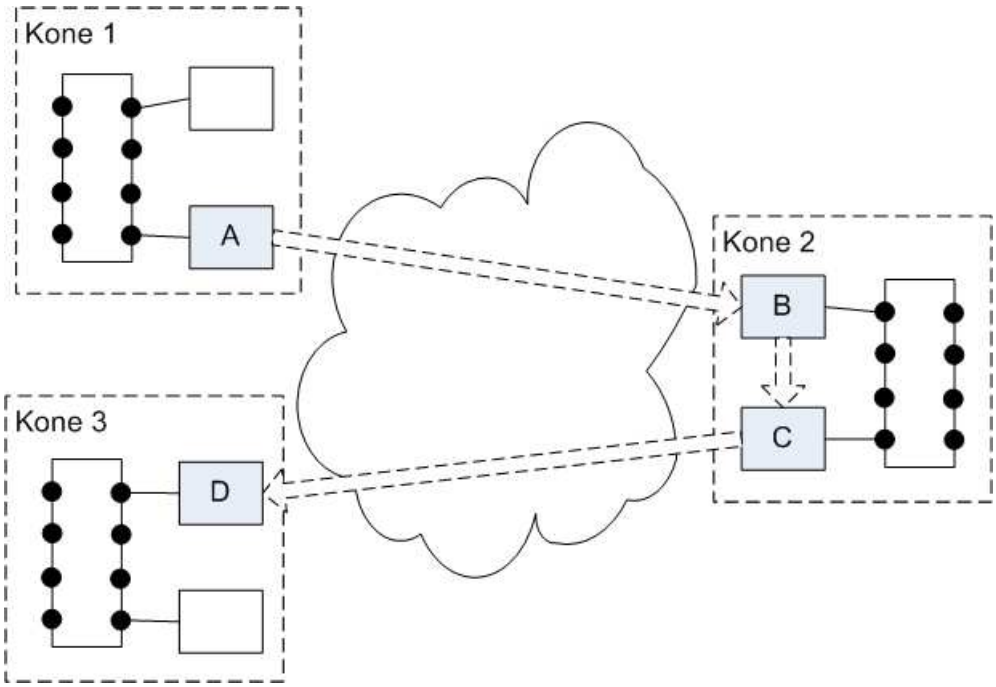
XMPL tuoterungon tuoterunkoarkkitehtuuri perustuu *tietovuoarkkitehtuuriin*, jossa tuoterunkoa hyödyntävät ohjelmistot muodostetaan määrittelemällä ohjelmistoon valittavien komponenttien välille tietovirtoja. Tietovirtoja muodostetaan komponenttien tuottaman datan avulla luvussa 4.4.6 esitellyn julkaise-tilaa arkkitehtuurin mukaisesti. Komponenttien tuottama tietovirroissa välitettävä data esitetään alustariippumattomina binaarisina viestipaketteina. Viestipakettien välityksestä komponenteille huolehtii luvussa 4.4.5 esitellyn viestijonoarkkitehtuurin mukainen tuoterungon viestinvälitysjärjestelmä. Hajautetun järjestelmän eri solmuissa komponentteja hallitsevat komponenttialustat, jotka muodostavat yhdessä sovellustason tilapäisverkon.

Kuvassa 7.1 on esitetty XMPL tuoterungon avulla muodostetun ohjelmiston ajonaikainen rakenne. Kuvan ohjelmistossa on määritelty tietovirta, jonka tiedon *tuottajana (source)* toimii koneella 1 sijaitseva komponentti A. Komponentilta A tietovirta kulkee kahden koneella 2 sijaitsevan komponentin (B ja C) kautta tiedon *hyväksikäyttäjänä (sink)* toimivalle komponentille D. XMPL ohjelmistossa komponentit voisivat esimerkiksi olla laitteistoliitynnästä vastaava komponentti (A), laitteiston tuottaman datan analysoinnista vastaavat komponentit (B ja C) ja analyysin tuloksien esittämisestä vastaava komponentti (D).

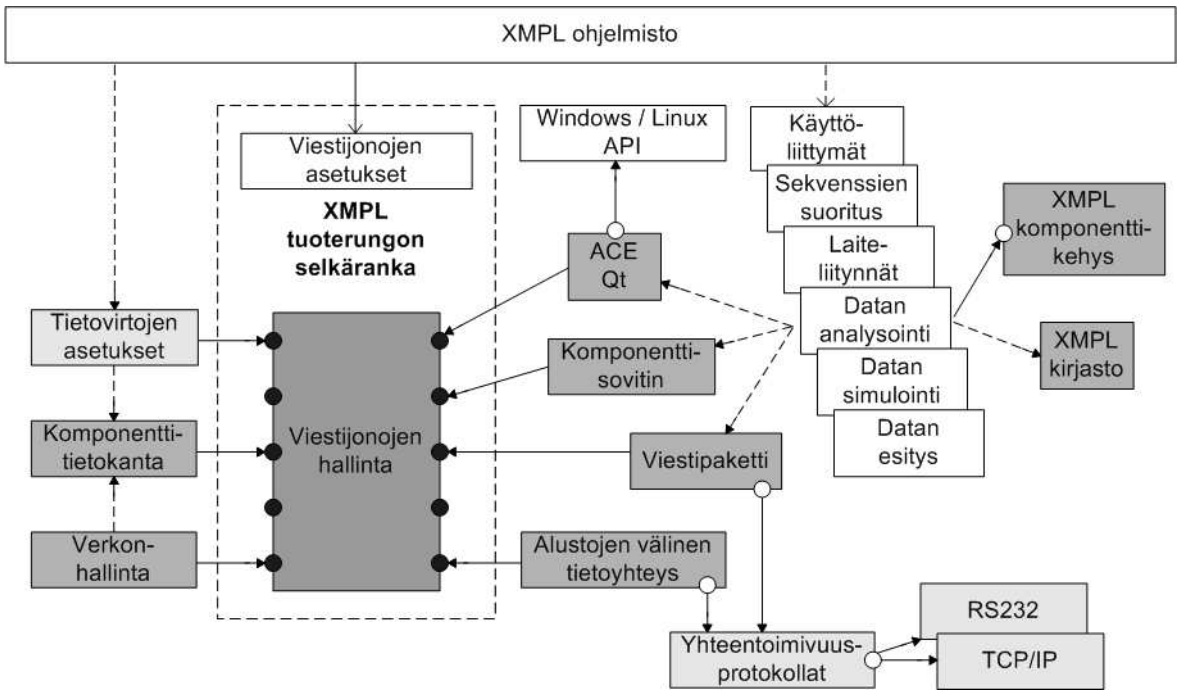
Kuvassa 7.2 esitetään XMPL tuoterungon toteutuksen arkkitehtuuri. Kaikki muut tuoterungon elementit paitsi selkäranka, isäntäriippumaton rajapinta, viestipaketti sekä XMPL kirjasto ja komponenttikehys ovat toteutettu jaetuista kirjastoista dynaamisesti ladattavina komponentteina. Tämä rakenne mahdollistaa tuoterunkoa hyödyntävien ohjelmistojen toiminnan voimakkaan ajonaikaisen muuntelun.

7.3.1 Selkäranka

Kuvassa 7.3 esitetään XMPL tuoterungon selkärangan rakenne. Selkäranka muodostuu komponenttien viestijonoja hallitsevasta komponenttialustasta sekä viestijonojen asetukset määrittävistä asetustiedoista. Jokaista selkärankaan liitettyä komponenttia

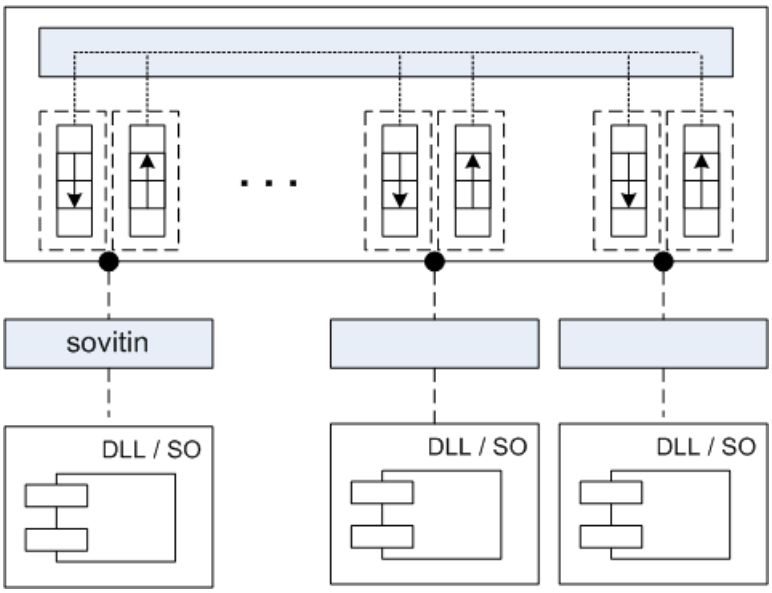


Kuva 7.1: XMPL tuoterunkoa hyödyntävä ohjelmisto.



Kuva 7.2: XMPL tuoterunko.

kohden alusta sisältää kaksi viestijonoa eli lähtevien ja saapuvien viestien jonon. Jokaiselle viestijonolle on osoitettu yksi jonon purkamisesta vastaava säie. Lähtevien viestien jonon purkusäikeessä jonon viestipaketit toimitetaan viestin vastaanottajan saapuvien viestien jonoon. Saapuvien viestien jonon purkusäikeessä jonon viestipaketit toimitetaan paketin vastaanottavan komponentin käsiteltäväksi. Paikallisen solmun sisällä viestipakettien välityksessä ei suoriteta pakettien kopiointia, vaan paketteihin viitataan viestipaketin varaamaan muistialueeseen osoittavan osoittimen avulla. Komponentti saa suoritussäikeensä saapuvien viestien jonon purusta vastaavasta säikeestä luvussa 4.4.1 esitetyn vuorottajamallin mukaisesti. Komponentille on osoitettu kerrallaan ainoastaan yksi suoritussäie eli komponentin toteutuksessa ei tarvitse huomioida komponentin tilan suojaamista säikeiden samanaikaiselta käsittelyltä.



Kuva 7.3: XMPL tuoterungon selkäranka.

Selkärangan toteutus on komponenttien välisen viestinvälitysjärjestelmän toteuttava sovelluskehys. Kehyksen laajennoskohdat muodostavat viestinvälitysjärjestelmään liitettävät komponentit. Viestinvälitysjärjestelmään liitettävien komponenttien eli kehyksen laajennoskohtien lukumäärää ei ole teoriassa rajattu, mutta käytännössä rajan asettaa viestijonojen käsittelystä vastaavien säikeiden käyttämät resurssit. Kehyksen laajennoskohdat ovat keskenään identtisiä ja laajennoskohdan täydentämiseen vaaditaan viestipaketin vastaanotosta huolehtivan rajapinnan toteutus. Kuten kuvasta 7.3 nähdään, rajapinta voidaan toteuttaa komponenttisovittimen avulla, joka välittää viestipaketit komponentin toteutukselle.

Viestijonoarkkitehtuuriin perustuvan viestinvälitysjärjestelmän toteuttama kompo-

nenttien välinen kommunikointi on asynkronista ja pysyvää. Jos komponenttia, jolle viestipaketti on osoitettu, ei ole vielä aktivoitu tai komponentti ei ehdi käsitellä pakettia, viestipaketti säilytetään komponentin saapuvien viestien jonossa. Tarvittaessa viestijonojen paketit voidaan myös tallentaa pysyvään muistiin.

7.3.2 Komponentit ja komponenttien integrointi

XMLP tuoterunko sisältää komponenttikehyksen, joka toteuttaa yleisimmät tuoterungon komponenttien kehityksessä tarvittavat palvelut. Komponenttikehyksen määrittelemästä kantaluokasta voidaan perimällä erikoistaa lopullisia tuoterunkoon lisättäviä komponentteja. Lisäksi tuoterunko sisältää kirjaston, joka tarjoaa binaaristen viestipakettien muodostamisessa ja käsittelyssä tarvittavia palveluja.

Komponenttien rajapinnat määrittyvät komponentin tuottamien ja kuluttamien viestipakettien muodon perusteella. Komponenttikehys sisältää luvussa 4.4.6 esitellyn julkaise-tilaa arkkitehtuurin mukaisen datapalvelun, jonka avulla komponentti voi määritellä kuluttamiensa viestipakettien muodon ja julkaista tuottamiansa viestipaketteja. Datapalvelun avulla voidaan selvittää komponentin tuottamien viestipakettien muoto ja rekisteröidä tilaajia komponentin tuottamille viestipaketeille. Komponentit välittävät tuottamansa datan datapalvelulle, joka puolestaan toimittaa datasta muodostetut viestipaketit rekisteröidyille tilaajille selkärangan viestinvälitysjärjestelmän kautta. Datapalvelu voi pilkkoa ja puskuroida komponenttien välittämää dataa palvelulle määrittyjen asetusten mukaisesti ennen pakettien toimitusta viestinvälitykseen.

Datapalveluun voidaan rekisteröidä tilaajaksi myös oman solmun ulkopuolisia komponentteja, joille viestipaketit välitetään alustojen välisen tietoyhteyden kautta. Datapalvelulla on keskeinen rooli tuoterunkoa hyödyntävien ohjelmistojen tietovirtojen muodostamisessa. Tietovirrat määritellään rekisteröimällä virrassa seuraavana tuleva komponentti edellisen komponentin tuottamaa dataa tilaavaksi datapalvelun asiakkaaksi.

Komponenttien käyttöliittymät voidaan myös erottaa omiksi komponenteikseen erilleen komponenttien ohjelmalogiikan toteutuksesta ja sijoittaa käyttöliittymä eri järjestelmän solmuun kuin ohjelmalogiikka. Käyttöliittymäkomponentit ovat tavallisia komponentteja, jotka kommunikoivat ohjaamansa komponentin kanssa selkärangan viestinvälitysjärjestelmän kautta.

Tuoterungon komponentit ovat C++-ohjelmointikielellä toteutettuja binaarisia komponentteja, joiden jakelumuoto on jaettu kirjasto (*dynamic link library* tai *shared object library*). Komponentit liitetään selkärangan viestinvälitysjärjestelmään komponenttisovittimen avulla. Komponenttisovitin toteuttaa selkärangan muodostavan sovelluske-

hyksen laajennoskohdassa määritetyn rajapinnan ja huolehtii jaetun kirjaston eksplisiittisesta dynaamisesta linkityksestä, kirjaston alustuksesta, komponentin ilmentymän luomisesta ja poistamisesta sekä viestinvälitysjärjestelmän viestipakettien välittämisestä komponentin ilmentymälle. Ladattavan jaetun kirjaston on julkaistava ja toteutettava edellä mainituissa tehtävien toteutuksessa tarvittavat neljä funktiota. Nämä funktiot voidaan generoida automaattisesti komponenttikehyksen tarjoaman skriptin avulla.

7.3.3 Heterogeenisyyden hallinta

Tuoterungon isäntäriippumaton rajapinta muodostuu ACE ohjelmistokehyksen tarjoamista C++-kääreluokista sekä käyttöliittymien kehityksessä käytettävän Qt-työkaluohjelmiston tarjoamista ohjelmarakenteista. Isäntäriippumaton rajapinta tarjoaa käyttöjärjestelmäriippumattoman ohjelmointirajapinnan tuoterungon Windows ja Linux -käyttöjärjestelmissä toimiville elementeille. Kaikki tuoterungon siirrettävien elementtien käyttöjärjestelmäkutsut kulkevat isäntäriippumattoman rajapinnan kautta.

Isäntäriippumatonta rajapintaa hyödynnetään selkärangan viestijonojen ja säikeistyksen, komponenttisovittimien eksplisiittisen dynaamisen linkityksen, tietoliikenneyhteyksistä huolehtivien komponenttien sekä siirrettävien valinnaisten komponenttien toteutuksessa. Isäntäriippumattoman rajapinnan avulla määritellään myös viestipaketeissa ja tuoterungon toteutuksessa käytettävät alustariippumattomat C++-datatyypit.

7.3.4 Komponenttitietokanta ja tilapäisverkon hallinta

Hajautetun järjestelmän solmuissa sijaitsevat komponenttialustat ovat yhteydessä toisiinsa alustojen välisen tietoyhteyden kautta. Toisiinsa yhteydessä olevat komponenttialustat muodostavat järjestelmään sovellustason tilapäisverkon, jossa jokainen alusta voi toimia verkon reitityspisteenä. Tilapäisverkko voi sisältää erilaisia tietoliikenneyhteyksiä, joiden päällä välitetään tuoterungon binaarisia viestipaketteja rungon määrittelemän yhteentoimivuusprotokollan mukaisesti. Yhteentoimivuusprotokolla määrittelee, miten viestipaketit puretaan ja lähetetään tiedonsiirtomedialle sekä miten paketit kootaan vastaanottopäässä.

Jokaiseen komponenttialustaan on liitetty alustaan liitettyjen komponenttien tiedot sisältävä komponenttitietokanta. Paikalliseen viestinvälitysjärjestelmään liitettyjen komponenttien lisäksi tietokanta sisältää tiedot sellaisista muiden alustojen komponenteista, jotka kuuluvat paikallisten komponenttien kanssa samaan tietovirtaan. Komponenttitietokanta voi myös synkronoida tietojaan muihin komponenttialustoihin liitettyjen komponenttitietokantojen kanssa. Komponenttitietokannan tietoja voidaan siis hyödyntää myös uusien tietovirtojen määrityksessä.

Komponenttialustalla voi olla useita yhtäaikaisia yhteyksiä järjestelmän eri solmuissa sijaitseviin muihin komponenttialustoihin ja alustojen muodostamassa tilapäisverkossa kahden verkon solmun välillä voi olla useita vaihtoehtoisia reittejä. Reittien sisältämien solmujen lukumäärä, solmujen reitityskapasiteetti sekä solmujen välisten tietoliikenneyhteyksien laatu voivat vaihdella. Verkonhallintakomponentti vastaa viestipakettien optimaalisesta reitityksestä komponenttitietokannassa määritellyille etäkomponenteille. Paikallisen viestinvälitysjärjestelmän ulkopuolelle osoitettu viestipaketti toimitetaan aina ensin verkonhallintakomponentille, joka toimittaa valitsemaansa reittiä pitkin paketin eteenpäin.

7.3.5 Tuoterungon käyttöä tukevat työkalut

Tuoterunko sisältää komponenttien välisten tietovirtojen määrittämiseen tarkoitetun työkalun. Työkalun avulla voidaan ohjelmiston ajonaikana liittää uusia komponentteja järjestelmän eri solmuissa sijaitseviin komponenttialustoihin, poistaa liitettyjä komponentteja, määrittää komponenttien viestijonojen asetuksia, tarkastella komponenttitietokannan tietoja paikallisista ja muiden komponenttialustojen hallitsemista komponenteista sekä määrittää komponenttien välisiä tietovirtoja. Tietovirtojen määritys suoritetaan määrittelemällä työkalun käyttöliittymän avulla virtaan kuuluvien komponenttien datapalveluasetukset luvussa 7.3.2 esitetyllä tavalla. Määriteltyjä tietovirtoja voidaan myös muokata ajonaikaisesti.

7.4 Arviointi

XMPL tuoterungon perustaminen perustui hyvin pitkälle aiemmin kehitettyjen, lähinnä laitteistojen testaukseen suunnattujen, ohjelmistojen vaatimusten sekä suunnitteluja toteutusratkaisujen keräämiseen tuoterungoksi. Keskeisimmät rakenteet toteutettiin uudelleen aiempien vaatimusten ja suunnitelmien pohjalta, mutta merkittäviä rakenteellisia muutoksia tuoterungon perustamisen yhteydessä ei tehty. Aiemmin kehitettyjen ohjelmistojen hajautus on perustunut lähinnä ennalta määritetyistä datalähteistä kerättävän datan hajauttamiseen eri koneiden käsiteltäväksi. Ohjelmiston eri osien välillä välitettävien viestipakettien koko sekä määrä on ollut ennustettavissa ja eri osien välisellä kommunikaatiolla ei ole ollut tiukkoja vaatimuksia palvelun laadun suhteen. Lisäksi ohjelmistojen muodostamien tilapäisverkkojen solmujen ja tietovirtojen sisältämien komponenttien lukumäärät ovat olleet suhteellisen pieniä. Tällaisessa toimintaympäristössä tuoterungon viestijonoarkkitehtuuriin perustuva viestinvälitysjärjestelmä sekä tietovirtoihin perustuva tuoterunkoarkkitehtuuri täyttävät tehtävänsä hyvin.

Tuoterungon viestinvälitysjärjestelmä ja tuoterunkkoarkkitehtuuri asettavat kuitenkin tuoterungon komponenteille ja runkoa hyödyntäville ohjelmistoille myös rajoitteita, jotka on otettava huomioon tuoteperheen ohjelmistoja suunniteltaessa.

Tuoterungon viestinvälitysjärjestelmän viestijonojen hallinta edellyttää, että komponenttien välillä välitettävien viestipakettien koko ja lukumäärä ovat ennustettavissa. Luvun 7.1 tuoterungon sovellusalueen määrittelyssä mainittiin, että järjestelmän sisältämien datalähteiden tuottaman datan määrä voi vaihdella yksittäisistä tapahtumailmoituksista hyvinkin suuriin datamääriin. Tällaisen datan välittämiseen tuoterungon viestinvälitysjärjestelmässä voitaisiin ajatella olevan kaksi vaihtoehtoa. Lähteen tuottama data voitaisiin joko paketoida heti viestipaketeiksi ja lähettää viestinvälityksen kautta prosessoitavaksi tai data voitaisiin pilkkoa tai puskuroida ennen lähetystä. Kun järjestelmä sisältää paljon erilaisia datamääriä tuottavia datalähteitä, ensin mainitun vaihtoehdon mukaisten viestijonojen dynaaminen hallinta on lähes mahdotonta toteuttaa. Suurten viestipakettien käsittely saapuvien viestien jonon purkusäikeessä voi pysäyttää jonon purun pitkäksi aikaa ja toisaalta suuri määrä pieniä paketteja täyttää jonot nopeasti. Ainoaksi järkeväksi vaihtoehdoksi jää datan pilkkominen tai puskuroiminen ennen lähetystä viestinvälitykseen, minkä mukaisesti myös tuoterungon komponenttikehyksen datapalvelu toimii. Komponenttien välillä välitettävän datan pilkkomisen ja puskuroinnin ansiosta tuoterungon viestinvälitysjärjestelmän toiminnan ennustettavuus paranee. Datalähteiden tuottaman datan määrä on kuitenkin oltava etukäteen tiedossa, jotta datan keräämisestä sekä käsittelystä vastaavien komponenttien viestijonojen asetukset osataan asettaa oikein.

Viestinvälitysjärjestelmän asynkronisuus ja tietovuoarkkitehtuuriin perustuva tuoterunkoarkkitehtuuri on huomioitava myös komponentteja suunniteltaessa. Tuoterunko ei tarjoa tukea synkronista kommunikaatiota vaativille komponenteille ja komponenttien välinen kommunikaatio ei ole luotettavaa. Lisäksi datapalvelun kautta tapahtuvassa viestinvälityksessä on otettava huomioon palvelun suorittama datan puskurointi. Tämä on otettava erityisesti huomioon yksittäisiä tapahtumailmoituksia tuottavien komponenttien, kuten käyttöliittymien, suunnittelussa. Tuoterungon hajautettujen käyttöliittymien toiminnan ei yleisesti ottaen tulisi perustua tavallisten käyttöliittymien tapaan runsaaseen tapahtumailmoitusten käyttöön, koska tuoterungon viestinvälitysjärjestelmän tarjoama tuki kyseiselle toimintamallille on hyvin vähäinen.

Tuoterungon hajautuksen läpinäkyvyyden aste on melko pieni ja rungon tarjoamat luvun 4.3.1 mukaiset läpinäkyvyyspalvelut ovat vähäisiä. Tuoterungon viestipaketin avulla toteutettu komponenttien yhteinen kommunikaatio- ja tiedonesitystapa toteuttavat saantiläpinäkyvyyden, eivätkä häiriöt komponenttien viestipakettien käsittelyssä jätä komponentteja epäjohdonmukaiseen tilaan eli myös järjestelmän häiriöläpinäky-

vyys toteutuu. Tietovirtojen määritys ei edellytä, että käyttäjä tietää komponentin fyysisen sijainnin järjestelmässä eli teoriassa sijaintiläpinäkyvyys toteutuu. Käytännössä tietovirtaa määrittelevän käyttäjän on kuitenkin tunnettava järjestelmän rakenne, järjestelmän sisältämien tietoliikenneyhteyksien laatu sekä tietovirtaan kuuluvien komponenttien sijainti järjestelmässä, jotta käyttäjä osaisi arvioida tietovirrassa välitettävää datamäärää tietovirran datan kuljetuskapasiteettiin. Komponentin sijainnin muuttuminen järjestelmässä ei vaikuta tapaan, jolla komponentin kanssa kommunikoidaan eli muuttoläpinäkyvyys toteutuu. Tietovirtaan sisältyvän komponentin sijaintia ei voida kuitenkaan muuttaa ilman, että se vaikuttaisi tietovirran toimintaan eli sijoitettavuusläpinäkyvyys ei toteudu. Monistusläpinäkyvyys ei toteudu, koska komponenttien datapalvelu on toteutettu komponenttikohtaisesti siten, että useampi komponentti ei voi toteuttaa tiettyä samaan palveluun määritettyä rajapintaa. Pysyvyysläpinäkyvyydestä eli komponentin tilan tallentamisesta ja palautuksesta komponentin aktivoinnin yhteydessä voidaan huolehtia yksittäisen komponentin toteutuksessa, mutta tuoterunko ei tarjoa komponenteille tilan hallintaa tukevia palveluja. Tapahtumaläpinäkyvyydelle ei ole tarvetta tuoterunkoarkkitehtuurin mukaisissa ohjelmistoissa.

Tuoterungon vähäisten läpinäkyvyyspalvelujen vuoksi viestijonojen sekä tietovirtojen konfigurointi hankaloituu huomattavasti järjestelmän sisältämien solmujen ja ohjelmiston sisältämien komponenttien lukumäärän kasvaessa. Tuoterunkoa hyödyntävät ohjelmistot muodostetaan määrittelemällä ohjelmistoon liitettävät komponentit, komponenttien viestijonojen asetukset sekä komponenttien väliset tietovirrat. Asetuksia määrittävän käyttäjän on tunnettava tuoterungon viestinvälitysjärjestelmän toimintaperiaate ja ohjelmiston sisältämien komponenttien käsittelemät datamäärät, jotta käyttäjä osaisi asettaa viestijonojen asetukset oikein. Lisäksi käyttäjän on tunnettava järjestelmän sisältämien tietoliikenneyhteyksien laatu ja otettava ne huomioon tietovirtoja määriteltäessä. Laajoissa järjestelmissä ohjelmiston asetusten määrityksestä tulee helposti hyvin vaativa tehtävä, jonka helpottamiseksi tuoterungon hajautuksen läpinäkyvyyden astetta tulisi nostaa ja tuoterungon tarjoamia läpinäkyvyyspalveluja sekä asetusten määritystä tukevien työkalujen toimintaa tulisi kehittää huomattavasti.

8 Yhteenveto

Tutkielmassa selvitettiin hajautetun, heterogeenisen ympäristön asettamia vaatimuksia tuoterungolle ja esitettiin, miten tuoterungon kehityksessä voidaan huomioida näitä vaatimuksia. Tuoterunko tehostaa ohjelmistojen uudelleenkäyttöä organisaatiossa, yhdistämällä uudelleenkäytettävät ohjelmistorakenteet sekä rakenteiden kehitystä ja käyttöä tukevan ohjelmistoarkkitehtuurin. Hajautettu ympäristö asettaa tuoterungolle monia vaatimuksia, joita ei tarvitse huomioida ei-hajautetussa tuoterungossa. Nämä vaatimukset on otettava huomioon sekä tuoterunkoarkkitehtuurin että tuoterungon komponenttien suunnittelussa ja toteutuksessa.

Tutkielman alussa esiteltiin perinteisen ei-hajautetun tuoterungon ominaisuuksia, kehitystä ja käyttöä yrityksen ohjelmistotuotannossa sekä käsiteltiin tuoterungon toteutuksessa tärkeässä roolissa olevien ohjelmistokehysten ominaisuuksia ja kehitystä. Tämän jälkeen siirryttiin järjestelmän hajautuksen ohjelmistojen kehitykselle asettamien haasteiden käsittelyyn ja esiteltiin hajautetun ympäristön ominaisuuksia, hajautetussa ympäristössä toimivilta ohjelmistoilta vaadittavia ominaisuuksia, hajautuksen toteutusta sekä väliohjelmistojen käyttöä hajautuksen hallinnan apuna.

Järjestelmän hajautuksen tarkastelun perusteella voitiin todeta hajautuksen aiheuttavan haasteita erityisesti tuoterungon komponenttien välisen kommunikaatioarkkitehtuurin sekä hajautetun, heterogeenisen ympäristön peittävien läpinäkyvyyspalvelujen kehityksessä. Tutkielman empiirisessä osuudessa koottiin aiemman tarkastelun pohjalta hajautuksen haasteet huomioiva, hajautetussa ympäristössä toimivan tuoterungon suunnittelua tukeva viitekehys. Viitekehys esitteli hajautetun tuoterungon arkkitehtuurimallin, jota voidaan käyttää tuoterungon hajautuksen toteuttavien elementtien suunnittelun pohjana. Lisäksi käsiteltiin hajautetun tuoterungon perustamista ja hajautuksen huomioimista tuoterungon perustamisessa.

Tutkielman lopuksi tarkasteltiin erään organisaation hajautetun tuoterungon perustamista ja tuoterungon hajautuksen toteuttavaa rakennetta. Tehdyn tarkastelun pohjalta voitiin todeta, kuinka tärkeää tuoterungon hajautus on huomioida tuoterunkoarkkitehtuurissa, jotta tuoteperheen ohjelmistojen muodostaminen tuoterungon avulla olisi mahdollisimman helppoa. Tuoterunkoarkkitehtuurissa on erityisesti huomioitava tuoterungon komponenttien välinen kommunikaatioarkkitehtuuri. Kommunikaatioarkkitehtuurin on tuettava ohjelmistojen kehitystä tuoterungon sovellusalueella ja huonosti valittu kommunikaatioarkkitehtuuri voi asettaa huomattavia rajoitteita

tuoteperheen ohjelmistoille. Lisäksi todettiin, kuinka tärkeää tuoterungon on tarjota tarpeeksi hajautetun ympäristön peittäviä läpinäkyvyyspalveluja, jotta tuoteperheen ohjelmistojen kehitys olisi mahdollisimman helppoa.

Tutkielma keskittyi tuoterungon hajautuksen kannalta välttämättömien ominaisuuksien käsittelyyn ja sivuutti useiden tärkeiden hajautukseen liittyvien tekijöiden, kuten ohjelmistojen turvallisuuden ja palvelun laadun, käsittelyn. Tuoterungon hajautuksen tarkastelu pyrittiin kuitenkin tekemään siten, ettei se aseta rajoitteita tarkastelun ulkopuolelle jääneiden tekijöiden suunnittelulle ja toteutukselle.

9 Viitteet

[1] Software Engineering Institute, Carnegie Mellon University, *Software Technology Roadmap – Object Request Broker*, saatavilla WWW-muodossa <URL: `http://www.sei.cmu.edu/str/descriptions/orb.html`>, 14.12.2005.

[2] DOC group, *The Adaptive Communication Environment*, saatavilla WWW-muodossa <URL: `http://www.cs.wustl.edu/~schmidt/ACE.html`>, 9.10.2006.

[3] DOC group, *The ACE ORB*, saatavilla WWW-muodossa <URL: `http://www.cs.wustl.edu/~schmidt/TAO.html`>, 9.10.2006.

[4] DOC group, *The Component-Integrated ACE ORB*, saatavilla WWW-muodossa <URL: `http://www.cs.wustl.edu/~schmidt/CIAO.html`>, 9.10.2006.

[5] Information technology – open distributed processing – reference model: Overview. Standard ISO/IEC 10746-1, International Organization for Standardization (ISO), 1998.

[6] Open Architecture (OA) Computing Environment design guidance. Tekninen raportti Version 1.0, Naval Surface Warfare Center Dahlgren Division (NSWCDD), 2004.

[7] Open Architecture (OA) Computing Environment technologies and standards. Tekninen raportti Version 1.0, Naval Surface Warfare Center Dahlgren Division (NSWCDD), 2004.

[8] AMQP Advanced Message Queuing Protocol. Protocol Specification Version 0.8, JPMorgan Chase Bank, Cisco Systems, Inc., Envoy Technologies Inc., iMatix Corporation, IONA Technologies, Red Hat, Inc., TWIST Process Innovations, 29West, Inc., 2006.

[9] Jan Bosch. *Design and use of software architectures – adopting and evolving a product-line approach.* Addison-Wesley, Iso-Britannia, 2000.

[10] David R. Butenhof. *Programming with POSIX threads.* Addison-Wesley, Yhdysvallat, 1997.

[11] D. Cohen. On holy wars and a plea for peace. *IEEE Computer Magazine*, 14:48–54, 1981.

[12] Sholom Cohen. Predicting when product line investment pays. Technical Note CMU/SEI-2003-TN-017, Software Engineering Institute, Carnegie Mellon University, 2003.

[13] M. Eisler. XDR: External Data Representation Standard. STD 0067, Internet Engineering Task Force, 2006.

[14] Object Management Group. Object management architecture guide revision 3.0, 1995.

[15] Object Management Group. Common object request broker: Core specification version 3.0.3, 2004.

[16] Object Management Group. Lightweight CORBA Component Model, 2004.

[17] Object Management Group. Notification service specification version 1.1, 2004.

[18] Object Management Group. CORBA component model specification version 4.0, 2006.

[19] Alan Burns ja Andy Wellings. *Real-Time Systems and Programming Languages*. Addison-Wesley, Iso-Britannia, 2001.

[20] David Parsons ja Awais Rashid ja Alexandru Telea ja Andreas Speck. An architectural pattern for designing component-based application frameworks. *Software–Practice and Experience*, 36(2):157–190, 2006.

[21] Gerardo Pardo-Castellote ja Bert Farabaugh ja Rick Warren. An introduction to DDS and data-centric communications. White paper, Real-Time Innovations, 2005.

[22] Ralph E. Johnson ja Brian Foote. Designing reusable classes. *Journal of Object-Oriented Programming*, 1(2):22–35, 1988.

[23] Andrew D. Birrell ja Bruce Jay Nelson. Implementing remote procedure calls. *ACM Transactions on Computer Systems*, 2(1):39–59, 1984.

[24] David M. Weiss ja Chi Tau Robert Lai. *Software Product-Line Engineering*. Addison-Wesley, Yhdysvallat, 1999.

[25] Robert Orfali ja Dan Harkey ja Jeri Edwards. *The Essential Distributed Objects Survival Guide*. John Wiley & Sons, Yhdysvallat, 1996.

[26] Mohamed E. Fayad ja Douglas C. Schmidt. Lessons learned building reusable oo frameworks for distributed software. *Communications of the ACM*, 40(10):85–87, 1997.

[27] Mohamed Fayad ja Douglas C. Schmidt. Object-oriented application frameworks. *Communications of the ACM*, 40(10):32–38, 1997.

[28] Richard E. Schantz ja Douglas C. Schmidt. Middleware for distributed systems: Evolving the common structure for network-centric applications. Kirjassa *Encyclopedia of Software Engineering*. Wiley & Sons, 2001.

[29] Nanbor Wang ja Douglas C. Schmidt ja Carlos O'Ryan. Overview of the CORBA component model. Kirjassa *Component-based software engineering: Putting the Pieces Together*, ss. 557–571. Addison-Wesley, Yhdysvallat, 2001.

[30] Butler W. Lampson ja Eric E. Schmidt. Practical use of a polymorphic applicative language. Kirjassa *POPL '83: Proceedings of the 10th ACM SIGACT-SIGPLAN symposium on Principles of programming languages*, ss. 237–255. ACM Press, 1983.

[31] Wolfgang Pree ja Gustav Pomberger ja Albert Schappert ja Peter Sommerlad. Active guidance of framework development. *Software – Concepts and Tools*, 16(3):136–, 1995.

[32] J. Andersson ja J. Bosch. Development and use of dynamic product-line architectures. *IEE Proceedings-Software*, 152(1), 2005.

[33] Stephen D. Huston ja James CE Johnson ja Umar Syyid. *The ACE Programmer's Guide*. Addison-Wesley, Yhdysvallat, 2004.

[34] Jilles Van Gurp ja Jan Bosch ja Mikael Svahnberg. On the notion of variability in software product lines. Kirjassa *Proceedings of the Working IEEE/IFIP Conference on Software Architecture (WICSA'01)*, ss. 45–54, 2001.

[35] George F. Coulouris ja Jean Dollimore ja Tim Kindberg. *Distributed Systems: Concepts and Design*. Addison-Wesley, Iso-Britannia, 2001.

[36] Martin L. Griss ja John Favaro ja Massimo d'Alessandro. Integrating feature modeling with the RSEB. Kirjassa *Proceedings of Fifth International Conference on Software Reuse*, 1998.

[37] Wolfgang Pree ja Kai Koskimies. Framelets – small and loosely coupled frameworks. *ACM Comput. Surv.*, 32(1es):6, 2000.

[38] Steve Sparks ja Kevin Benner ja Chris Faris. Managing object-oriented framework reuse. *Computer*, 29(9):52–61, 1996.

[39] Randall R. Macala ja Lynn D. Stuckey Jr. ja David C. Gross. Managing domain-specific product-line development. *IEEE Softw.*, 13(3):57–67, 1996.

[40] Sybren Deelstra ja Marco Sinnema ja Jan Bosch. Product derivation in software product families: a case study. *The journal of Systems and Software*, 74:173–194, 2005.

[41] Ivar Jacobson ja Martin Griss ja Patrik Jonsson. *Software Reuse – Architecture, Process and Organization for Business Success*. Addison-Wesley, Yhdysvallat, 1997.

[42] Andrew S. Tanenbaum ja Martin van Steen. *Distributed Systems – Principles and Paradigms*. Prentice-Hall, Iso-Britannia, 2003.

[43] Len Bass ja Paul Clements ja Rick Kazman. *Software Architecture in Practice*. Addison-Wesley, Yhdysvallat, 1997.

[44] Erich Gamma ja Richard Helm ja Ralph Johnson ja John Vlissides. *Design Patterns – Elements of Reusable Object-Oriented Software*. Addison-Wesley, Yhdysvallat, 1995.

[45] Kyo C. Kang ja Sajoong Kim ja Jaejoon Lee ja Kijoo Kim ja Euiseob Shin ja Moonhang Huh. FORM: A feature-oriented reuse method with domain-specific reference architectures. *Annals of Software Engineering*, 5(0):143–168, 1998.

[46] Kyo C. Kang ja Sholom G. Cohen ja James A. Hess ja William E. Novak ja A. Spencer Peterson. Feature-oriented domain analysis (FODA) feasibility study. Technical Report CMU/SEI-90-TR-21, Software Engineering Institute, Carnegie Mellon University, 1990.

[47] Douglas C. Schmidt ja Stephen D. Huston. *C++ Network Programming Volume 1*. Addison-Wesley, Yhdysvallat, 2002.

[48] Douglas C. Schmidt ja Stephen D. Huston. *C++ Network Programming Volume 2*. Addison-Wesley, Yhdysvallat, 2003.

[49] Kai Koskimies ja Tommi Mikkonen. *Ohjelmistoarkkitehtuurit*. Talentum, Suomi, 2005.

[50] Dejan S. Milojicic ja Vana Kalogeraki ja Rajan Lukose ja Kiran Nagaraja ja Jim Pruyne ja Bruno Richard ja Sami Rollins ja Zhichen Xu. Peer-to-peer computing. White paper, Hewlett-Packard, 2002.

[51] Ralph E. Johnson. Frameworks = (components + patterns). *Communications of the ACM*, 40(10):39–42, 1997.

[52] B. Clifford Neuman. Scale in distributed systems. Kirjassa *Readings in Distributed Computing Systems*, ss. 463–489. IEEE Computer Society, 1994.

[53] Charles J. Northrup. *Programming with UNIX threads*. John Wiley & Sons, Yhdysvallat, 1996.

[54] J. Postel. Internet protocol. STD 0067, Internet Engineering Task Force, 1981.

[55] Wolfgang Pree. *Design Patterns for Object-Oriented Software Development*. Addison-Wesley, Iso-Britannia, 1995.

[56] Douglas C. Schmidt. The Adaptive Communication Environment – An object-oriented network programming toolkit for developing communication software. Kirjassa *Proceedings of the 11th and 12th Sun Users Group Conference*, 1993, 1994.

[57] Douglas C. Schmidt. An architectural overview of the ACE framework – a case-study of successful cross-platform systems software reuse. *The USENIX login Magazine*, 1999.

[58] R. Srinivasan. RPC: remote procedure call protocol specification version 2. RFC 1831, Internet Engineering Task Force, 1995.