Arkkitehtuurikuvauksia hyödyntävä meklaus

Markku Vähäaho

Helsinki 2. huhtikuuta 2003
Pro gradu -tutkielma
HELSINGIN YLIOPISTO
Tietojenkäsittelytieteen laitos

Sisältö

Luku 1

Johdanto

Kaikkien ohjelmistojen, niin hajautettujen kuin keskitettyjen, voidaan ajatella koostuvan erillisistä, toistensa kanssa viestivistä komponenteista. Keskitetyissä järjestelmissä komponentit sijaitsevat yhdessä tietokoneessa, mahdollisesti jopa samassa muistiavaruudessa. Komponenttien väliset suhteet voidaan tällöin helposti kiinnittää etukäteen.

Hajautetuissa järjestelmissä komponentit voivat sijaita fyysisesti eri palvelimilla, jotka ovat yhteydessä toisiinsa ainoastaan tietoliikenneverkon välityksellä. Esimerkkejä ovat erilaiset asiakas-palvelin-järjestelmät, kuten web-selain ja web-palvelin, sekä hajautetut komponenttimallit, kuten Enterprise Java Beans (EJB). Hajautettaessa komponentit eri palvelimille keskeinen ongelma on, miten asiakaskomponentit löytävät tarvitsemansa palvelut järjestelmästä ja miten asiakas- ja palvelinkomponenttien väliset yhteydet muodostetaan.

Palvelinkomponenttien osoitteet tietoverkossa voidaan kiinnittää ohjelmiston koodiin tai alustustietoihin jo ennen ohjelmiston suoritusta. Tämä niin sanottu aikainen sidonta tekee kuitenkin järjestelmästä hauraan ja vaikean muuttaa, koska järjestelmään tehtävät muutokset edellyttävät ohjelmakoodin tai alustustietojen muuttamista käsin. Joustavampi vaihtoehto on valita käytettävä palvelin vasta ohjelmiston suoritusaikana eli käyttää myöhäistä sidontaa.

Myöhäisen sidonnan käyttö tekee järjestelmästä helpommin muutettavan ja parantaa sen laajennettavuutta ja vikasietoisuutta aikaiseen sidontaan verrattuna. Hyvin pienimuotoisia järjestelmiä lukuunottamatta myöhäinen sidonta onkin nykyään käytössä lähes kaikissa hajautetuissa järjestelmissä. Yleisin myöhäisen sidonnan toteutustapa on nimipalvelun käyttö. Nimipalvelu kuvaa palvelimen nimen sen verkko-osoitteeksi. Tunnetuin esimerkki nimipalvelusta lienee Internetin *Domain Name Service (DNS)*, joka kuvaa aluenimen, esimerkiksi www-osoitteen, palvelimen IP-osoitteeksi.

Nimipalvelu ei säilytä palvelimista muita tietoja kuin nimen ja verkko-osoitteen, joten sen käyttö on mahdollista vain, jos etsittävillä palveluilla on tunnetut nimet. Nimipalvelu ei myöskään tue valintaa usean samaa palvelua tarjoavan palvelimen välillä. Tässä suhteessa nimipalvelu vastaa puhelinluettelon valkoisia sivuja.

Toinen tapa toteuttaa myöhäinen sidonta on käyttää meklauspalvelua. Meklaus-

palvelua voidaan pitää hajautetun järjestelmän automaattisena keltaiset sivut - palveluna. Palveluja ei etsitä meklauspalvelusta palvelun nimen vaan luokituksen ja ominaisuuksien perusteella. Samoin kuin nimipalvelua käytettäessä, myös meklauksessa sidonta voi tapahtua täysin automaattisesti, käyttäjän näkymättömissä.

Meklauksella on useita erilaisia käyttökohteita. Sitä voidaan käyttää suurten hajautettujen järjestelmien, kuten puhelinkeskusten, suoritusaikaiseen konfigurointiin. Palvelujen löytäminen liikkuvalle käyttäjälle paikallisesta langattomasta verkosta on toinen mahdollinen käyttökohde; esimerkiksi Jini-tekniikan *Lookup*-palvelu on tämäntapaiseen käyttöön tarkoitettu meklauspalvelu [34].

Vielä laajempi mahdollinen käyttökohde on yritysten ja yksityishenkilöiden väliset sähköiset palvelumarkkinat, jotka tekevät vahvasti tuloaan etenkin niin sanottujen web-palvelujen (*web services*) muodossa. Web-palvelujen tavoitteena on mahdollistaa organisaatioiden tietojärjestelmien välinen saumaton yhteistyö Internetin välityksellä [2]. Web-palvelujen löytämiseen tarvitaan palveluhakemistoja, jotka voivat perustua esimerkiksi UDDI-standardiin [19]. Yritykset voivat ilmoittaa hakemistoihin paitsi yleisiä tietoja itsestään myös tietoja tarjoamistaan web-palveluista. UDDI-hakemisto ei ole varsinainen meklauspalvelu, vaikka se sisältääkin monia meklauspalvelun piirteitä.

Tietojenkäsittelytieteen laitoksen Pilarcos-projektissa, jossa tämä tutkielma on syntynyt, on tutkittu myöhäistä sidontaa meklauksen avulla organisaatioiden välillä. Perinteisistä meklauspalveluista palveluja etsitään yksi kerrallaan; organisaatioiden välisillä palvelumarkkinoilla syntyy kuitenkin tilanteita, joissa jonkin prosessin suorittamiseen tarvitaan useita yhdessä toimivia palveluja. Esimerkiksi hotellin varaamiseen saatetaan tarvita hotellinvarauspalvelun lisäksi erillistä luotettua maksupalvelua, jonka kautta maksuliikenne välitetään. Tutkielman aiheena olevan Pilarcos-meklauspalvelun avulla on mahdollista löytää automaattisesti keskinäiseen yhteistyöhön kykenevät palvelut.

Pilarcos-meklauksessa käytetään yhteistyön kuvaamiseen arkkitehtuurikuvausta, joka kertoo palveluja tarjoavien ja käyttävien komponenttien muodostaman yhteisön rakenteen. Asiakas- ja palvelinkomponentit esitetään kuvauksessa rooleina, ja Pilarcos-meklauspalvelu etsii rooleihin keskenään yhteensopivat komponentit. Meklauspalvelun prototyyppi on toteutettu CORBA-väliohjelmistoalustalla, mutta sen periaatteet ovat sovellettavissa muihinkin ympäristöihin. Pilarcos-meklaus ei myöskään ole sidoksissa ainoastaan organisaatioiden välisiin palvelumarkkinoihin. Arkkitehtuurikuvauksia voidaan yhtä hyvin käyttää hajautettujen sovellusten hallintaan organisaatioiden sisäisissä tietoverkoissa.

Tämä tutkielma pyrkii käsittelemään Pilarcos-meklausta toteutusympäristöstä riippumattomalla tavalla. Tutkielman rakenne on seuraavanlainen. Luku 2 käsittelee meklausta hajautetuissa järjestelmissä. Useista meklauspalvelustandardeista tässä tutustutaan CORBA-meklauspalveluun, johon Pilarcos-meklauspalvelu paljolti perustuu. Luvun lopuksi esitellään tutkimusongelma eli Pilarcos-meklaus ja sovellusesimerkki, johon viitataan myöhemmissä luvuissa.

Pilarcos-meklauksen käyttämiä palvelu- ja arkkitehtuurikuvauksia pienemmistä rakenneosista suurempiin käsitellään luvussa 3. Arkkitehtuurikuvaukset perustuvat

paljolti ODP-viitemallin tavoitenäkökulmakieleen sekä arkkitehtuurinkuvauskieliin.

Luku 4 käsittelee varsinaista Pilarcos-meklausta. Tyyppien hallinnan, palvelutarjousten ja palvelukyselyjen tarkastelun jälkeen kuvaillaan algoritmi, jota Pilarcos-meklauspalvelu käyttää palvelutarjousyhdistelmien etsimiseen. Algoritmin perusajatukset ovat kirjoittajan omia, mutta optimointimenetelmä on syntynyt keväällä 2002 Pilarcos-meklarin prototyypin toteuttaneessa ohjelmistotuotantoryhmässä "PIMEA" [26]. Luvun lopuksi tarkastellaan Pilarcos-meklauksen laskennallista vaativuutta.

Luku 5 esittelee CORBA-komponenttimallia käyttäen toteutetun Pilarcos-meklarin prototyypin. Prototyypin suorituskykyä on tutkittu laajoilla mittauksilla, joiden tulokset käydään läpi. Mittausten perusteella Pilarcos-meklauspalvelun suorituskyky on mahdollista saada riittävän hyväksi moniin ei-reaaliaikaisiin sovelluksiin. Luvussa 6 käsitellään Pilarcos-meklauspalvelun jatkokehitysmahdollisuuksia sekä suunnitteluratkaisujen että käyttöalueen laajentamisen osalta. Tutkielma päättyy yhteenvetoon luvussa 7. Tekstin lukemista erityisesti englanninkielisiin lähteisiin vertaillen helpottaa liitteen A suomi-englanti -sanasto.

Luku 2

Meklaus

Tässä luvussa käsitellään perinteistä meklausta hajautetuissa järjestelmissä. Aluksi määritellään meklaus ja esitellään meklausta käyttävien komponenttien roolit meklausyhteisössä. Tämän jälkeen tutustutaan tarkemmin CORBA-standardin määrittelemään meklauspalveluun, joka muodostaa pohjan jatkotyölle. Lopuksi esitellään tutkimusongelma, jonka ratkaisemiseen tarvitaan Pilarcos-meklausta.

2.1 Meklaus hajautetussa järjestelmässä

Tässä tutkielmassa esiteltävät periaatteet ovat sovellettavissa kaikkiin hajautettuihin ympäristöihin, joiden voidaan katsoa koostuvan erillisistä, toistensa kanssa viestivistä *komponenteista*. Komponentilla tarkoitetaan tässä mitä tahansa ohjelmiston koostamisyksikköä, jonka ulkoiset rajapinnat ovat hyvin määritellyt. Komponentti tarjoaa ulkoisten rajapintojensa kautta yhtä tai useampaa *palvelua* eli toimintoa, jota sen asiakkaat käyttävät. Komponentti voi myös itse käyttää toisia palveluja, eli se voi toimia *asiakkaana*, *palvelimena* tai molempina tilanteesta riippuen.

ANSA-arkkitehtuuri oli ensimmäisiä malleja, joissa meklauspalvelu oli kiinteä osa hajautettua järjestelmää. ANSA-arkkitehtuurissa meklaus määritellään seuraavasti: *"Palvelujen valinta siten, että ne vastaavat joitakin palveluvaatimuksia. Valinta perustuu vaaditun palvelun kuvauksen (jonka tekee asiakas) ja palvelun tarjoajan tai sen edustajan toimittaman palvelukuvauksen vertaamiseen"* [6]. Komponenttia, joka tarjoaa meklauspalvelua, kutsutaan *meklariksi*. Meklari on vastuussa palvelutarjousten säilyttämisestä ja sopivien palvelutarjousten löytämisestä vastaukseksi palvelukyselyihin.

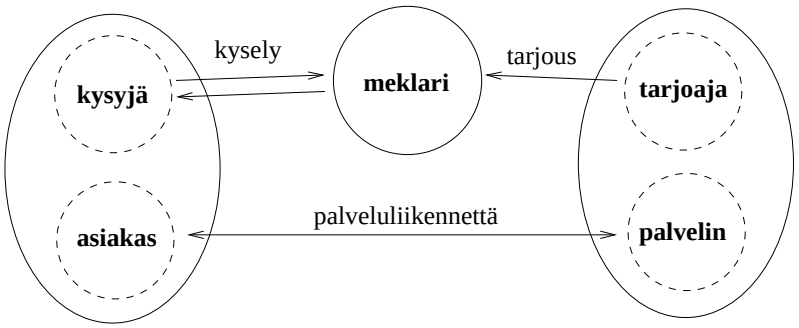
Kuva 2.1 esittää meklausyhteisön eli meklarin ja sen käyttäjien vuorovaikutuksia [10]. Yhteisön roolit ovat seuraavat:

- *Tarjoaja* ilmoittaa *palvelutarjouksia* meklarille. Palvelutarjous sisältää tarjotun palvelun kuvauksen ja ominaisuudet sekä palvelimen verkko-osoitteen, josta palvelu on saatavissa.

- *Kysyjä* tekee *palvelukyselyjä* meklarille. Palvelukysely sisältää valintaehdot

palvelutarjouksille.

- *Meklari* tallentaa palvelutarjoukset tietokantaansa. Palvelukyselyn saadessaan se etsii kannasta palvelutarjouksia, jotka vastaavat kyselyssä asetettuja vaatimuksia. Soveltuvat tarjoukset palautetaan kysyjälle vastauksena kyselyyn.
- *Asiakas* käyttää palvelua eli lähettää palvelupyyntöjä palvelutarjouksessa ilmoitetulle palvelimelle.
- *Palvelin* tarjoaa tarjoajan ilmoittaman palvelun ja vastaa asiakkaan palvelupyyntöihin.



Kuva 2.1: Meklausroolit [14, kuva 5.1].

Kysyjä saa vastauksena palvelukyselyyn yhden tai useamman palvelutarjouksen, jos valintaehdot täyttäviä tarjouksia löytyi meklarin tietokannasta. Kysyjä voi itse valita, mitä tarjouksista käytetään, tai vaihtoehtoisesti meklari voi valita parhaiten valintaehtoja vastaavan tarjouksen kysyjän puolesta. Tämän jälkeen asiakas voi ottaa yhteyttä suoraan tarjouksessa ilmoitettuun palvelimeen. Meklari ei siis osallistu palvelun käyttöön, vaan toimii ainoastaan markkinapaikkana palveluiden tarjoajille ja tarvitsijoille. Asiakas ja palvelin on tässä erotettu kysyjästä ja tarjoajasta, koska meklattua tietoa voivat käyttää muutkin kuin varsinainen kysyjä ja tarjoaja. Roolit eivät ole sidottuja tiettyihin komponentteihin: sama komponentti voi tarjota yhtä palvelua ja kysyä toista.

Meklareita voidaan kytkeä toisiinsa eli federoida. Federoidussa tilanteessa jokaisella meklarilla on oma alueensa, jossa tapahtuva meklaus hoidetaan paikallisen meklarin kautta. Jos palvelukyselyyn ei löydy sopivia tarjouksia paikalliselta meklarilta, meklari voi lähettää palvelukyselyn eteenpäin siihen kytketyille toisille meklareille. Tällöin kysyjän roolissa toimii lähettävä meklari [14, luku 5.3]. Meklarifederaatiossa palvelukysely voi kulkeutua meklarilta toiselle pitkällekin, jollei kulkeutumista rajoiteta. Palvelukyselyn vastaukset kulkeutuvat samaa reittiä takaisin kysyjälle. Kukin tarjolla oleva palvelu on aina rekisteröity vain yhteen meklariin. Meklarit voivat käyttää paikallista palvelutarjousvälimuistia toiminnan tehostamiseksi, mutta tällöin palvelutarjousten ajantasaisuudesta on huolehdittava erityisen tarkkaan.

2.2 Vaatimuksia meklauspalvelulle

Meklauspalvelujen käyttökohteet voivat poiketa toisistaan suurestikin esimerkiksi siinä, kuinka paljon tarjouksia niihin on rekisteröity ja kuinka suurta asiakasmäärää ne palvelevat. Reaaliaikainen meklaus sulautetussa järjestelmässä ja toisaalta palvelujen hallinta laaja-alaisessa tietoverkossa asettavat erilaisia vaatimuksia käytettävälle meklauspalvelulle [3].

Joitakin yleisiä vaatimuksia meklauspalvelulle voidaan kuitenkin erottaa. Tyypillistä meklaukselle on tiheät päivitykset palvelutarjoustietokantaan. Uusia palvelutarjouksia tehdään, vanhoja poistetaan ja olemassa olevia muutetaan usein tavanomaisiin hakemistopalveluihin verrattuna. Meklauspalvelun täytyy toimia tehokkaasti tällaisessakin käyttöympäristössä.

Tarjottavien palvelujen ominaisuudet, esimerkiksi tulostimen työjonon pituus, vaihtelevat, joten jotkin meklauspalvelut antavat mahdollisuuden käyttää dynaamisia ominaisuuksia. Dynaamisten ominaisuuksien arvoja ei kiinnitetä palvelutarjouksessa, vaan meklari käy kysymässä tarjoajalta senhetkiset arvot palvelukyselyä käsitellessään.

Meklauspalvelun täytyy myös skaalautua vaivattomasti palvelutarjousten ja käyttäjien määrän kasvaessa. Yksi keskitetty meklari on tällöin huono valinta. Meklarien federoinnin avulla saadaan hyödynnettyä paikallisuutta, jolloin skaalautuvuus paranee. Federointi ei varsinaisesti kuulu tämän tutkielman aihepiiriin, vaikka sitä joissakin kohdin sivutaankin.

2.3 CORBA-meklauspalvelu

Seuraavissa luvuissa tutustutaan yksityiskohtaisesti CORBA-meklauspalveluun ja siihen, miten palvelutarjoukset ja -kyselyt CORBA-meklarille muodostetaan.

2.3.1 Yleiskuvaus

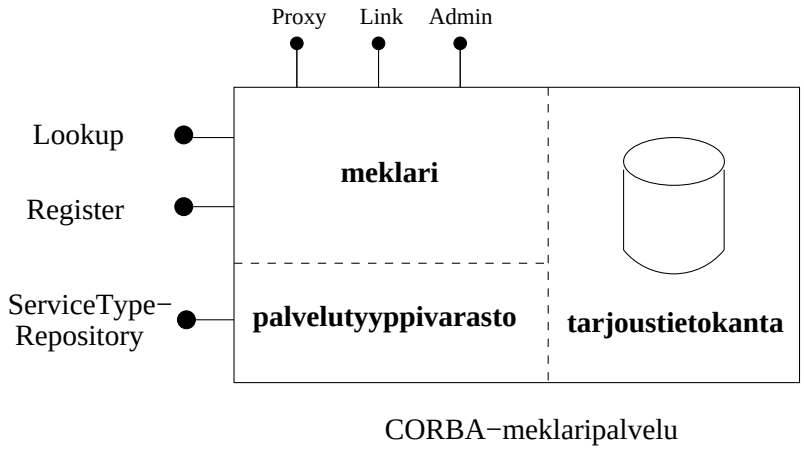
CORBA-meklauspalvelu on Object Management Group (OMG) -konsortion määrittelemä standardi [20], joka on teknisesti yhteneväinen ODP-referenssimallin meklaustoimintostandardin [10] kanssa. Meklauspalvelustandardi määrittelee CORBA-meklauspalvelun ulkoiset rajapinnat ja vaaditun toiminnallisuuden, mutta ei ota kantaa meklarin toteutukseen.

CORBA-meklauspalvelu on osa OMG-konsortion OMA-arkkitehtuuria *(Object Management Architecture)*. OMA on väliohjelmistoarkkitehtuuri, joka pitää sisällään hajautetun oliomallin ja hajautetussa järjestelmässä tarvittavia standardipalveluja. OMA-mallissa olioiden rajapinnat määritellään kieli- ja alustariippumattomalla OMG IDL-määrittelykielellä *(Interface Definition Language)*, joka syntaksiltaan muistuttaa C++-kieltä. Palvelinolio toteuttaa IDL:llä määritellyn rajapinnan tai useita rajapintoja, joiden operaatioita asiakas kutsuu etäältä. CORBA-järjestelmän ORB-komponentti *(Object Request Broker)* välittää operaatiokutsut tietoliikenne-

verkon ylitse periaatteessa sijaintituntumattomasti [22].

Palvelinolioon viitataan olioviitteellä, joka sisältää palvelimen verkko-osoitteen sekä olion tunnuksen. CORBA-meklauspalvelua käytetään nimenomaan olioviitteiden välittämiseen.

CORBA-meklarin rakenne ja sen tarjoamat rajapinnat on esitetty kuvassa 2.2. Palvelutarjoukset ovat kukin jotakin *palvelutyyppiä*, joka määrittelee palvelun luokituksen ja palvelurajapinnan tyypin. Palvelutyyppikuvaukset tallennetaan *palvelutyyppivarastoon*, joka on osa CORBA-meklauspalvelua. Palvelutarjoukset tallennetaan meklarin palvelutarjoustietokantaan.



Kuva 2.2: CORBA-meklarin rakenne ja rajapinnat.

CORBA-standardin mukainen meklari tarjoaa vähintään kolme rajapintaa: `ServiceTypeRepository`-rajapinnan palvelutyyppien rekisteröintiin ja hallintaan, `Register`-rajapinnan palvelutarjousten tekemiseen ja `Lookup`-rajapinnan palvelukyselyjen tekemiseen. Näiden lisäksi meklauspalvelutoteutus voi tarjota `Proxy`-rajapinnan välitystarjouksien käsittelyyn, `Link`-rajapinnan meklarien federointiin ja `Admin`-rajapinnan hallintatoimenpiteitä varten, mutta näitä ei käsitellä tarkemmin tässä tutkielmassa.

2.3.2 Palvelutyypit

CORBA-meklauksessa palvelutarjoukset ovat vahvasti tyypitettyjä: ne edustavat aina jotakin tiettyä palvelutyyppiä. Palvelutyyppi on luokka palveluja, joilla on samanlainen toiminnallisuus ja rajapinta, mutta erilaisia ominaisuuksia. Palvelutyyppi määrää, millä tavoin palvelun ominaisuuksia kuvaillaan palvelutarjouksessa. Uudenlaista palvelua varten suunnittelija laatii palvelutyyppikuvauksen ja tallentaa sen meklarin palvelutyyppivarastoon. Sekä palvelutarjoukset että palvelukyselyt meklarille kohdistuvat aina johonkin etukäteen tallennettuun palvelutyyppiin.

Tarjousten tyypittämisellä saavutetaan kahdenlaista etua. Ensinnäkin palvelutarjoukset saadaan ryhmiteltyä ja niiden sisältämän tiedon muoto määriteltyä siten,

että tarjousten ja kyselyjen automaattinen vertaileminen meklarissa on mahdollista. Samalla estetään väärin muodostettujen tarjousten ja kyselyjen tekeminen. Toiseksi CORBA-meklaritoteutus voi käyttää palvelutyyppiluokitusta hyödyksi nopeuttamaan palvelutarjousten hakua.

Palvelutyyppikuvaus sisältää seuraavat osat:

- nimen, joka on yksikäsitteinen meklarifederaation sisällä,
- mahdolliset *ylityypit*, joista tämä palvelutyyppi periytyy,
- palvelurajapinnan OMG IDL-määrittelyn ja
- *ominaisuustyypit*, jotka määrittelevät, millaisilla ominaisuuksilla palvelua kuvataan.

Osia käsitellään seuraavaksi tarkemmin.

Palvelutyyppi voi periytyä toisesta palvelutyypistä. Periytyminen helpottaa palvelutyyppien järjestämistä loogiseksi hierarkiaksi. Perivä palvelutyyppi tarjoaa vähintään saman palvelun kuin ylityyppinsä, mutta voi tarjota laajempaakin palvelua.

Meklari voi palauttaa vastauksena palvelukyselyyn paitsi pyydettyä palvelutyyppiä myös sen alityyppiä olevia palvelutarjouksia, jollei tätä erillisellä hakupolitiikalla kielletä. Palvelutyyppien periytymiselle on tiukat säännöt: alityypin palvelurajapinnan täytyy olla sama kuin ylityypin tai periytyä ylityypin rajapinnasta, ja alityypin täytyy sisältää vähintään samat ominaisuustyypit kuin ylityypinkin.

Palvelutarjoukseen sisältyviä, palvelua kuvaavia määreitä kutsutaan palvelun *ominaisuuksiksi*. Esimerkiksi tulostinpalvelua kuvaavia ominaisuuksia voisivat olla värillisyys/mustavalkoisuus ja tulostimen nopeus (yksikkönä sivua minuutissa). Ominaisuustyypeillä määritellään, millaisilla ominaisuuksilla palvelua kuvaillaan. Kukin ominaisuustyyppi koostuu seuraavista osista:

- ominaisuuden nimi, yksikäsitteinen palvelutyypin sisällä,
- ominaisuuden OMG IDL-tyyppi ja
- *moodi*, joka määrittelee, onko ominaisuus pakollinen vai vapaaehtoinen ja onko arvo jälkikäteen muutettavissa.

OMG IDL-määrittelykieli sisältää Java-ohjelmointikielen tapaiset perustyypit eri mittaisille kokonaisluvuille, liukuluvuille, merkkijonoille ja totuusarvoille. Muita tyyppejä ovat rajapintatyypit sekä tietueet (`struct`), jotka kokoavat yhteen joukon muita tyyppejä.

Moodin avulla määrätään, onko ominaisuus pakollinen vai vapaaehtoinen. Pakollisuus merkitsee, että palvelun tarjoajan on annettava ominaisuudelle jokin arvo palvelutarjouksessa. Ominaisuuden arvon voi moodin avulla myös asettaa muutettavaksi, jolloin tarjoaja voi muuttaa sen arvoa rekisteröidyssä palvelutarjouksessa poistamatta palvelutarjousta välillä meklarilta.

Kuvassa 2.3 on esimerkki Tulostuspalvelu-palvelutyypin tietosisällöstä pseudokoodinomaisella merkintätavalla. Merkintätavassa varattujen sanojen jälkeen tulee kaksoispiste ja määrittelylauseke päättyy puolipisteeseen. Tyyppien nimet kirjoitetaan isolla alkukirjaimella, muut nimet pienellä. Samaa merkintätapaa käytetään myös myöhemmissä esimerkeissä.

```
palvelutyyppi: Tulostuspalvelu
{
    rajapinta:  IDL/Tulostusrajapinta/1.0;

    pakollinen ominaisuus:    Totuusarvo   väritulostus;
    pakollinen ominaisuus:    Kokonaisluku tulostusnopeus;
    pakollinen ominaisuus:    Merkkijono   sijaintihuone;
    vapaaehtoinen ominaisuus: Kokonaisluku jonon_pituus;
}
```

Kuva 2.3: Esimerkki Tulostuspalvelu-palvelutyypin tietosisällöstä.

CORBA-meklarissa palvelutyypit tallennetaan palvelutyyppivarastoon. Meklari käyttää palvelutyyppivaraston operaatioita palvelutyyppien ja rajapintojen kuvausten ja periytymissuhteiden kysymiseen palvelutarjouksen rekisteröinnin ja palvelukyselyn yhteydessä.

2.3.3 Palvelutarjous

Palvelutarjous kuvaa meklarin kautta asiakkaille tarjottavaa palvelua. CORBA-arkkitehtuuri ei ota kantaa siihen, kuka on vastuussa palvelutarjouksen tarkkuudesta ja oikeellisuudesta. Palvelutarjous sisältää

- palvelutyypin nimen,
- olioviitteen palvelurajapintaan ja
- ominaisuuksien arvot tälle palvelulle.

Ominaisuudet ilmoitetaan nimi-arvo-pareina. Palvelutarjouksessa täytyy olla oikean tyyppinen arvo jokaiselle palvelutyypissä pakolliseksi määrätylle ominaisuudelle. Vapaaehtoisiksi määritellyille ominaisuuksille arvoja ei tarvitse ilmoittaa. Tarjoukseen voi myös lisätä ominaisuuksia, joita ei ole määritelty palvelutyypissä lainkaan; niiden hyödyntäminen on mahdollista vain, jos myös kysyjä on niistä ennalta tietoinen.

Meklaritoteutus voi lisäksi tukea myös *dynaamisia ominaisuuksia*. Dynaamisen ominaisuuden arvoa ei kiinnitetä palvelutarjouksessa, vaan tarjoukseen tallennetaan

olioviite rajapintaan, jonka avulla meklari voi hakua tehdessään kysyä ominaisuuden senhetkisen arvon. Esimerkiksi tulostinpalvelun työjonon pituus voidaan kuvata dynaamisena ominaisuutena, jonka arvon meklari kysyy tulostinpalvelimelta vasta tarjousten haun yhteydessä.

Palvelutarjoukset tallennetaan meklarin tarjoustietokantaan. Se voi ympäristön vaatimuksista riippuen olla toteutettu eri tavoin, esimerkiksi levymuisti- tai pelkästään keskusmuistipohjaisena. Kukin palvelutarjous saa rekisteröinnissä meklarilta tunnuksen, jolla tarjoukseen voidaan viitata, kun sitä halutaan muuttaa tai se halutaan poistaa.

CORBA-meklarin `Register`-rajapinta tarjoaa operaatiot `export`, `modify` ja `withdraw` palvelutarjouksien rekisteröintiin, muuttamiseen ja poistamiseen. Rajapinnan IDL-kuvaus on tutkielman liitteessä B.

2.3.4 Palvelukysely

Palvelukyselyllä kysyjä hakee palvelutarjouksia meklarin tarjoustietokannasta. Kysely kohdistuu aina tiettyyn palvelutyyppiin ja sisältää kysyjän valintaehdot palvelutarjouksille sekä politiikat, jotka rajoittavat haun laajuutta.

Täsmällisemmin palvelukysely sisältää

- palvelutyypin nimen,
- rajoitelausekkeen ominaisuuksille

sekä vapaaehtoisesti

- suositummuuslausekkeen, jonka mukaan löydetyt palvelutarjoukset järjestetään,
- luettelon ominaisuuksista, joiden arvot kysyjä haluaa tietää ja
- hakupolitiikat, joilla säädellään muun muassa haun laajuutta.

Kyselyyn palautetaan vastauksena ainoastaan tarjoukset, jotka ovat oikeaa palvelutyyppiä tai sen alityyppiä ja jotka täyttävät rajoitelausekkeella ilmaistut ehdot. Alityyppien huomioimisen haussa voi kieltää hakupolitiikkasäännöllä.

Rajoitelauseke kirjoitetaan OMG:n rajoitekielellä [20, liite B], joka sisältää aritmeettiset perusoperaattorit $+$, $-$, $*$, $/$, vertailuoperaattorit, loogiset konnektiivit *ja*, *tai*, *ei* ja joukkoonkuulumisoperaattorin. Vain tarjoukset, joille rajoitelausekkeen arvo on tosi, palautetaan vastauksena kyselyyn. Alla on esimerkki rajoitelausekkeesta.

```
väritulostus == true and jonon_pituus < 3
```

Ominaisuuksia, joita ei mainita rajoitelausekkeessa, meklari ei ota lainkaan huomioon palvelutarjouksia etsiessään.

Koska rajoitelausekkeet voivat olla monimutkaisia, jopa yhtälöryhmiä, meklari joutuu useimmiten käymään läpi kaikki palvelutyypiltään sopivat tarjoukset ja laskemaan rajoitelausekkeen totuusarvon erikseen jokaiselle niistä. Palvelykyselyn vastausaika riippuu yleensä lineaarisesti tarjoustietokannan koosta ja rajoitelausekkeen monimutkaisuudesta [13].

Suositummuuslausekkeella säädellään järjestystä, jossa löydetyt palvelutarjoukset palautetaan asiakkaalle. Lausekkeella voidaan valita yksi numeerista tyyppiä oleva ominaisuus, jonka mukaan tarjoukset järjestetään nousevaan tai laskevaan suuruusjärjestykseen. Vaihtoehtoisesti tarjoukset voidaan palauttaa myös satunnaisessa tai löytymisjärjestyksessä. Palautetut palvelutarjoukset sisältävät tilan säästämiseksi vain niiden ominaisuuksien arvot, jotka kysyjä on merkinnyt haluavansa tietää.

Hakupolitiikoilla voidaan säädellä muun muassa haun laajuutta meklarifederaatiossa, vastauksena palautettavien tarjousten enimmäismäärää sekä sitä, otetaanko muutettavissa olevat ja dynaamiset ominaisuudet huomioon haussa.

CORBA-meklarin `Lookup`-rajapinta sisältää ainoastaan `query`-operaation palvelukyselyjen tekemiseen. Operaatiolla on kaikkiaan kuusi syöte- ja kolme tulosparametria (nähtävissä liitteessä B). Kyselyrajapintaa on kritisoitu tarpeettomasta monimutkaisuudesta: yksinkertaisenkin palvelukyselyn tekemiseen tarvitaan usein kymmeniä rivejä ohjelmakoodia. CORBA-meklarin kyselyrajapinta on siis ehkä liiankin joustava ja täydellinen. Palvelukyselyn tyyppitarkistusta ei tehdä käännösaikana, vaan vasta suoritusaikana, mikä hidastaa virheiden löytämistä [16].

2.4 Sovellusesimerkki: turistipalvelu

Pilarcos-projekti kehittää väliohjelmistoratkaisuja organisaatiorajat ylittävien hajautettujen sovellusten hallintaan. Tällaisia sovelluksia ovat muun muassa sähköinen tiedonvälitys ja erilaisten sähköisten palvelujen myyminen yritysten ja yksityiskäyttäjien kesken. Web-palveluilla tähdätään samankaltaisiin päämääriin, mutta eri tekniikoin.

Esimerkkinä usean organisaation laajuisesta hajautetusta sovelluksesta tarkastellaan kuvitteellista turistipalvelua. Turistipalvelu on portaali, joka tarjoaa turisteille erilaisia sähköisiä palveluja, kuten hotellinvaraus, säätiedotukset, autonvuokraus ja niin edelleen. Turistilla on kotimikrossaan tai kämmentietokoneessaan sovellus, jonka avulla hän voi tehdä matkajärjestelyjä. Turistisovellus puolestaan käyttää etäältä jotakin verkossa tarjolla olevaa turistipalvelua. Turistipalvelun käyttö on yleensä maksullista, ja maksu tapahtuu luotetun osapuolen — sähköisen maksupalvelun — välityksellä. Sähköisiä maksupalveluja tarjoavat yleensä pankit ja luottokorttiyhtiöt.

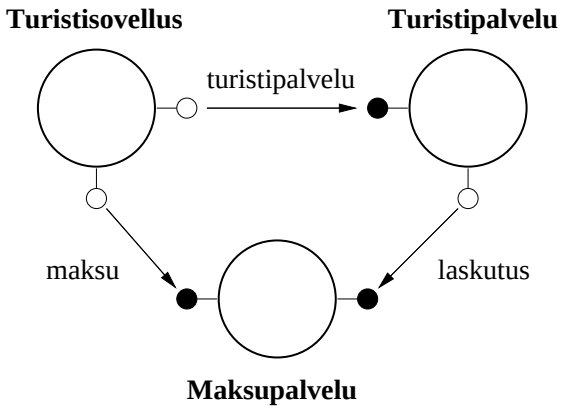
Sekä turistipalvelua että maksupalvelua tarjoaa usea yritys, ja nämä palvelut ovat kaikkien käyttäjien saatavilla tietoverkossa. Turistipalvelua on tarjolla eri alueille, erilaisin palveluvalikoimin ja eri hinnoilla. Kun turisti käynnistää turistisovelluksen, ensimmäinen ongelma on, miten sovellus löytää verkosta sopivan turistipalvelun ja maksupalvelun. Pilarcos-arkkitehtuurissa tähän käytetään julkista meklauspalve-

lua, joka mahdollistaa avoimet ja automatisoidut sähköiset palvelumarkkinat. Turistipalvelun ja maksupalvelun tarjoajat tallennetavat tiedot tarjoamistaan palveluista meklarille, jolloin turistisovellus voi etsiä meklarilta automaattisesti käyttäjän kulloisiakin toiveita vastaavat palvelut.

2.5 Pilarcos-meklaus

Meklauspalvelun käyttö turisti- ja maksupalvelun löytämiseen on mutkikkaampaa kuin aluksi näyttää, koska turistipalvelu ja maksupalvelu ovat sidoksissa toisiinsa. Jotta maksunvälitys tapahtuisi vaaditulla tavalla, turistipalvelun ja maksupalvelun täytyy käyttää yhteisesti sovittua laskutusmenettelyä. Erilaisia laskutusmenettelyjä voivat olla esimerkiksi reaaliaikainen sähköinen tilisiirto tai jälkikäteen veloitettava luottotili. Turistisovelluksen, turistipalvelun ja maksupalvelun on myös käytettävä samaa viestintäprotokollaa tai ainakin tunnettava toistensa käyttämät protokollat. Ei siis riitä, että turistisovellus voi löytää sopivan turisti- ja maksupalvelun, vaan näiden keskinäinen yhteensopivuus on myös varmistettava.

Perinteisessä asiakas-palvelin-mallissa yhteensopivuuden takaaminen on tällaisessa tapauksessa hankalaa. Joko sopivat palveluyhdistelmät on kiinnitettävä etukäteen joustavuuden kustannuksella, tai turistisovelluksen on itse löydettävä sopiva yhdistelmä ja saatava palvelut toimimaan keskenään. Tämä johtuu siitä, ettei asiakas-palvelin-malli kykene kuvaamaan useamman kuin kahden komponentin keskinäisiä suhteita. Pilarcos-malli laajentaa asiakas-palvelin-mallin monenväliseksi malliksi, jossa komponenttien väliset suhteet kuvataan erillisen *arkkitehtuurikuvauksen* avulla.



Kuva 2.4: Turistipalvelun arkkitehtuurikuvaus havainnollistettuna.

Kuva 2.4 esittää turistipalvelun arkkitehtuurikuvausta. Arkkitehtuurissa on kolme roolia: turistisovellus, turistipalvelu ja maksupalvelu. Niiden väliset riippuvuudet on ilmaistu nuolilla merkittyjen sidosten avulla. Kuvan arkkitehtuurikuvaus ilmaisee, että turistisovellus toimii asiakkana sekä turisti- että maksupalvelulle, ja että myös turistipalvelu on maksupalvelun asiakas. Pilarcos-väliohjelmistopalvelut käyt-

tävät arkkitehtuurikuvauksia tämänkaltaisten monenvälisten hajautettujen sovellusten hallintaan.

Koska perinteiset meklauspalvelut perustuvat puhtaasti asiakas-palvelin-malliin, Pilarcos-projektissa on kehitetty uudenlainen, arkkitehtuurikuvauksia hyödyntävä meklausmenettely, jota kutsutaan tässä *Pilarcos-meklaukseksi*. Pilarcos-meklari hakee tarjoustietokannasta palvelutarjouksia kaikkiin arkkitehtuurikuvauksen täyttämättömiin rooleihin, ja palauttaa vastauksena kyselyyn ainoastaan keskenään yhteensopivia palvelutarjousyhdistelmiä. Tällöin riittää, että turistisovellus tekee yhden meklarikyselyn; vastauksena se saa ehdokkaat sekä turistipalveluksi että maksupalveluksi.

Pilarcos-meklaus on kirjoittajan tietämyksen mukaan periaatteeltaan uudenlainen, vaikka sekä arkkitehtuurikuvauksia että meklauspalveluja on erikseen tutkittu paljon. Aikaisemmat komponenttien meklausta tutkineet projektit perustuvat palvelukuvausten [27] tai palvelujen [30, 7] koostamiseen hierarkkisesti, eivät arkkitehtuurikuvausten hyväksikäyttöön. Arkkitehtuurikuvauksiin liittyvää tutkimusta esitellään luvussa 3. Varsinaiseen Pilarcos-meklaukseen paneudutaan luvussa 4.

Luku 3

Arkkitehtuurikuvaukset

Edellisessä luvussa esitelty Pilarcos-meklaus käyttää hyödyksi palvelutyyppi- ja arkkitehtuurikuvauksia, jotka kuvaavat hajautetun sovelluksen komponenttien väliset suhteet. Monet Pilarcos-meklauksen käsitteet, kuten ominaisuus ja palvelutyyppi, ovat sisällöltään samankaltaisia mutta laajempia kuin CORBA-meklarissa. Tässä luvussa kuvaillaan Pilarcos-meklarin käyttämät arkkitehtuurikuvaukset ja niiden rakenneosat.

3.1 Yleiset suunnitteluperiaatteet

Pilarcos-arkkitehtuurikuvauksilla on kolme päätavoitetta: helpottaa organisaatioiden välisten hajautettujen järjestelmien suunnittelua ja toteuttamista, edistää metatiedon eli kuvausten ja myös komponenttien uudelleenkäyttöä, ja tukea suoritusaikaista palvelujen meklausta ja hallintaa. Näistä tavoitteista voidaan johtaa seuraavat periaatteet, joita arkkitehtuurikuvausten suunnittelussa on noudatettu.

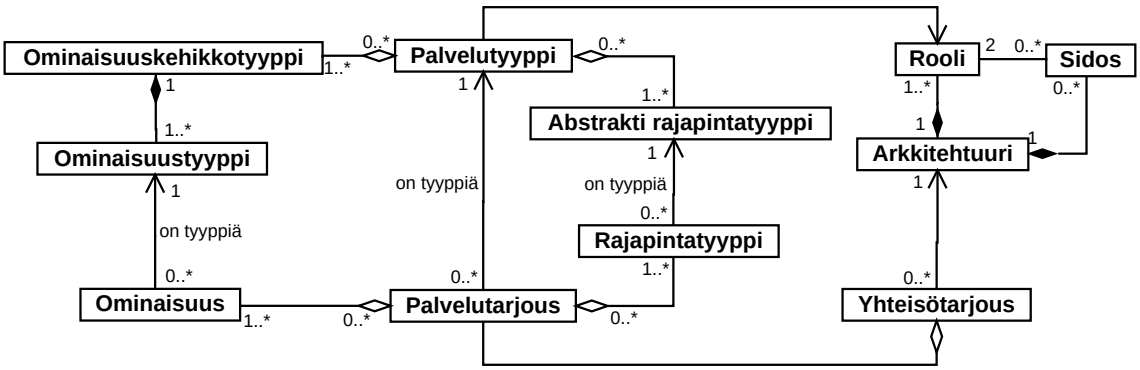
Sovellussuunnittelun helpottamiseksi arkkitehtuurikuvausten käsitteiden täytyy vastata ohjelmistosuunnittelijalle tuttuja rakenteita. Käsitteille on oltava selkeät vastineet myös toteutuksissa, jotta ohjelmointi olisi suoraviivaista.

Koska tavoitteena on sekä kuvausten että komponenttien mahdollisimman hyvä uudelleenkäytettävyys, on palvelu- ja arkkitehtuurikuvaukset erotettava selkeästi varsinaisista ohjelmistokomponenteista. Kenen tahansa on pystyttävä tarjoamaan palvelukuvausten mukaisia palveluja muista palveluntarjoajista riippumatta, jotta palvelumarkkinat olisivat aidosti avoimet. Kuvausten rakenneosien on lisäksi oltava sopivan kokoisia uudelleenkäytettävyyden kannalta.

Koska Pilarcos-väliohjelmistopalvelut käyttävät arkkitehtuurikuvauksia suoritusaikaiseen palvelujen meklaukseen ja hallintaan, kuvausten on oltava riittävän täsmällisiä ja yksityiskohtaisia koneellista käsittelyä ajatellen. Kuvausten täytyy myös rakenteeltaan soveltua työkaluavusteiseen suunnitteluun.

3.2 Käsitteiden väliset suhteet

Kuva 3.1 esittää Pilarcos-meklauksen käsitteiden suhteet UML-luokkakaaviona. Pystysuunnassa kuva jakautuu karkeasti kahtia: ylhäällä sijaitsevat tyyppiluokat, alimpana kaavain- eli toteutusluokat. Vaakasuunnassa kuvassa voidaan erottaa neljä osaa. Ominaisuuksiin liittyvät käsitteet ovat vasemmalla, palvelujen ja rajapintojen kuvaamiseen käytetyt käsitteet keskellä, ja arkkitehtuuriin liittyvät käsitteet oikealla.



Kuva 3.1: Pilarcos-meklauksen käsitteet UML-luokkakaaviona.

Käsitteet käydään seuraavaksi läpi alkaen palvelutyyppien rakenneosista, ominaisuuskehikoista ja rajapintatyypeistä. Palvelutyyppien määrittelyn jälkeen käsitellään arkkitehtuurikuvauksia, jotka yhdistävät palvelutyyppejä toisiinsa kokonaisuuksiksi.

3.3 Palvelujen kuvaaminen

CORBA-meklauspalvelussa palvelutyyppi sisältää yhden rajapinnan ja joukon ominaisuustyyppejä (luku 2.3.2). Pilarcos-meklauksessa palveluun voi liittyä useita eri osapuolille tarjottuja rajapintoja. Esimerkiksi turistipalvelun (luku 2.4) tukena toimiva maksupalvelu tarjoaa eri rajapinnat laskuttaja- ja maksajaosapuolille. Tästä syystä Pilarcos-mallissa palvelutyypit koostuvat useista, toisiinsa liittyvistä rajapinnoista ja ominaisuuskehikoista. Ominaisuuskehikot ja rajapinnat määritellään uudelleenkäytettävillä tyyppikuvauksilla, joita käsitellään seuraavaksi.

3.3.1 Ominaisuuskehikot

Ominaisuuskehikko kokoaa yhteen joukon loogisesti toisiinsa liittyviä ominaisuustyyppejä. Esimerkiksi turistipalvelun käyttöön liittyvät ominaisuustyypit – kohdealue, hinta, saatavissa olevat palvelut, ja niin edelleen – voidaan koota yhteen ominaisuuskehikkoon. *Ominaisuuskehikkotyyppi* koostuu

- nimestä ja
- joukosta ominaisuustyyppejä.

Ominaisuustyypit ovat samankaltaisia kuin CORBA-meklarissa.

Ominaisuuskehikkotyyppejä voidaan käyttää useissa palvelutyypeissä, joten ne kannattaa määritellä siten, että ne sisältävät vain kiinteästi toisiinsa liittyviä ominaisuustyyppejä. Esimerkiksi kaikki laskunmaksuun liittyvät ominaisuustyypit kannattaa koota yhteen ominaisuuskehikkotyyppiin ja viestintäprotokollaan liittyvät toiseen. Kuvassa 3.2 on esimerkkejä ominaisuuskehikkotyyppien tietosisällöstä. Nämä esimerkit, kuten myöhemmätkin, perustuvat luvussa 2.4 esiteltyyn turistipalvelusovellukseen.

```
ominaisuuskehikkotyyppi: TuristipalveluOminaisuudet
{
    Merkkijono                              palvelualue;
    Vaihtoehto (HOTELLIT, SÄÄ, AIKATAULUT)  palvelut;
    Reaaliluku                              hinta;
    Vaihtoehto (ETUKÄTEEN, JÄLKIKÄTEEN)     maksutavat;
}

ominaisuuskehikkotyyppi: MaksuOminaisuudet
{
    Vaihtoehto (PANKKITILI, VISA, AMEX, MASTERCARD)  maksuväline;
    Vaihtoehto (I, II, III)                          turvaluokitus;
}
```

Kuva 3.2: Esimerkkejä ominaisuuskehikkotyypeistä.

Ominaisuuskehikkojen taustalla on Pilarcos-projektin ajatus hajautettujen järjestelmien politiikkaperustaisesta hallinnasta. Siinä politiikkojen kuvataan politiikkakehikkojen avulla [31]. Tässä tutkielmassa tarkastellaan ominaisuuskehikkoja vain meklauksen kannalta.

3.3.2 Abstraktit rajapinnat

Yleisten suunnitteluperiaatteiden mukaisesti Pilarcos-arkkitehtuurissa erotetaan palveluiden kuvaukset ja tekniset toteutukset mahdollisimman täydellisesti toisistaan. Siksi — toisin kun CORBA-meklarissa — Pilarcos-meklarin palvelukuvauksissa ei määritellä palvelun rajapintojen yksityiskohtia, vaan ainoastaan niiden tarjoama looginen toiminnallisuus. Tällä tavoin määriteltyjä rajapintoja kutsutaan *abstrakteiksi rajapinnoiksi*.

Abstraktin rajapinnan tyyppi on yksinkertaisinta määritellä luokaksi toiminnallisuudeltaan samanlaisia mutta teknisesti eriäviä konkreettisia tyyppikuvauksia. Suo-

ritusaikainen koneisto voi tällöin käsitellä abstrakteja rajapintatyyppejä pelkkinä tunnettuina niminä näille luokille.

Sovellussuunnittelua ja kuvausten uudelleenkäyttöä varten abstraktien rajapintatyyppien semantiikka on kuitenkin määriteltävä tarkemmin. Tässä tutkielmassa käytetään seuraavanlaista kuvaustapaa: *Abstrakti rajapintatyyppi* koostuu

- nimestä,
- käyttötavasta (operaatiokutsu, vuo, tai signaali) ja
- rajapinnan tarjoaman toiminnallisuuden kuvauksesta, joka voi olla epämuodollinen tai jonkin formaalin kuvauskielen mukainen.

Sovellusesimerkin maksupalveluun liittyy laskunmaksurajapinta, joka tarjoaa operaatiot laskun sisällön pyytämiseen ja laskun maksamiseen. Turistipalvelu taas tarjota turistipalvelurajapintaa. Esimerkkejä abstraktin rajapintatyypin määritelmän tietosisällöstä on kuvassa 3.3.

```
abstrakti rajapintatyyppi: TuristipalveluRajapinta
{
    käyttötapa: operaatiokutsu
    operaatiot: "aloitaIstunto", "teeKysely", "teeVaraus",
                "lopetaIstunto"
}

abstrakti rajapintatyyppi: LaskunmaksuRajapinta
{
    käyttötapa: operaatiokutsu
    operaatiot: "haeLasku", "maksaLasku", "kieltäydyLaskusta"
}
```

Kuva 3.3: Esimerkkejä abstrakteista rajapintatyypeistä.

Rajapintaan liittyvän toiminnallisuuden kuvaukseen ei tässä puututa enempää, vaikka se sovellussuunnittelun kannalta onkin olennaista. Mainittakoon, että toiminnallisuuden kuvaukseen voitaisiin käyttää esimerkiksi OMG:ssa kehitteillä olevan MDA-arkkitehtuurin *(Model Driven Architecture)* alustariippumattomia malleja *(platform independent models)* [21].

Abstrakteista rajapinnoista voi olla useita erilaisia teknisiä toteutuksia, esimerkiksi CORBA- tai EJB-rajapintoina. Saman abstraktin rajapinnan poikkeavatkin tekniset toteutukset voivat olla yhteensopivia, jos niiden välille on mahdollista asentaa sovitin. Pilarcos-meklauksessa rajapintojen yhteensopivuussuhteet tallennetaan tyyppivarastoon ja otetaan huomioon palvelutarjouksia valittaessa. Tähän palataan luvussa 4.1. Abstraktien rajapintojen käytön ansiosta palvelumäärittelyt eivät ole sidottuja toteutustekniikkaan eivätkä ohjelmistoversioihin, vaan ovat monikäyttöisiä ja pitkäikäisiä.

3.3.3 Palvelutyypit

Palvelutyypillä on Pilarcos-meklarissa samanlainen rooli kuin CORBA-meklarissa (luku 2.3.2): se edustaa tiettyä palveluluokkaa ja määrittelee kyseiseen luokkaan kuuluvien palvelutarjousten muodon. Tavoitteena on, että riippumaton palvelun-tuottaja voi toteuttaa palvelun ja tehdä siitä palvelutarjouksen pelkästään palvelu-tyyppimäärittelyn perusteella.

Palvelutyyppi määrittelee yhden, itsenäisesti toteutettavan ja käytettävän palve-lukokonaisuuden. Toisin kuin CORBA-meklarissa, Pilarcosissa palvelutyyppiin voi liittyä useita eri tahoille tarjottuja rajapintoja. Kuten aiemmin mainittiin, sovellus-esimerkin maksupalvelu tarjoaa erilliset rajapinnat laskuttajaa ja maksajaa varten. Palvelutyyppi liittää ominaisuuskehikot rajapintoihin, jolloin rajapinnoista muodos-tuu eri tahoille näkyviä osapalveluja.

Muodollisesti *palvelutyyppi* muodostuu

- nimestä,
- joukosta nimettyjä abstrakteja rajapintoja *(rajapintarooli, abstrakti rajapinta-tyyppi, rajapinnan nimi)*-kolmikoina ja
- joukosta nimettyjä ominaisuuskehikoita liitettyinä rajapintoihin, eli *(ominai-suuskehikko, ominaisuuskehikon nimi, lista rajapintojen nimiä)*-kolmikoita.

Kuvassa 3.4 on esimerkki Turistipalvelu- ja Maksupalvelu-palvelutyyppien tietosi-sällöistä.

Palvelutyyppi kertoo, mitä abstrakteja rajapintoja kyseinen palvelu tarjoaa muil-le tahoille ja myös, mitä rajapintoja se itse tarvitsee toimiakseen. Palvelutyyppi siis määrittelee abstraktilla tasolla palvelun *sopimuksen* ympäristönsä kanssa, mi-kä mahdollistaa palvelutyypin toteuttavan komponentin riippumattoman uudelleen-käytön [29]. Käytetyt ja tarjotut rajapinnat erotellaan palvelutyypissä rajapintaroo-lin ("`tarjottu`", "`käytetty`") avulla. Ero on looginen, eikä välttämättä edellytä ra-japinnan toteuttamista epäsymmetrisesti. Mikäli rajapinnan käyttäjän ja tarjoajan välillä ei haluta tehdä lainkaan eroa, on mahdollista lisätä tyyppimäärittelyihin eril-linen rajapintarooli symmetristä tilannetta varten.

Kullekin palvelutyyppiin sisältyvälle abstraktille rajapinnalle annetaan oma ni-mi, jotta palvelutyyppi voisi sisältää useita saman tyyppisiä mutta erillisiä rajapin-toja. Pilarcos-arkkitehtuurissa nimillä on toinenkin käyttötarkoitus: niillä viitataan kyseisiin rajapintoihin työkaluissa ja sovellusohjelmakoodissa.

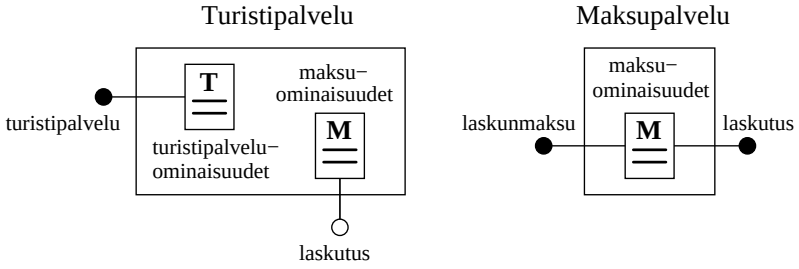
Palvelutyyppiin sisältyvät ominaisuuskehikot nimetään erikseen samalla tavoin kuin abstraktit rajapinnat. Ominaisuuskehikoiden yhteydessä määritellään myös, mihin palvelutyypin rajapinnoista ne liittyvät. Ominaisuuskehikko-rajapinta-liitok-set ovat Pilarcos-meklauksen erityispiirre, joka mahdollistaa monenvälisten riippu-vuussuhteiden automaattisen käsittelyn meklarissa, kuten myöhemmin nähdään.

Ominaisuuskehikko-rajapinta-liitokset ilmaisevat, mille ominaisuuksille eri ra-japintoja käyttävät tahot voivat asettaa ehtoja. Kuvan 3.4 Maksupalvelu-palvelu-

```
palvelutyyppi: Turistipalvelu
{
    tarjottu rajapinta: TuristipalveluRajapinta turistipalvelu;
    käytetty rajapinta: LaskutusRajapinta       laskutus;

    ominaisuuskehikko: TuristipalveluOminaisuudet turistipalveluominaisuudet
        liittyy rajapintaan: turistipalvelu;

    ominaisuuskehikko: MaksuOminaisuudet maksuominaisuudet
        liittyy rajapintaan: laskutus;
}


palvelutyyppi: Maksupalvelu
{
    tarjottu rajapinta: LaskutusRajapinta laskutus;
    tarjottu rajapinta: MaksuRajapinta    laskunmaksu;

    ominaisuuskehikko: MaksuOminaisuudet maksuominaisuudet
        liittyy rajapintoihin: laskutus, laskunmaksu;
}
```

Kuva 3.4: Esimerkkejä palvelutyyppimäärittelyjen tietosisällöstä.

tyypissä ”maksuominaisuudet”-ominaisuuskehikko liittyy sekä ”laskutus”- että ”laskunmaksu”-rajapintoihin. Näin ollen molempiin rajapintoihin liittyvät tahot voivat asettaa omat ehtonsa maksupalvelun maksuominaisuuksille (kuten käytetylle maksuvälineelle). Sen sijaan Turistipalvelu-palvelutyypin ominaisuuskehikot liittyvät eri rajapintoihin, jolloin niille annetut ehdot eivät vaikuta toisiinsa. Kuva 3.5 havainnollistaa ominaisuuskehikkojen liittymistä rajapintoihin. Palvelutyypin tarjoamat rajapinnat on merkitty mustatulla ja sen käyttämät rajapinnat avoimella ympyrällä.



Kuva 3.5: Rajapintojen ja ominaisuuskehikkojen liittyminen toisiinsa.

Pilarcos-meklauksessa myös palvelun kysyjällä on oma palvelutyyppinsä, joka on määritellään samalla tavoin kuin palvelun tarjoajien palvelutyypit. Palvelukyselyssä kysyjä antaa erillisen palvelutarjouksen asemesta oman "palvelutarjouksensa", jolloin kysyjä ja tarjoaja ovat symmetrisessä asemassa. Tätä käsitellään tarkemmin luvussa 4.

Käytännössä tiettyihin abstrakteihin rajapintatyyppeihin liittyy usein kiinteästi joitakin ominaisuuskehikkotyyppejä. Sovellusesimerkin MaksuOminaisuudet-kehikkotyyppi sisältää ominaisuuksia, jotka kuvailevat MaksuRajapinta- ja LaskutusRajapinta-tyyppisten rajapintojen kautta tarjottua toiminnallisuutta. Voisi ajatella, että ominaisuuskehikkotyypit ja abstraktit rajapintatyypit kytkettäisiin jollain tavoin toisiinsa jo rajapintatyyppien määrittelyn yhteydessä eikä vasta palvelutyypeissä. Tähän palataan sidosten yhteydessä luvussa 3.4.3.

Samaan tapaan kuin CORBA-meklauksessa voitaisiin Pilarcos-meklauksessakin periaatteessa sallia palvelutyyppien periminen eli alityypitys. Palvelutyyppien periytyminen edellyttäisi myös ominaisuuskehikkojen ja rajapintojen periytymistä. Aihetta ei ole tutkittu Pilarcos-projektissa, joten sitä ei käsitellä enempää.

3.4 Arkkitehtuurien kuvaaminen

Arkkitehtuurikuvaus kokoaa yhteen joukon palvelutyyppejä liittäen ne toisiinsa yhdeksi kokonaisuudeksi. Seuraavaksi käsitellään arkkitehtuurikuvauksen yleistä rakennetta ja se rakenneosia, rooleja ja sidoksia.

3.4.1 Arkkitehtuurit

Arkkitehtuurikuvaus koostuu rooleista, jotka liittyvät toisiinsa sidoksilla. Kuhunkin rooliin liittyy palvelutyyppi, ja sidokset liittävät palvelutyyppien rajapintoja toisiinsa. Myös mahdollisella kysyjällä eli asiakaskomponentilla on oma roolinsa arkkitehtuurissa. Arkkitehtuurikuvaus muodostaa siten kokonaisen useista komponenteista koostuvan yhteisön mallin.

Arkkitehtuurikuvausten käsitteet ovat lähtöisin ODP-viitemallin tavoitenäkökulmakielestä, jolla mallitetaan organisaatiotason rooleja ja niiden välisiä vuorovaikutuksia [12]. Roolit voivat ODP-viitemallisssa edustaa ihmisiä tai tietojärjestelmiä, joita molempia kutsutaan olioiksi. Roolit määritellään erillään olioista; oliot voidaan sijoittaa rooleihin joko etukäteen tai vasta järjestelmän tai prosessin suoritusaikana. Toimivat oliot muodostavat keskenään *yhteisön*. Pilarcos-arkkitehtuuri noudattaa samaa perusajatusta: arkkitehtuurikuvaukset ja niiden rooleissa toimivat palvelutoteutukset ovat toisistaan riippumattomia.

Pilarcosissa arkkitehtuurikuvaus tai lyhyemmin *arkkitehtuuri* koostuu

- nimestä,
- joukosta rooleja ja

- joukosta roolien välisiä sidoksia.

Kuvassa 3.6 on esimerkki arkkitehtuurimäärittelyn tietosisällöstä. Laajemmassa Pilarcos-arkkitehtuurissa arkkitehtuurikuvaukseen voi sisältyä myös ominaisuuksia ja politiikkoja, jotka vaikuttavat esimerkiksi roolien täyttösääntöihin [15]. Näitä ei kuitenkaan käsitellä tässä tutkielmassa.

```
arkkitehtuuri: TuristipalveluArkkitehtuuri
{
    rooli: TuristiSovellus asiakas;
    rooli: TuristiPalvelu  palvelin;
    rooli: MaksuPalvelu    maksunvälittäjä;

    sidos: asiakas.turistipalvelu -> palvelin.turistipalvelu;
    sidos: asiakas.laskunmaksu    -> maksunvälittäjä.laskunmaksu;
    sidos: palvelin.laskutus      -> maksunvälittäjä.laskutus;
}
```

Kuva 3.6: Esimerkki arkkitehtuurimäärittelyn tietosisällöstä.

Arkkitehtuurikuvausten rakenteet ovat sukua arkkitehtuurikielille [17]. Arkkitehtuurikielet on tarkoitettu suurten ohjelmistojen rakenteen mallittamiseen ja analysointiin. Pilarcosin arkkitehtuurikuvaus vastaa abstraktilla tasolla arkkitehtuurikielten konfiguraatiota, rooli komponenttia ja sidos liitintä. Arkkitehtuurikielet on kuitenkin tarkoitettu kiinteiden kokoonpanojen kuvaamiseen, ei suoritusaikaiseen komponenttien valintaan, mistä syystä ne eivät ole suoraan sovellettavissa Pilarcos-arkkitehtuuriin.

Pilarcos-arkkitehtuurikuvaukset on tarkoitettu laadittavaksi graafisella suunnittelutyökalulla, joka tukee tyyppimäärittelyjen selaamista ja käyttöä julkisesta tyyppitietovarastosta. Vaikka suunnittelutyökalua ei ole vielä toteutettu, palvelutyyppimäärittelyt ja arkkitehtuurikuvaukset soveltuvat hyvin käytettäviksi myös tekstuaalisina määrittelyinä.

3.4.2 Roolit

Kuten edellä mainittiin, rooli-käsite on peräisin ODP-viitemallin tavoitenäkökulmakielestä. Pilarcos-arkkitehtuurissa kukin rooli edustaa yhtä arkkitehtuurissa toimivaa palvelin- tai asiakaskomponenttia. Rooli koostuu

- palvelutyypin nimestä ja
- roolin nimestä.

Jokaiseen rooliin liittyy siis täsmälleen yksi palvelutyyppi, joka määrää rooliin sijoittuvan komponentin ulospäin näkyvät rajapinnat ja ominaisuuskehikot.

Palvelutyypit ovat riippumattomia arkkitehtuureista, joissa niitä käytetään. Samaa palvelutyyppiä voidaan käyttää eri rooleissa yhdessä tai useammassa arkkitehtuurissa. Koska palvelun tuottajien täytyy pystyä toteuttamaan palvelu periaatteessa pelkästään palvelutyypin avulla, komponenttien täytyy olla riippumattomia käytetystä arkkitehtuureista. Tällä parannetaan komponenttien uudelleenkäytettävyyttä. Sama komponentti voi toimia myös samanaikaisesti useissa rooleissa eri arkkitehtuureissa.

Jos komponentit sidottaisiin tiettyihin arkkitehtuureihin, niiden suunnittelussa olisi helpompi ottaa huomioon kokonaisen yhteisön toiminta. Komponentit voitaisiin myös testata osana yhteisöä. Samalla komponenteista kuitenkin tulisi tiukemmin toisistaan riippuvia ja arkkitehtuureista vaikeampia muuttaa, minkä vuoksi tämä suunnitteluvaihtoehto hylättiin. Miten sitten voidaan varmistua arkkitehtuurien oikeellisuudesta? Käytännössä arkkitehtuurikuvauksen laatijan vastuulla on varmistaa, että arkkitehtuurin eri palvelutyypit todella ovat yhteensopivia. Tässä voidaan käyttää apuna suunnittelutyökaluja, kenties myös formaaleja menetelmiä.

Koska komponentit viestivät ulkomaailman kanssa ainoastaan rajapintojensa kautta, riittää, että rajapinnat ja protokollat kuvataan riittävän tarkasti ja varmistetaan sitten eri komponenttien rajapintojen ja protokollien yhteensopivuus. Wright-arkkitehtuurikieli [1], jossa komponenttien käyttäytyminen mallitetaan formaalisti CSP-kielellä [8], perustuu tähän ajatukseen. Wrightissä mallitetaan erikseen komponentin sisäinen käyttäytyminen ja sen rajapintojen eli porttien käyttäytyminen. Komponentin ja sen porttien yhteentoimivuus verifioidaan CSP-mallintarkistustyökalulla. Komponentteja kytketään toisiinsa konfiguraatioiksi eli kokoonpanoiksi kiinnittämällä niiden portteja yhteen liittimien avulla. Liittimet kuvastavat erilaisia viestintäkanavia, ja myös niiden toiminta mallitetaan CSP:llä.

Konfiguraatio verifioidaan tarkistamalla kaikkien sen liittimien ja porttien protokollien yhteentoimivuus. Konfiguraation toiminta ei muutu, vaikka jonkin komponentin sisäistä toimintaa muutetaan, kunhan komponentin portit pysyvät ennallaan. Monimutkaisia järjestelmiä mallitettaessa tämä vähentää verifioinnissa tarvittavan laskennan määrää. Se myös edistää aitoa komponenttiperustaista ohjelmistosuunnittelua, jossa komponentin kaikki riippuvuudet, ulkoiset rajapinnat ja protokollat ovat tarkasti määritellyt [29]. Wrightin malli sopii hyvin Pilarcos-arkkitehtuuriin, jossa komponenttien vaihdettavuus on keskeistä. Wright antaa samalla esimerkin siitä, miten Pilarcos-arkkitehtuurikuvauksia olisi mahdollista mallittaa ja verifioida formaalein menetelmin.

Palataan vielä roolin ja palvelutyypin suhteeseen. Määriteltäessä rooli Pilarcos-arkkitehtuurikuvauksessa tulee rooliin liittyvän palvelutyypin abstrakteista rajapinnoista samalla roolin rajapintoja. Rajapinnat ja niihin liittyvät ominaisuuskehikot ovat roolin ainoat yhteydet toisiin rooleihin. Rooleja kytketään yhteen sitomalla niiden rajapintoja ja samalla myös ominaisuuskehikoita toisiinsa; tätä käsitellään seuraavassa luvussa.

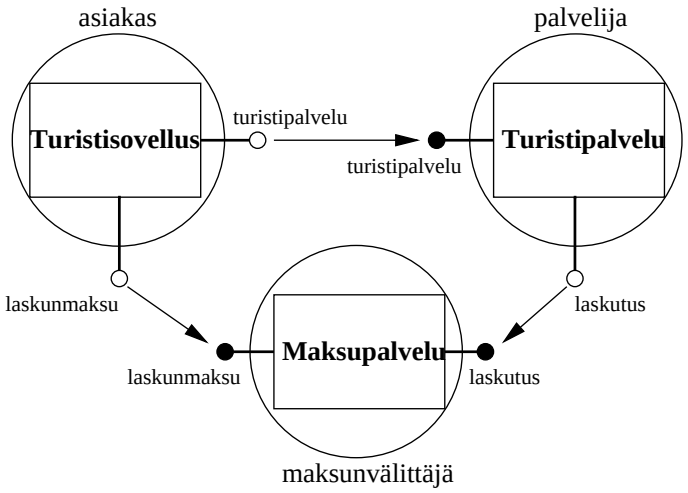
Jos rooliin voisi liittyä useampi kuin yksi palvelutyyppi, roolin täyttävän komponentin pitäisi toteuttaa ne kaikki. Jokaiselle palvelutyypille täytyisi tällöin kuitenkin tehdä omat palvelutarjoukset, mikä ei ole järkevää. Palvelutyyppien yhdistäminen

voitaisiin ehkä toteuttaa paremmin sallimalla palvelutyyppien moniperiytyminen.

3.4.3 Sidokset

Sidos liittää toisiinsa yhden roolin tarjoaman ja toisen roolin käyttämän rajapinnan. Rooleja, joiden välillä on sidos, kutsutaan *naapureiksi*. Sidos määritellään kahtena *(roolin nimi, roolin rajapinnan nimi)*-parina, joissa rajapintojen on oltava samaa abstraktia rajapintatyyppiä. Lisäksi toisen roolin on tarjottava ja toisen käytettävä kyseistä rajapintatyyppiä.

Kuten edellä kerrottiin, roolin rajapinnat määräytyvät palvelutyypistä, joten sidos liittää toisiinsa yhden palvelutyypin tarjoaman ja toisen käyttämän rajapinnan. Kuva 3.7 esittää sidoksia turistipalveluarkkitehtuurissa. Roolit on merkitty ympyröillä, palvelutyypit suorakaiteilla ja sidokset nuolilla.



Kuva 3.7: Sidokset turistipalveluarkkitehtuurissa.

Sidoksilla on arkkitehtuurikuvauksessa kahtalainen merkitys. Ensinnäkin, kun arkkitehtuurikuvausta vastaavaa yhteisöä luodaan, toisiinsa sidottuihin rooleihin sijoittuvien komponenttien välille luodaan viestintäkanava, kuten esimerkiksi TCP/IP-yhteys tai CORBA-väliohjelmistoa käytettäessä IIOP-yhteys. Rajapintojen välinen viestiliikenne kulkee tämän yhteyden kautta. Kanavanmuodostus tapahtuu vasta meklauksen jälkeen, joten sitä ei käsitellä tässä tutkielmassa. Pilarcos-projektissa on tutkittu ja prototyypitetty yhteisön ja kanavien muodostusmekanismeja [31].

Toiseksi, sidos liittää toisiinsa myös rajapintoihin liittyvät samantyyppiset ominaisuuskehikot. Tämä on oleellista Pilarcos-meklauksen kannalta: etsiessään tarjousyhdistelmiä meklari varmistaa, että toisiinsa sidoksella liitettyjen kehikkojen ominaisuusarvot ovat yhteensopivia. Myös yksinkertaisia rajoitteita voidaan käyttää ominaisuusarvoina, joten tämä mekanismi sallii palvelutarjousten asettaa rajoitteita naapurirooleihin ehdolla oleville palvelutarjouksille. Tähän palataan luvussa 4.

Arkkitehtuurikuvauksia suunniteltaessa on ajateltu, että ne laaditaan erityisellä

työkalulla. Työkalu voi tarkistaa, että sidottuihin rajapintoihin liittyvät ominaisuuskehikot ovat samaa tyyppiä. Olisi myös mahdollista sisällyttää ominaisuuskehikot rajapintatyypin määritelmään, jolloin erillistä tarkistusta ei tarvittaisi. Rajapintatyypistä tulisi tällöin kokonaisuus, eräänlainen pienoispalvelutyyppi. Tämä rajoittaisi jonkin verran rajapinta- ja ominaisuuskehikkotyyppien yhdistelyä, mutta tekisi rajapintatyyppien kuvaukset täydellisemmiksi ja siten helpommin uudelleenkäytettäviksi. Tällainen muutos olisi todennäköisesti kannattava ja se kuuluu jatkokehityskohteisiin.

Tässä tutkielmassa sidoksia käsitellään ainoastaan kahdenvälisinä. Ne voidaan kuitenkin yleistää helposti myös yhdestä-moneen- tai monesta-yhteen-tyyppisiksi, jolloin ne vastaavat ryhmälähetystä tai yhtä vastaanottajaa useaa lähettäjää kohti. Ominaisuuskehikkojen väliset sidokset muodostuvat aivan samoin kuin kahdenvälisessä tapauksessa. Monenväliset sidokset kuuluvat myös mahdollisiin jatkokehityskohteisiin.

Sidoksiin itseensä voisi liittää ominaisuuskehikkoja ja valintaehtoja, joiden avulla olisi mahdollista muodostaa eri tyyppisiä kanavia komponenttien välille. Kanavat itsessään voisivat tällöin olla meklattavia olioita [28]. Tämä vaatisi kuitenkin paljon lisätyötä kanavien rakenteen ja kanavanmuodostusmekanismin osalta, eikä se kuulu tämän tutkielman aihepiiriin.

Luku 4

Pilarcos-meklaus

Tässä luvussa kuvaillaan, millä tavoin Pilarcos-meklauksessa tehdään palvelutarjouksia ja palvelukyselyjä ja millä tavoin meklari käsittelee niitä. Erityisesti kiinnitetään huomiota eroihin CORBA-meklaukseen verrattuna. Pilarcos-meklaria varten on kehitetty uudenlainen algoritmi palvelutarjousyhdistelmien etsintään. Algoritmin avulla Pilarcos-meklauksesta saadaan riittävän tehokasta ongelman suuresta laskennallisesta vaativuudesta huolimatta.

4.1 Tyyppivarasto

Tyyppivarasto, toiselta nimeltä tyyppitietovarasto, on palvelu, joka välittää tyyppitietoa hajautetussa järjestelmässä. Meklari käyttää tyyppivarastoa muun muassa palvelutyyppien ja rajapintojen yhteensopivuustarkistuksiin. Tyyppimäärittelyt tallennetaan tyyppivarastoon ohjelmallisesti tai hallintatyökalua käyttäen.

CORBA-meklarin palvelutyyppivarasto (luku 2.3.2) on esimerkki yksinkertaisesta tyyppivarastosta, joka käsittelee ainoastaan palvelutyyppejä. CORBA-arkkitehtuurissa rajapintakuvaukset tallennetaan erilliseen rajapintavarastoon [20], joka niin ikään on eräänlainen tyyppivarasto. Pilarcos-mallissa tyyppivarasto on yleinen tyyppitiedon hallintapalvelu, joka on toiminnaltaan ODP-referenssimallin tyyppivarastotoiminnon [11] kaltainen.

Tyyppivarasto tarjoaa rajapinnat, joiden avulla tyyppikuvauksia ja tyyppien välisiä suhteita voidaan rekisteröidä ja kysellä. Tyyppivarastossa säilytetään muun muassa kaikkia luvussa 3 esiteltyjä tyypimäärittelyjä eli ominaisuuskehikko-, rajapinta- ja palvelutyyppikuvauksia sekä arkkitehtuurikuvauksia. Tyyppivarasto pitää yllä tyyppikuvausten nimiavaruutta ja tietoja tyyppien välisistä suhteista. Pilarcos-mallissa tyyppivarasto toimii lisäksi tyyppitiedon julkaisupaikkana: sopivia työkaluja käyttämällä ohjelmistosuunnittelijat voivat selailla ja ottaa käyttöön jo julkaistuja tyyppikuvauksia tai julkaista omiaan. Tyyppivaraston ei silti tarvitse olla globaali palvelu, vaan tyyppivarastot voidaan federoida kuten meklaritkin.

Perinteisesti — kuten CORBA-palvelutyyppivarastossa — tyyppien väliset suhteet ovat olleet periytymissuhteita: alityyppiä voidaan käyttää sen ylityypin asemes-

ta tietyin rajoituksin. Teknisesti tai hallinnollisesti epäyhtenäisessä ympäristössä yhtenäinen perintähierarkia olisi liian jäykkä ja usein teknisesti mahdotonkin ratkaisu. Eri järjestelmillä voi olla esimerkiksi samaa abstraktia rajapintatyyppiä edustavia rajapintoja, jotka ovat teknisesti erilaisia, mutta sovitettavissa yhteen erillisen muuntimen avulla. Tyyppivarastoon täytyy voida tallentaa tällaistakin yhteensopivuustietoa. Pilarcos-tyyppivarasto käyttää tähän kolmitasoista mallia, jossa abstraktit eli ideaalit tyypit, tyyppikuvaukset eli tekniset tyyppimäärittelyt ja kaavaimet eli ohjelmalliset toteutukset on erotettu [14, luku 6.2].

Pilarcos-meklauksen kannalta riittää, että tyyppivarastoon tallennetaan tieto siitä, mitkä rajapintakuvaukset ovat yhteensopivia suoraan tai muuntimen avulla. Meklari käyttää tätä tietoa palvelutarjouksissa annettujen rajapintojen yhteensopivuuden tarkistamiseen. Muuntimen käyttö ei tässä mallissa ole meklarin vaan palvelun tarjoajien ja käyttäjien vastuulla. Ominaisuuskehikkojen muuntaminen samalla peraatteella kuin rajapintojen olisi periaatteessa mahdollista, mutta sitä ei ole toistaiseksi toteutettu. Ominaisuuskehikkomäärittelyjen oletetaan siis olevan yhteisiä eri järjestelmille.

4.2 Palvelutarjous

Pilarcos-meklarin palvelutarjous on samankaltainen kuin CORBA-meklarin. Sen rakenne poikkeaa niiltä osin kuin Pilarcos-mallin palvelutyyppi (luku 3.3.3) poikkeaa CORBA-meklarin palvelutyypistä. Pilarcos-meklarin palvelutarjous koostuu seuraavista osista:

- palvelutyypin nimi,
- ominaisuusarvot kaikille palvelutyypin ominaisuuskehikoille ja
- tekninen rajapintatyyppi ja rajapintaviite kaikille palvelutyypin tarjotuille rajapinnoille.

Palvelutyypin kaikkien ominaisuuskehikoiden ominaisuuksille on palvelutarjouksessa lueteltava arvot tai käytettävä erityistä ”ei arvoa”-merkintää, jolloin ominaisuuden arvoksi kelpaa mikä tahansa. Myös dynaamisten ominaisuuksien käyttö on periaatteessa mahdollista.

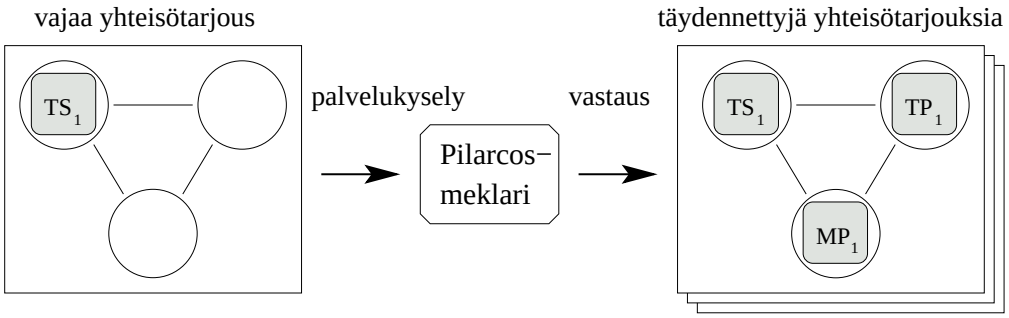
CORBA-meklarin palvelutarjoukseen sisältyy vain yksi olioviite, koska palvelu kuvataan yhtenä oliorajapintana; Pilarcos-meklarissa palvelulla voi sen sijaan olla useita rajapintoja, joiden kaikkien viitteet on annettava tajouksessa. Koska palvelutyyppi määrää vain kunkin rajapinnan abstraktin tyypin, tarjouksessa on lisäksi kerrottava rajapintojen tekniset tyypit. Tekniset tyyppikuvaukset kuvaukset säilytetään tyyppivarastossa, ja ne voivat olla esimerkiksi OMG IDL:llä määriteltyjä.

Palvelutarjous ilmoitetaan Pilarcos-meklarille rekisteröintirajapinnan kautta, samaan tapaan kuin CORBA-meklarillekin. Pilarcos-meklari tallentaa palvelutarjoukset tietokantaansa.

4.3 Palvelukysely

Tavanomaiselta meklauspalvelulta, kuten CORBA-meklarilta, haetaan tiettyä palvelutyyppiä olevia palvelutarjouksia erillistä rajoitelauseketta käyttäen. Pilarcos-meklari sen sijaan etsii keskenään yhteensopivat palvelutarjoukset kaikkiin annetun arkkitehtuurin rooleihin. Tällaista palvelutarjousyhdistelmää kutsutaan *yhteisötarjoukseksi*.

Erillistä rajoitelauseketta ei Pilarcos-meklauksessa käytetä lainkaan, vaan kysyjä antaa rajoitteensa sijoittamalla oman "palvelutarjouksensa" valmiiksi siihen rooliin, jossa itse toimii. Pilarcos-meklarille annetaan palvelukyselyssä vajaa yhteisötarjous, joka sisältää tavallisesti vain kysyjän palvelutarjouksen. Vastauksena kyselyyn meklari palauttaa joukon täydennettyjä yhteisötarjouksia, jotka sisältävät yhden palvelutarjouksen kutakin roolia kohti. Kuva 4.1 havainnollistaa yhteisötarjousten täydentämistä Pilarcos-meklarin avulla.



Kuva 4.1: Palvelukysely Pilarcos-meklauksessa.

Yhteisötarjous liittyy aina tiettyyn arkkitehtuuriin ja koostuu

- arkkitehtuurin nimestä sekä
- *(roolin nimi, palvelutarjous)*-pareista, joita on korkeintaan yksi kutakin arkkitehtuurin roolia kohden.

Yhteisötarjous on *vajaa*, jos se ei sisällä palvelutarjousta jollekin arkkitehtuurin roolille.

Pilarcos-meklauksessa kysyjä siis on symmetrisessä asemassa palvelujen tarjoajiin nähden. Kysyjän palvelutarjousta ei tosin tallenneta meklarin tietokantaan, vaan meklari pitää sitä muistissa ainoastaan palvelukyselyn suorittamisen ajan. Symmetrisyys on looginen seuraus siitä, että Pilarcos-malli yleistää kahdenväliset asiakas-palvelin-suhteet monenvälisiksi, jolloin palvelun tarjoaja voi asettaa toisille palveluille samanlaisia rajoitteita kuin palvelun kysyjäkin.

Yhteisötarjouksessa voi olla valmiina muitakin kuin kysyjän palvelutarjous. Jos joihinkin rooleihin halutut palvelutarjoukset ovat jo etukäteen tiedossa, ne voidaan täyttää ennalta yhteisötarjoukseen. Tällä ominaisuudella on useita käyttökohteita:

etukäteen sovittujen yhteistyöpalvelujen käyttäminen, kaatuneen palvelimen korvaaminen uudella muun yhteisön jatkaessa toimintaansa, ja yhteisön rakenteen muutosten kuten roolien lukumäärän muuttumisen hallinta. On myös mahdollista antaa palvelukyselyssä kokonaan tyhjä yhteisötarjous esimerkiksi silloin, kun meklaria käytetään hajautetun järjestelmän alustamiseen.

Pilarcos-meklauksessa ominaisuudet voivat olla yksittäisten arvojen lisäksi myös rajoitemuotoisia. Meklarin tehtävänä on palvelutarjouksia valitessaan tarkistaa, että toisiinsa sidoksien kautta liittyvien ominaisuuskehikoiden ominaisuusarvot ovat yhteensopivia. Yksittäiset arvot ovat yhteensopivia jos ne ovat samoja, ja rajoitteet ovat yhteensopivia jos niille on olemassa on yhteisiä ratkaisuja eli yhteisiä arvoja. Seuraavassa luvussa pohditaan tarkemmin, millaisia rajoitemuotoisten ominaisuuksien pitäisi olla.

Pilarcos-meklari laskee palvelutarjouksia vertaillessaan toisiinsa liittyvien ominaisuuksien leikkaukset, ja palauttaa laskemansa leikkaukset alkuperäisten ominaisuusarvojen tilalla vastauksessaan palvelukyselyyn. Yksittäisille arvoille tämä ei merkitse mitään, rajoitemuotoisille ominaisuuksille kyllä. Leikkausten laskeminen valmiiksi on järkevää, koska tapauksessa laskemaan leikkaukset vertailua tehdessään, ja koska yhteisöä muodostettaessa rajoitemuotoisten ominaisuuksien lopulliset arvot määräytyvät eri osapuolten vaatimusten leikkauksista.

4.4 Rajoitemuotoiset ominaisuudet

CORBA-meklarin palvelukyselyssä kysyjä antaa rajoitelausekkeen, jonka avulla sopivat palvelutarjoukset valitaan. Rajoitteet voivat olla yhtälö- tai epäyhtälömuotoisia, ja yksittäisiä rajoitteita voi yhdistellä loogisilla konnektiiveilla *ja*, *tai* ja *ei*, joten kokonaiset rajoitelausekkeet voivat olla hyvinkin monimutkaisia. Palvelutarjouksissa ominaisuuksilla sen sijaan voi olla vain yksittäisiä, kiinnitettyjä arvoja.

Pilarcos-meklarissa kysyjän antaman erillisen rajoitelausekkeen sijasta käytetään rajoitemuotoisia ominaisuuksia, joiden avulla sekä palvelujen tarjoajat että kysyjät voivat ilmaista vaatimuksensa. Yhteisötarjouksia muodostettaessa ominaisuuksia vertaillaan keskenään, jotta nähdään, ovatko ne yhteensopivia. Millaisia rajoitemuotoisten ominaisuuksia pitäisi olla? Useimmissa tilanteissa on tarpeen rajoittaa numeeristen arvojen (esimerkiksi jonon pituus) suuruutta tai jonkin muuttujan (esimerkiksi viestintäprotokollan tunnus) mahdollisia arvoja. On tärkeää, ettei rajoitteiden keskinäinen vertailu ole liian raskasta, koska meklarin tärkeimpiä piirteitä on suhteellisen lyhyt vastausaika palvelukyselyihin. Toisaalta rajoitteiden täytyy olla riittävän ilmaisuvoimaisia.

Numeerisille ominaisuuksille voitaisiin sallia vertailuoperaattoreihin ($=, \neq, <, >$) perustuvat rajoitteet. Tällöin esimerkiksi numeerisen ominaisuuden `jonon_pituus` arvot `jonon_pituus > 5` ja `jonon_pituus < 10` olisivat yhteensopivat, koska on olemassa muuttujan `jonon_pituus` arvoja, jotka toteuttavat molemmat rajoitteet. Rajoitteiden leikkaus olisi `5 > jonon_pituus < 10`. Arvovälien ilmaiseminen edellyttäisi vähintään kahden rajoitteen (ylä- ja alaraja) käyttöä, joten rajoitteita pitäisi

voida yhdistellä loogisin konnektiivein *ja*, *tai*. Jos tällaista yhdistelmärajoitetta ajatellaan yhden muuttujan yhtälöryhmänä, sillä voi olla useita ratkaisuja, joko yksittäisiä arvoja tai arvovälejä. Tällainen toteutus on mahdollinen, mutta tarpeettoman monimutkainen ottaen huomioon, että yleensä on tarpeen vain rajoittaa arvo yhdelle yhtenäiselle välille. Lisäksi suositummuuksien ilmaiseminen tulee mutkikkaaksi, jos mahdollisia arvovälejä on useita. Kevyt ratkaisu on sallia numeerisille ominaisuuksille yksinkertaiset arvovälit, kuten `[5, 10]`, joiden leikkausten laskeminen on helppoa. Tämä ratkaisu on käytössä Pilarcos-meklarin prototyyppitoteutuksessa.

Entä ominaisuudet, joilla on lueteltu joukko sallittuja arvoja? Esimerkiksi protokollan tunnus -ominaisuudella on toteutusympäristöstä riippuen muutama mahdollinen arvo. Mahdollisia arvoja voidaan yleisimmin ilmaista yhtäsuuruusoperaattorilla sekä *ja*-, *tai*- ja *ei*-konnektiiveilla. Esimerkkinä tästä on lauseke

```
protokollan_tunnus = IIOP_1.2 tai protokollan_tunnus = SOAP_1.1,
```

joka sallii `protokollan_tunnus`-muuttujalle kaksi mahdollista arvoa. Tällainen lauseke voidaan ilmaista Boolen algebralla, jos jokaista yhtäsuuruusvertailua pidetään omana totuusarvoisena muuttujanaan ja sallitaan ominaisuudella olevan useita mahdollisia arvoja. Erilliset lausekkeet ovat yhteensopivat, jos ne ovat yhtäaikaa voimassa joillakin muuttujien totuusarvoilla. Yhteensopivuuden tarkistaminen on tällöin ns. lausekalkyylin toteutuvuusongelma eli SAT-ongelma, joka on laskennalliselta vaativuudeltaan NP-täydellinen [25, luku 7.9]. Tämä tarkoittaa, että ongelmalle ei ole löydetty aikavaativuudeltaan syötteen pituuden suhteen polynomista ratkaisualgoritmia. Tämän tyyppiset rajoitelausekkeet näyttävät siis olevan liian monimutkaisia.

Useimmiten rajoitteilla halutaan ilmaista hyväksyttäviä vaihtoehtoja, harvemmin sulkea tiettyjä vaihtoehtoja erikseen pois. Rajoitteita voidaan yksinkertaistaa sallimalla ainoastaan hyväksyttävien vaihtoehtojen luettelemisen *tai*-konnektiiveilla eroteltuna. Näin menetellen yhteensopivuuden tarkistaminen palautuu vaihtoehtojoukkojen leikkauksen epätyhjyyden tarkistamiseen. Järjestämällä joukot leikkaus voidaan laskea joukkojen yhteenlasketun koon suhteen lineaarisessa ajassa. Kun tähän lisätään mahdollisuus ilmaista, saako lopullisessa leikkausjoukossa olla vain yksi vaiko useampia arvoja, päädytään meklarin prototyyppitoteutuksessa käytettyyn ratkaisuun, joka esitetään tarkemmin luvussa 5.3.1.

Arvovälit ja vaihtoehtojoukot eivät riitä ilmaisemaan ehtoja tilanteessa, jossa yksi rooli — useimmiten kysyjän — on epäsymmetrisessä asemassa muihin arkkitehtuurin rooleihin nähden. Esimerkiksi turistipalveluarkkitehtuurissa, jossa asiakas käyttää sekä turisti- että maksupalvelua, asiakas voi määrätä ylärajan turistipalvelun ja maksupalvelun hinnoille erikseen, mutta ei niiden summalle. Palvelukyselyyn voidaan näitä tilanteita varten ottaa mukaan CORBA-meklarin tapaan vapaamuotoinen rajoitelauseke, jolla kysyjä voi asettaa lisäehtoja tarjousyhdistelmille. Turistipalveluarkkitehtuurissa (luku 3.4) kysyjä voisi määritellä yhteishinnan ylärajan viideksi rahayksiköksi seuraavalla lausekkeella:

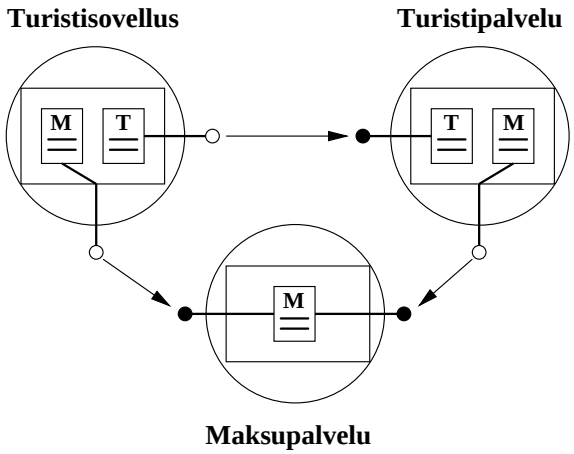
```
palvelin.turistipalveluominaisuudet.hinta +
maksupalvelu.veloitusominaisuudet.hinta < 5.0
```

Erillisen rajoitelausekkeen käyttäminen rikkoo kysyjän ja tarjoajien symmetrian, mutta lisää joissain tapauksissa Pilarcos-meklauksen tarkkuutta ja siten myös käyttökelpoisuutta.

4.5 Ominaisuuskehikkoverkot

Palvelukyselyn saadessaan Pilarcos-meklari etsii rajapinnoiltaan ja ominaisuuksiltaan yhteensopivia palvelutarjouksia annetun yhteisötarjouksen tyhjiin rooleihin. Tarjousten yhteensopivuusvaatimukset määräytyvät arkkitehtuurikuvauksen sidosten perusteella (luku 3.4.3). Kukin sidos yhdistää kaksi roolia, joista toinen käyttää ja toinen tarjoaa sidottua rajapintaa. Sidos yhdistää myös rajapintaan liittyvät ominaisuuskehikot. Sidottuihin rooleihin sijoitettavat palvelutarjoukset ovat yhteensopivia vain, jos sekä sidotut rajapinnat että niihin liittyvien ominaisuuskehikoiden arvot ovat yhteensopivia.

Kaikkiin sidottuihin rooleihin ehdolla olevia palvelutarjouksia on siis vertailtava keskenään. Arkkitehtuurikuvausten rakenteessa on piirre, joka mutkistaa vertailua: palvelutyyppiä määriteltäessä yhden ominaisuuskehikon voi liittää useaan rajapintaan, jolloin näiden rajapintojen välille muodostuu yhteys (luku 3.3.3). Rooliin sijoitettuna tällainen palvelutyyppi yhdistää siihen sidottujen roolien ominaisuuskehikot. Kuva 4.2 esittää turistipalveluarkkitehtuuriin muodostuvaa ominaisuuskehikkojen ketjua. Maksupalvelu-palvelutyyppiin sisältyvän ”maksuominaisuudet”-kehikon kautta liittyvät toisiinsa välillisesti myös Turistipalvelun ja Turistiasiakkaan vastaavat kehikot, jotka on kuvassa merkitty **M**-kirjaimella. Palvelutarjousten yhteensopivuus edellyttää tässä tapauksessa kaikkien kolmen ominaisuuskehikon arvojen yhteensopivuutta.



Kuva 4.2: Ominaisuuskehikkojen ketjuuntuminen turistipalveluarkkitehtuurissa.

Edellä olevassa esimerkissä ominaisuuskehikoista muodostuu ketju. Yleisessä tapauksessa ominaisuuskehikkojen liitoksista muodostuu yhtenäinen, yksinkertainen

ja suuntaamaton verkko, jossa ominaisuuskehikot ovat solmuja ja niiden väliset liitokset kaaria. Tällaisia verkkoja nimitetään *ominaisuuskehikkoverkoiksi*. Esimerkissä toinen ominaisuuskehikkoverkko muodostuu Turistisovelluksen ja Turistipalvelun **T**-kirjaimella merkittyjen kehikkojen välille. Kutakin ominaisuuskehikkotyyppiä kohti muodostuu oma verkko; samaa kehikkotyyppiä kohti voi muodostua useampikin verkko, jos kehikkotyyppi on arkkitehtuurissa käytössä useissa erillisissä roolisaarekkeissa.

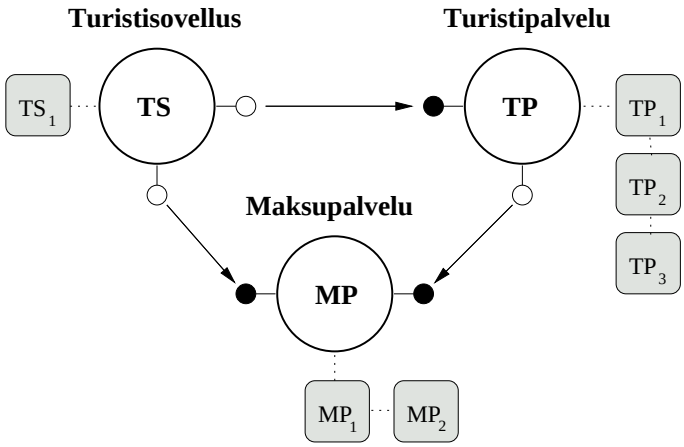
Palvelutarjouksia valitessaan meklarin tehtävä on varmistaa, että kunkin verkon sisällä ominaisuusarvot ovat keskenään yhteensopivia, ja lisäksi laskea arvojen leikkaukset palautettaviksi alkuperäisten paikoilla. Ominaisuuskehikkoverkon voidaan ajatella edustavan yhtä yhteistä ominaisuuskehikkoa, jonka arvoista on yhteisön kesken päästävä sopuun. Palvelutarjouksia ei näin ollen pidä meklarissa vertailla sidosten mukaisesti pareittain, vaan ominaisuuskehikkoverkoittain.

Ominaisuuskehikkoverkot muodostuvat, kun palvelutyyppejä sijoitetaan arkkitehtuurikuvauksen rooleihin ja roolien välille tehdään sidoksia. Koska verkot eivät riipu palvelutarjouksista millään tavoin, ne voidaan etsiä etukäteen. Työkalu, jolla arkkitehtuurikuvauksia laaditaan, voisi etsiä muodostuvat verkot ja näyttää ne käyttäjälle, mikä helpottaisi arkkitehtuurin suunnittelua. Työkalu voisi tallentaa verkot arkkitehtuurikuvauksen yhteydessä tyyppivarastoon. Tällaista työkalua ei ole kuitenkaan vielä olemassa, joten prototyyppitoteutuksessa tyyppivarasto etsii ja tallentaa verkot muistiinsa arkkitehtuurikuvausta rekisteröitäessä. Kuhunkin rooliin liitetään tieto, mihin verkkoihin se kuuluu. Meklari saa verkkotiedot tyyppivarastolta valmiina, mikä nopeuttaa palvelukyselyyn vastaamista.

4.6 Tarjousyhdistelmien etsintä

Kun Pilarcos-meklari vastaanottaa palvelukyselyn, sen tehtävänä on muodostaa kokonaisia yhteisötarjouksia erillisistä palvelutarjouksista. Kysyjälle on yleensä tärkeämpää saada vastauksena vähintään yksi yhteisötarjous suhteellisen nopeasti kuin saada kaikki mahdolliset yhteisötarjoukset viiveellä, koska yksikin tarjous riittää yhteisön muodostamiseen. Mahdollisia yhteisötarjouksia saattaa myös olla liian paljon: erilaisten palvelutarjousyhdistelmien määrä saadaan kertomalla eri rooleihin ehdolla olevien tarjousten määrät keskenään. Näistä syistä palvelukyselyn yhteydessä pitää voida asettaa yläraja palautettavien yhteisötarjousten määrälle, palvelukyselyn kestolle tai mieluiten molemmille.

Meklarin tarjousyhdistelmien etsinnän täytyy edetä syvyyssuuntaisesti arkkitehtuurin roolien suhteen, jotta kokonaisia yhdistelmiä löydettäisiin mahdollisimman nopeasti. Kuhunkin arkkitehtuurin rooliin liittyy jokin palvelutyyppi, ja yleensä eri rooleilla on eri palvelutyypit. Oletettavasti meklari tallentaa palvelutarjoukset palvelutyypeittäin, joten kuhunkin rooliin on ehdolla kaikki palvelutyypin mukaiset palvelutarjoukset. Esitäytettyjä rooleja – niitä, joihin on jo palvelukyselyssä kiinnitetty palvelutarjous – käsitellään kuten muitakin, mutta niihin on ainoastaan yksi ehdokas.



Kuva 4.3: Palvelutarjoukset turistipalveluarkkitehtuurin rooleille.

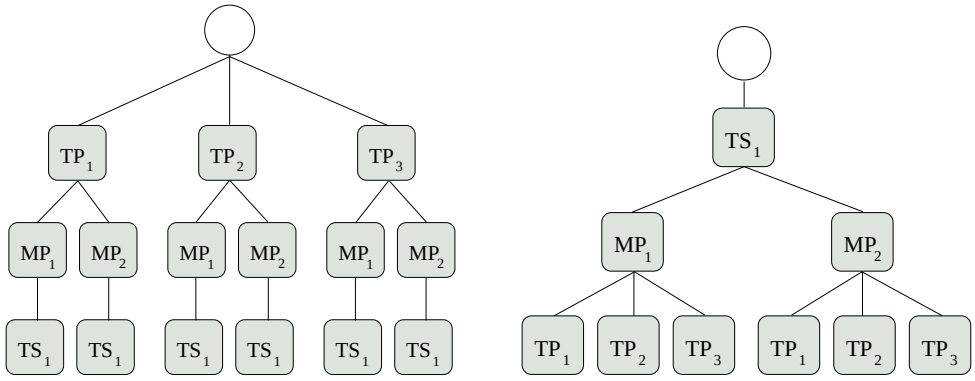
Esimerkkitapauksena tarkastellaan turistipalveluarkkitehtuuriin kohdistuvaa palvelukyselyä, jossa turistisovelluksen rooli on valmiiksi täytetty tarjouksella (TS_1). Meklarilla on kolme Turistipalvelu-tyyppistä palvelutarjousta (TP_1, TP_2, TP_3) ja kaksi Maksupalvelu-tyyppistä tarjousta (MP_1, MP_2). Tilanne on esitetty kaaviona kuvassa 4.3. Oletetaan, että meklari alkaa kokeilla palvelutarjousyhdistelmiä rooli kerrallaan järjestyksessä Turistipalvelu, Maksupalvelu, Turistisovellus (järjestyksellä ei ole lopputuloksen kannalta merkitystä). Tällöin palvelutarjouksista muodostuu etsintäpuu, joka on esitetty kuvassa 4.4(a). Meklari käy puun läpi esijärjestyksessä tarkistaen kunkin solmun kohdalla palvelutarjouksen rajapintojen ja ominaisuuksien yhteensopivuuden edellisten tarjousten kanssa. Jos tarjous ei ole yhteensopiva, palataan takaisin edelliseen solmuun. Jos päästään puun lehteen asti, polulla juuresta lehteen olevat palvelutarjoukset muodostavat kokonaisen yhteisötarjouksen, joka voidaan palauttaa kysyjälle.

Jos kysyjä voi palvelukyselyssä antaa myös erillisen rajoitelausekkeen (luku 4.4), meklari voi laskea sen totuusarvon lopuksi kokonaiselle yhteisötarjoukselle. Jos rajoitelauseke ei toteudu, meklari hylkää yhteisötarjouksen. Erillisen rajoitelausekkeen käyttö ei ole Pilarcos-meklauksessa keskeistä eikä välttämätöntä, joten sitä ei käsitellä tarkemmin.

Palvelutarjouksen rajapintojen yhteensopivuuden meklari selvittää tyyppivaraston (luku 4.1) tarjoaman operaation avulla. Kutakin rajapintaa verrataan siihen sidoksissa olevaan rooliin valitun tarjouksen vastaavaan rajapintaan.

Ominaisuuksia vertaillaan ominaisuuskehikkoverkkojen avulla. Etsinnän alussa meklari pyytää tyyppivarastolta arkkitehtuurikuvauksen ja siihen liittyvät ominaisuuskehikkoverkkotiedot. Kutakin verkkoa kohden luodaan yksi ominaisuuskehikko. Kuva 4.5 esittää verkkokohtaisia ominaisuuskehikoita esimerkkitapauksessamme.

Verkkotiedoista ilmenee, mihin verkkoihin kukin rooli kuuluu. Kun meklari tulee etsintäpuussa uuteen solmuun eli kokeilee uutta palvelutarjousta, se laskee samalla tarjouksen ominaisuuskehikoiden ja vastaavien verkkokohtaisten ominaisuuskehikoiden arvojen leikkaukset. Jos leikkaukset ovat epätyhjiä, palvelutarjous on yhteen-

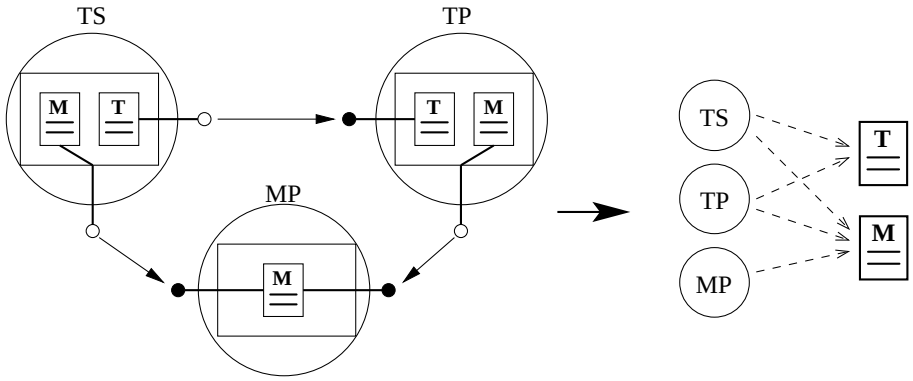


(a) Ei-optimoitu etsintäpuu. (b) Optimoitu etsintäpuu.

Kuva 4.4: Esimerkkejä meklarin etsintäpuista.

sopiva, ja lasketut leikkaukset tallennetaan verkkokohtaisiin ominaisuuskehikoihin. Nämä siis sisältävät kullakin hetkellä leikkaukset kaikista etsintäpolun varrella olevien palvelutarjousten ominaisuusarvoista.

Verkkokohtaisten ominaisuuskehikkojen arvot on tallennettava pinoon, jotta voidaan palata edellisiin arvoihin kun etsinnässä joudutaan peruuttamaan. Tämä voidaan saada helposti aikaan käyttämällä rekursiivista algoritmia. Jos etsinnässä päästään puun lehteen asti, ominaisuusarvojen leikkaukset on laskettu valmiiksi, ja ne kopioidaan alkuperäisten arvojen tilalle ennen valmiin yhteisötarjouksen palauttamista kysyjälle. Koska leikkausten laskeminen tapahtuu tarjousten vertailun yhteydessä, se ei pidennä meklarin vastausaikaa.



Kuva 4.5: Verkkokohtaisten ominaisuuskehikkojen muodostaminen.

Esitetty algoritmi tekee minimimäärän ominaisuusarvovertailuja, koska vertailut tehdään ominaisuuskehikkoverkoittain. Palvelutarjousten läpikäyntiä sen sijaan on mahdollista tehostaa etenkin tapauksissa, joissa yhteensopimattomia yhdistelmiä esiintyy runsaasti. Kuvan 4.4(a) etsintäpuussa on 15 solmua, jotka on pahimmassa

tapauksessa kaikki käytävä läpi, ennen kuin voidaan todeta, ettei sopivia yhdistelmiä löytynyt. Selvästi olisi eduksi, jos etsintäpuun haaran läpikäynti lopetettaisiin mahdollisimman aikaisin yhteensopimattomuuden ilmetessä ja jos tällä tavoin voitaisiin ohittaa mahdollisimman monta kelpaamatonta palvelutarjousta.

Tilannetta voidaan parantaa yksinkertaisella menetelmällä: järjestämällä roolit niihin ehdolla olevien palvelutarjousten määrän mukaisesti nousevaan järjestykseen. Tällä menetelmällä esimerkkitapauksen etsintäpuu saadaan kuvan 4.4(b) mukaiseksi, jolloin siinä on enää yhdeksän solmua. Menetelmä edellyttää, että palvelutarjousten määrä roolia kohti on etukäteen tiedossa; näin yleensä on, jos meklari tallentaa palvelutarjoukset palvelutyypeittäin. Tätä tarkempi optimointi edellyttäisi palvelutyyppien sisällön huomioon ottamista ja olisi selvästi vaikeampi toteuttaa, eikä välttämättä johtaisi merkittävästi parempaan lopputulokseen.

Edellä kuvatun algoritmin päävaiheet on esitetty pseudokoodina kuvassa 4.6. Esitetyssä muodossa algoritmi hyödyntää rekursiota ja kerää kaikki mahdolliset palvelutarjousyhdistelmät.

Missä järjestyksessä löytyneet yhteisötarjoukset pitäisi palauttaa kysyjälle? Edellä esitetty etsintäalgoritmi palauttaa yhteisötarjoukset syvyyssuuntaisen läpikäynnin mukaisessa järjestyksessä, varioiden ensin niitä rooleja, joihin on eniten palvelutarjouksia. Kysyjä tuskin haluaa palvelukyselyn vastauksena listaa, jossa kukin yhteisötarjous on lähes identtinen edellisen kanssa poiketen siitä vain yhdellä palvelutarjouksella.

Liiallisesta yhteisötarjousten samankaltaisuudesta päästään eroon järjestämällä rooliin ehdolla olevat palvelutarjoukset satunnaiseen järjestykseen aina ennen etsinnän aloittamista ja aloittamalla etsintä alusta ensimmäisestä roolista, kun kokonainen yhteisötarjous on löydetty. Koska palvelutarjousyhdistelmien määrä on yleensä suuri, näin voidaan saada aikaan toisistaan selvästi poikkeavia yhteisötarjouksia.

Toisaalta järjestykseltään satunnaiset yhteisötarjoukset eivät ole kysyjän kannalta parhaita mahdollisia vastauksia, vaan kysyjä yleensä haluaa vastaukset paremmuusjärjestyksessä. CORBA-meklarissa järjestyksen voi määrätä suositummuuslausekkeella, jonka mukaisesti palvelutarjoukset järjestetään ennen palauttamista (luku 2.3.4). Pilarcos-meklarissa yhteisötarjoukset koostuvat useista palvelutarjouksista, joten kysymys on monimutkaisempi. Pilarcos-meklarissakin olisi periaatteessa mahdollista käyttää suositummuuslauseketta, jonka mukaan kokonaiset yhteisötarjoukset järjestetään ennen palauttamista. Tilanne on kuitenkin erilainen kuin CORBA-meklarissa: Pilarcos-meklarilla ei ole yhteisötarjouksia valmiina, vaan ne täytyy muodostaa ensin, mikä on laskennallisesti raskasta.

Eräs ratkaisu näihin edellä mainittuihin ongelmiin olisi järjestää kunkin roolin palvelutarjoukset suositummuuskriteerien mukaiseen järjestykseen etsinnän aluksi. Suositummuudeltaan samanarvoiset palvelutarjoukset kullekin roolille voitaisiin edelleen järjestää keskenään satunnaiseen järjestykseen. Suositummuuskriteerit voisivat olla kysyjän erikseen antamia tai sisältyä ominaisuuksien arvoihin. Tällä tavoin Pilarcos-meklari löytäisi nopeasti kysyjän kannalta hyviä ja riittävästi toisistaan poikkeavia yhteisötarjouksia. Suositummuuskriteerien lisääminen kuuluu jatkokehityskohteisiin.

Algoritmi **EtsiYhteisötarjoukset**(vajaa yhteisötarjous V):

1. Hae tyyppivarastolta V:n arkkitehtuurikuvaus ja ominaisuuskehikkoverkot.
2. Kerää mahdolliset palvelutarjoukset kullekin arkkitehtuurikuvauksen roolille R_i seuraavasti:
   (a) Jos V sisältää esitäytetyn palvelutarjouksen R_i:lle, merkitse se R_i:n ainoaksi palvelutarjoukseksi; muuten
   (b) etsi palvelutarjoustietokannasta R_i:n palvelutyyppiä olevat tarjoukset.
3. Järjestä roolit ($R_1 \ldots R_n$) palvelutarjousten määrän mukaisesti nousevaan järjestykseen.
4. Alusta valmiiden yhteisötarjousten lista L tyhjäksi.
5. Kutsu **EtsiRooliin**(R_1, tyhjä yhteisötarjous, tyhjät verkkokohtaiset ominaisuuskehikot).
6. Palauta yhteisötarjousten lista L.

Algoritmi **EtsiRooliin**(rooli R_i, yhteisötarjous Y, verkkokohtaiset ominaisuuskehikot O):

1. Kaikille R_i:n palvelutarjouksille P_i:
   (a) Tarkista tyyppivarastolta P_i:n rajapintojen yhteensopivuus Y:ssä olevien naapuriroolien palvelutarjousten rajapintojen kanssa.
   (b) Jos jokin rajapinta ei ollut yhteensopiva, palaa algoritmista.
   (c) Laske P_i:n ja O:n vastaavien ominaisuusarvojen leikkaukset ja sijoita ne O:hon vanhojen arvojen päälle.
   (d) Jos jokin leikkaus oli tyhjä, palaa algoritmista.
   (e) Lisää palvelutarjous P_i yhteisötarjoukseen Y.
   (f) Jos R_i on viimeinen rooli, niin:
      i. Kopioi O:n arvot vastaaviin ominaisuuskehikoihin kaikkiin Y:n palvelutarjouksiin.
      ii. Lisää Y valmiiden yhteisötarjousten listaan L.

   muuten:
      i. Kutsu **EtsiRooliin**(rooli R_{i+1}, kopio Y:stä, kopio O:sta).
2. Palaa algoritmista.

Kuva 4.6: Yhteisötarjousten etsintäalgoritmin päävaiheet.

4.7 Etsinnän laskennallinen vaativuus

Palvelutarjousyhdistelmien etsintä on luonteeltaan kombinatorinen ongelma ja siten laskennallisesti hyvin vaativa. Arkkitehtuurikuvauksen mielivaltaisuus lisää osaltaan sekin vaativuutta. Päätösongelma "Löytyykö annetusta mielivaltaisesta arkkitehtuurista ja mielivaltaisesta palvelutarjousjoukosta vähintään yksi täydellinen yhteisötarjous?" on NP-täydellinen. Tämä voidaan osoittaa toteamalla ongelman olevan luokassa NP ja laatimalla aikavaativuudeltaan polynominen palautus klikkiongelmasta (CLIQUE) [25, luku 7.7] meklausongelmaan.

Palautusta klikkiongelmasta ei esitetä tässä formaalisti, mutta se voidaan tehdä seuraavaan tapaan [33]. Löydettävää k:n solmun klikkiä vastaamaan laaditaan arkkitehtuurikuvaus, jossa on k roolia, ja jossa kaikkien roolien välillä on kaksi sidosta, yksi molempiin suuntiin. Yksinkertaisinta on käyttää kaikille rooleille samaa palvelutyyppiä, jossa on $k-1$ tarjottua ja $k-1$ käytettyä rajapintaa. Tarjottuihin rajapintoihin liittyy yksi jaettu ominaisuuskehikko ("sisään"-kehikko). Jokaiseen käytettyyn rajapintaan liittyy erillinen ominaisuuskehikko ("ulos"-kehikot). Kaikki ominaisuuskehikot sisältävät yhden vaihtoehtojoukkotyyppisen ominaisuuden (luku 4.4).

Kutakin verkon solmua vastaamaan laaditaan palvelutarjous, jossa "ulos"-kehikon ainoaksi mahdolliseksi arvoksi asetetaan solmun tunnus. "Sisään"-kehikkojen ominaisuuksien mahdollisiksi arvoiksi asetetaan naapurisolmujen tunnukset. Kyseessä on siis verkon vieruslistaesitys. Näin k-klikin olemassaolo-ongelma on saatu palautettua ongelmaan, löytyykö klikin määrittelevälle arkkitehtuurikuvaukselle verkon solmuja kuvaavista palvelutarjouksista koostuvaa yhteisötarjousta.

Meklausongelman NP-täydellisyys merkitsee, että sille ei tunneta syötteen koon suhteen polynomisessa ajassa toimivia ratkaisualgoritmeja. Ongelma on siis yleisessä tapauksessa hyvin vaativa. Käytännön tapauksissa arkkitehtuurikuvausten koko lienee onneksi pieni, alle kymmenen roolin luokkaa. Koska arkkitehtuurikuvaus on myös ennalta tunnettu, voidaan ajatella, että se ei ole osa syötettä. Tällöin kokeiltavien yhdistelmien enimmäismäärä saadaan tuloperiaatteen mukaisesti kertomalla keskenään kuhunkin rooliin tarjolla olevien palvelutarjousten määrä. Näillä oletuksilla meklausongelma on aikavaativuudeltaan vain polynominen palvelutarjousten määrän suhteen; tarkemmin sanoen ongelma on luokassa $O(n^r)$, missä n on palvelutarjousten enimmäismäärä roolia kohti ja r on roolien lukumäärä.

Yhteisötarjousten etsintä on rajoitetussakin tapauksissa laskennallisesti raskas prosessi, joka ei ole riittävän tehokas, jos sopivia tarjousyhdistelmiä on hyvin vähän verrattuna yhdistelmien kokonaismäärään. Käytännössä kyselyn enimmäiskestoa täytyy pystyä rajoittamaan, jotta kysyjä välttyy yllättäviltä viiveiltä. Samoin palautettavien yhteisötarjousten määrälle täytyy pystyä asettamaan yläraja, koska useimmiten yksi tai muutama yhteisötarjous riittää kysyjälle. Näiden parametrien käytännön toteutusta ja vaikutusta käsitellään seuraavassa luvussa.

Luku 5

Pilarcos-meklarin prototyyppitoteutus

Pilarcos-projektissa on kehitetty CORBA-väliohjelmistoalustalla toimiva prototyyppi Pilarcos-meklauspalvelusta ja sitä tukevasta tyyppivarastosta. Prototyyppitoteutus konkretisoi edellisissä luvuissa esitetyt periaatteet ja antaa keinot Pilarcos-meklauksen skaalautuvuuden arviointiin suorituskykymittausten avulla. Tässä luvussa esitellään prototyyppitoteutuksen rakenne, valitut tekniset ratkaisut sekä suorituskykymittaukset ja niiden tulokset.

5.1 Toteutustekniikka

Pilarcos-meklarista on tehty kaksi prototyyppitoteutusta. Ensimmäisen laati kirjoittaja itse osaksi Pilarcos-prototyypin ensimmäistä versiota [32]. Toteutus oli kirjoitettu Java-kielellä ja se toimi OpenCCM-alustalla [24]. Vain osa toiminnallisuudesta oli toteutettu, ja meklarin käyttämät arkkitehtuurikuvaukset olivat yksinkertaisempia kuin tässä tutkielmassa käsitellyt.

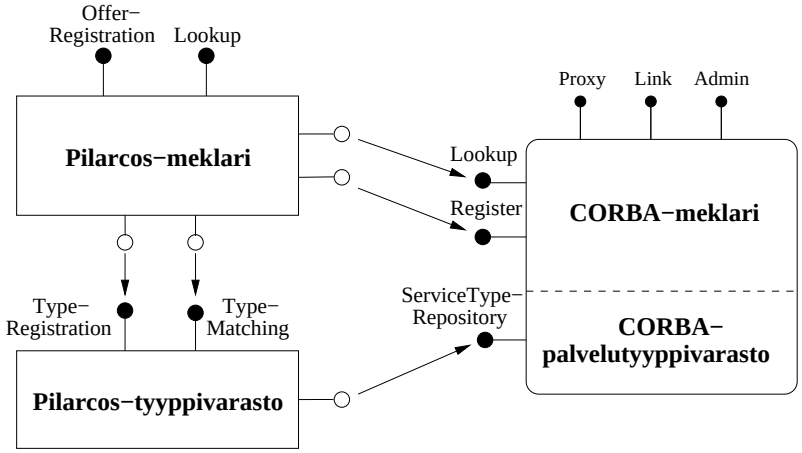
Uudempi Pilarcos-meklarin prototyyppitoteutus sai alkunsa PIMEA-ohjelmistotuotantoprojektissa [26], jossa kirjoittaja ohjasi työtä Pilarcos-projektin edustajana. Toteutus viimeisteltiin myöhemmin Pilarcos-projektissa, ja se yhdistettiin osaksi laajempaa Pilarcos II-prototyyppiä [31]. Tässä luvussa käsitellään ainoastaan uutta prototyyppitoteutusta, joka on toiminnoiltaan täydellisempi ja myös tehokkaampi kuin edeltäjänsä.

Uusi prototyyppitoteutus on kirjoitettu C++-kielellä MicoCCM-alustalle [18]. MicoCCM on avoimeen lähdekoodiin perustuva CORBA-komponenttimallin C++-kielinen toteutus. CORBA-komponenttimalli (*CORBA Component Model, CCM*) on OMG-standardoinnin loppuvaiheessa oleva CORBA-mallin laajennos, joka tuo CORBA-standardiin komponenttiperustaisen palvelinten ohjelmointimallin [23]. Pilarcos-meklariprototyypin rakenteen ymmärtämiseksi riittää tieto, että CORBA-komponentin käyttämät ja tarjoamat rajapinnat voidaan kuvailla eksplisiittisesti sen IDL-määrittelyn avulla, ja että komponentteja voidaan helposti kytkeä toisiinsa

kokoonpanoiksi.

5.2 Prototyypin rakenne

Prototyyppitoteutus sisältää Pilarcos-meklarin, sitä tukevan tyyppivaraston ja erillisen testausohjelman. Pilarcos-meklarilla on kaksi mahdollista toimintatilaa: itsenäinen ja CORBA-meklaria palvelutarjoustietokantana käyttävä tila. Itsenäinen tila on suorituskyvyn kannalta tehokkaampi, koska palvelutarjoukset säilytetään Pilarcos-meklarin omassa tietorakenteessa. CORBA-meklaria käyttämällä puolestaan on mahdollista hyödyntää CORBA-meklarin erityispiirteitä kuten dynaamisia ominaisuuksia ja meklarien federointia (luku 2). CORBA-meklaritoteutuksena kokeiltiin sekä Mico-alustan omaa CORBA-meklaria että Java-pohjaista ORBacus-meklaria [9]. Mico-meklari on kokonaan keskusmuistipohjainen, mutta ORBacus-meklari pystyy säilyttämään palvelutarjoukset myös levymuistissa niin haluttaessa.



Kuva 5.1: Pilarcos-meklarin prototyyppitoteutuksen rakenneosat ja rajapinnat.

Pilarcos-meklariprototyypin rakenne on esitetty kuvassa 5.1. Pilarcos-meklari ja -tyyppivarasto on molemmat toteutettu CORBA-komponentteina. Kuten kuvasta ilmenee, Pilarcos-meklari käyttää CORBA-meklauspalvelua ja tyyppivarastoa. Tyyppivarasto puolestaan käyttää CORBA-palvelutyyppivarastoa. Ainoastaan Pilarcos-meklarin ja -tyyppivaraston rajapinnat ovat näkyvissä kysyjille ja tarjoajille; CORBA-meklauspalvelun rajapinnat ovat piilossa Pilarcos-meklarin käyttäjiltä. Kuvassa esitettyjen osien lisäksi prototyyppiin kuuluu testausohjelma, joka lukee tyyppimäärittelyjä, palvelutarjouksia sekä palvelukyselyitä tiedostosta ja suorittaa niitä tallentaen tulokset lokitiedostoon.

Tyyppivarasto on toteutukseltaan yksinkertainen. Varastoon voi rekisteröidä ja sieltä voi hakea palvelutyyppi- ja arkkitehtuurikuvauksia sekä muita luvussa 3 esiteltyjä tyyppikuvauksia. Tyyppivarasto myös muuntaa palvelutyyppikuvaukset CORBA-meklarin ymmärtämään muotoon CORBA-palvelutyyppivarastoon. Muunnos kuvaillaan luvussa 5.4.1.

Arkkitehtuurikuvausta rekisteröitäessä tyyppivarasto etsii ja tallentaa muistiinsa arkkitehtuurin ominaisuuskehikkoverkot (luku 4.5), joita meklari käyttää yhteisötarjousten etsinnässä. Ominaisuuskehikkoverkkojen etsintäalgoritmi perustuu arkkitehtuurin roolien ja sidosten muodostaman verkon syvyyssuuntaiseen läpikäyntiin. Tyyppivarasto tarjoaa myös operaation, jonka avulla meklari voi tarkistaa kahden teknisen rajapinnan yhteensopivuuden. Tyyppivarasto ei tue tyyppien perintää. Organisaatioiden välisessä ympäristössä, jota varten se on suunniteltu, ei yhtenäinen perintähierarkia yleensä ole mahdollinenkaan (luku 4.1).

CORBA-meklaria hyödyntävässä toimintatilassa Pilarcos-meklari muuntaa ja tallentaa palvelutarjoukset CORBA-meklarille rekisteröinnin yhteydessä. Palvelukyselyn saapuessa Pilarcos-meklari hakee mahdolliset palvelutarjoukset tyhjiin rooleihin CORBA-meklarilta suorittaen muunnoksen toiseen suuntaan, ja alkaa vasta tämän jälkeen etsiä kokonaisia yhteisötarjouksia. Prototyyppi ei vielä tue CORBA-meklarin dynaamisia ominaisuuksia, mutta ne on mahdollista ottaa käyttöön vähäisellä lisätyöllä.

Luvussa 4 kuvailtiin Pilarcos-meklari keskitettynä palveluna. CORBA-meklarin käytön ansiosta Pilarcos-meklarit on periaatteessa mahdollista federoida keskenään tai muiden CORBA-meklarien kanssa. Prototyypin rakenne mahdollistaa myös Pilarcos-meklarikomponentin asentamisen paikalliselle palvelimelle erilleen CORBA-meklarista, jolloin Pilarcos-meklarin laskennallisesti raskaampia toimintoja käytetään vain paikallisesta verkosta. Tällainen ratkaisu tosin lisää verkkoliikennettä, ja palvelutarjousten siirto kuormittaa sekin osaltaan Pilarcos-meklaria. Erilaisiin hajautusratkaisuihin palataan myöhemmin suorituskykymittausten yhteydessä.

5.3 Tietorakenteet ja rajapinnat

Seuraavissa luvuissa käsitellään Pilarcos-meklariprototyypin julkisia rajapintoja ja niihin liittyviä tietorakenteita. Tietorakenteet ja rajapinnat on määritelty OMG IDL-kielellä pyrkien selkeyteen ja luettavuuteen. Liite C sisältää keskeisimmät IDL-kuvaukset tiiviissä muodossa.

5.3.1 Ominaisuustyypit ja -kehikot

Ominaisuustyypit on toteutettu samaan tapaan kuin CORBA-meklarissa (luku 2.3.2) siten, että ominaisuuden tyyppi voi olla mikä tahansa IDL-perustyyppi. Tämän lisäksi kolmella tietueella — `LongRange`, `DoubleRange` ja `StringSet` — on ominaisuustyyppinä meklarille erityismerkitys: niillä ilmaistaan luvussa 4.4 käsiteltyjä rajoitemuotoisia ominaisuuksia.

`LongRange`- ja `DoubleRange`-tietueita käytetään kokonais- ja liukulukuarvovälien ilmaisemiseen. Ensimmäinen on suljettuja kokonaislukuvälejä, jälkimmäinen suljettuja liukulukuvälejä varten. Kahden arvovälityyppisen ominaisuuden leikkaus saadaan arvovälien leikkauksena. Arvovälityyppiset ominaisuudet ovat yhteensopivia vain, jos arvovälien leikkaus on epätyhjä.

`StringSet`-tietue ilmaistaan ominaisuuksia, joilla on lueteltu joukko mahdollisia merkkijonomuotoisia arvoja. `StringSet`-tietue koostuu joukosta merkkijonoja ja rajoitetyypistä. Rajoitetyyppi määrää, millä tavoin ominaisuuden lopullinen arvo (tai arvot) valitaan mahdollisten arvojen joukosta. Rajoitetyypit ja niiden merkitykset ovat:

- `some_of` määrää, että lopulliseksi arvoksi voidaan valita yksi tai useampi merkkijonojoukon arvo;
- `one_of` määrää, että lopulliseksi arvoksi on valittava täsmälleen yksi merkkijonojoukon arvo; ja
- `exactly` määrää, että lopulliseksi arvoksi kelpaa ainoastaan koko merkkijonojoukko alkuperäisessä muodossaan.

Yhteensopivuus- ja leikkausrelaatioiden esittämistä varten käsitellään `StringSet`-ominaisuutta parina (c, S), jossa c on rajoitetyyppi ja S on merkkijonojoukko. Yhteensopivuusehdot eri rajoitetyypeille on esitetty taulukossa 5.1. Yhteensopivien `StringSet`-ominaisuuksien leikkauksenmuodostamissäännöt ovat taulukossa 5.2. Esimerkiksi rajoitteet `some_of` ("A", "B", "C") ja `one_of` ("B", "C", "D") ovat yhteensopivat, ja niiden leikkaukseksi muodostuu `one_of` ("B", "C"). On huomattava, että `StringSet`-ominaisuustyypeissä voitaisiin merkkijonojen sijasta käyttää minkä tahansa tyyppisiä arvoja samoja sääntöjä käyttäen; merkkijonot valittiin toteutuksen yksinkertaistamiseksi.

Taulukko 5.1: Yhteensopivuusehdot `StringSet`-ominaisuuksille (c_i, S_i) ja (c_j, S_j).

	$c_i =$ `some_of`	$c_i =$ `one_of`	$c_i =$ `exactly`
$c_j =$ `some_of`	$S_i \cap S_j \neq \emptyset$	$S_i \cap S_j \neq \emptyset$	$S_i \subset S_j$
$c_j =$ `one_of`		$S_i \cap S_j \neq \emptyset$	$S_i \subset S_j \wedge \lvert S_i \rvert = 1$
$c_j =$ `exactly`			$S_i = S_j$

Taulukko 5.2: Yhteensopivien `StringSet`-ominaisuuksien (c_i, S_i) ja (c_j, S_j) leikkaus.

	$c_i =$ `some_of`	$c_i =$ `one_of`	$c_i =$ `exactly`
$c_j =$ `some_of`	(`some_of`, $S_i \cap S_j$)	(`one_of`, $S_i \cap S_j$)	(`exactly`, S_i)
$c_j =$ `one_of`		(`one_of`, $S_i \cap S_j$)	(`exactly`, S_i)
$c_j =$ `exactly`			(`exactly`, S_i)

Sekä arvoväli- että `StringSet`-tyyppisten ominaisuuksien leikkausten laskeminen on vaihdannainen ja liitännäinen operaatio, joten meklari voi laskea leikkaukset missä tahansa järjestyksessä. Tämä on olennaista tarjousten etsintäalgoritmin (luku 4.6) toiminnan kannalta. Monimutkaisempia ehtoja voidaan tarvittaessa ilmaista käyttämällä usean ominaisuuden yhdistelmiä.

Ominaisuuskehikot on toteutettu nimettyinä ominaisuustyyppilistoina, jotka rekisteröidään tyyppivarastoon. Koska Pilarcos-meklariprototyyppi on osa laajempaa

Pilarcos-prototyyppiä, jossa meklauksessa käytetyt ominaisuudet ja yhteisöjen hallinnassa käytetyt politiikat on yhtenäistetty, kutsutaan ominaisuuksia prototyypissä *politiikoiksi*. Ominaisuuskehikkoja kutsutaan vastaavasti *politiikkakehikoiksi*, ja ne esitetään `PolicyFramework`-tietueina.

`LongRange`-, `DoubleRange`- ja `StringSet`-ominaisuustyypit ovat tärkeitä esimerkkejä siitä, millä tavoin ominaisuustyypeistä voidaan saada sekä ilmaisuvoimaisia että tehokkaita käsitellä. Ne eivät ole ainoita mahdollisia vaihtoehtoja. Eri ympäristöt vaativat erilaisia ominaisuuksien ilmaisutapoja, joita on harkittava huolellisesti, jotta saavutetaan tasapaino ilmaisuvoiman ja käsittelytehokkuuden välillä. Pilarcos-meklauksen käyttökelpoisuus riippuu ratkaisevasti tästä.

5.3.2 Palvelutyypit ja arkkitehtuurit

Palvelutyypit on toteutettu prototyypissä luvun 3.3.3 kuvauksen mukaisesti `ServiceType`-tietueina. Poikkeuksena aiemmasta kuvauksesta abstrakteja rajapintatyyppejä ei rekisteröidä tyyppivarastoon erikseen, vaan ainoastaan osana palvelutyyppiä. Abstraktit rajapintatyypit on toteutettu pelkkinä niminä; saman nimen oletetaan aina viittaavan samaan abstraktiin rajapintaan. Arkkitehtuurikuvaukset on toteutettu luvussa 3.4.1 kuvatulla tavalla.

5.3.3 Palvelu- ja yhteisötarjoukset

Palvelutarjouksen sisältävää tietorakennetta kutsutaan prototyypissä *palvelusopimukseksi*, koska sitä käytetään paitsi palvelutarjouksen myös lopullisen yhteistyösopimuksen kuvaamiseen Pilarcos-ympäristössä. `ServiceContract`-tietue, joka sisältää palvelusopimuksen, noudattaa luvussa 4.2 esitettyä palvelutarjouksen muotoa.

Tekniset rajapintatyypit ilmaistaan määrämuotoisilla merkkijonoilla, esimerkiksi "`IDL:/repository/Maksupalvelu/MaksuRajapinta:1.0`", joiden oletetaan olevan yksikäsitteisiä, kuten abstraktien rajapintatyyppienkin. Rajapintaviitteitä ei merkitä palvelusopimukseen sellaisinaan, koska eri väliohjelmistoalustoilla kuten CORBA ja Java-RMI on erilaiset rajapintaviitteet, joita ei voida esittää yhtenäisessä muodossa. Suorien viitteiden sijaan sopimukseen tallennetaan nimipalveluviitteitä, jotka ovat (*nimipalvelun osoite*, *rajapinnan nimi nimipalvelussa*)-merkkijonopareja. Näiden lisäksi palvelusopimus sisältää viitteen palvelun tarjoajan `FederationManager`-komponenttiin, jonka avulla yhteisöt Pilarcos-prototyypissä muodostetaan.

Yhteisötarjouksen sisältävää tietorakennetta kutsutaan Pilarcos-prototyypissä *federaatiosopimukseksi*, koska tietorakenne edustaa monenvälisen federoidun yhteisön keskinäistä sopimusta. Yhteisötarjouksia kutsutaan vastaavasti *federaatiotarjouksiksi*. Luvun 4.3 mukaisesti federaatiosopimuksen sisältävä `FederationContract`-tietue sisältää korkeintaan yhden `ServiceContract`-tietueen jokaista roolia kohti.

5.3.4 Tyyppivaraston rajapinnat

Tyyppivaraston prototyyppitoteutuksella on kaksi julkista rajapintaa. `TypeRegistration`-rajapinta sisältää tyyppien rekisteröintiä ja kyselyä varten `register`- ja `describe`-operaatiot ominaisuuskehikoille, palvelutyypeille ja arkkitehtuureille. Lisäksi rajapinta sisältää operaation, joka palauttaa arkkitehtuurin mukana myös arkkitehtuurin ominaisuuskehikkoverkot.

`TypeMatching`-rajapinta sisältää toistaiseksi vain yhden operaation, jonka avulla voi tarkistaa kahden teknisen rajapintatyypin yhteensopivuuden. Nykyisellään operaatio suorittaa vain merkkijonojen vertailun, mutta se on mahdollista laajentaa ottamaan huomioon tyyppien väliset suhteet (luku 4.1).

5.3.5 Meklarin rajapinnat

Pilarcos-meklaritoteutus tarjoaa kaksi rajapintaa. `OfferRegistration`-rajapinta sisältää operaatiot yksittäisten palvelutarjousten rekisteröintiin, kuvailuun ja poistamiseen. `Lookup`-rajapinta taas sisältää `populate`-operaation palvelukyselyjen tekemiseen. Operaatio ottaa parametreinaan vajaan yhteisötarjouksen, palautettavien yhteisötarjousten enimmäismäärän sekä pehmeän ylärajan kyselyn kestolle millisekunteina. Operaatio palauttaa listan löytyneistä yhteisötarjouksista.

Pilarcos-meklari lopettaa yhteisötarjousten etsinnän, kun jompikumpi parametrina annettu yläraja saavutetaan tai kun kaikki mahdolliset tarjousyhdistelmät on käyty läpi. Aikarajaan ei sisälly palvelukyselyn siirto kysyjältä Pilarcos-meklarille eikä tulosten siirto takaisin kysyjälle. Aikaraja on luonteeltaan pehmeä: Pilarcos-meklari tarkkailee etsinnän aikana pienin väliajoin järjestelmän kelloa, ja keskeyttää etsinnän, kun se huomaa aikarajan ylittyneen.

5.4 Toiminta

Pilarcos-meklariprototyyppi toimii pääosin luvun 4 kuvauksen mukaisesti. CORBA-meklaria hyödyntävässä toimintatilassa tarvitaan kuitenkin lisävaiheita, jotka eivät sisältyneet aiempaan kuvaukseen. Lisävaiheita seuraa myös Pilarcos-meklarin käytöstä osana laajempaa Pilarcos-prototyyppiä.

5.4.1 Palvelutyyppien ja palvelutarjousten tallentaminen

CORBA-meklaria hyödyntävässä toimintatilassa Pilarcos-meklari rekisteröi palvelutyypit CORBA-meklariin. Pilarcos-tyyppivaraston täytyy vastaavasti rekisteröidä palvelutyypit CORBA-palvelutyyppivarastoon, jotta CORBA-meklari voisi käyttää niitä.

Pilarcos-palvelutyyppi poikkeaa CORBA-palvelutyypistä kahdella tavalla. Pilarcos-palvelutyyppiin voi kuulua useita rajapintoja, CORBA-palvelutyyppiin vain yksi. Pilarcos-palvelutyypissä ominaisuustyypit on jaettu ominaisuuskehikoihin, mut-

ta CORBA-palvelutyypissä ei. Kun Pilarcos-palvelutyyppi rekisteröidään, tyyppivarasto muuntaa sekä rajapintatyyppi- että ominaisuuskehikkomäärittelyt CORBA-palvelutyypin ominaisuustyypeiksi. CORBA-palvelutyypin rajapintatyyppiä ei käytetä lainkaan.

Pilarcos-meklari tekee vastaavan muunnoksen palvelutarjouksille rekisteröinnin yhteydessä. Samalla meklari järjestää `StringSet`-tietueiden merkkijonot valmiiksi aakkosjärjestykseen, jotta merkkijonojoukkoja käsittelevät operaatiot saadaan etsinnän yhteydessä suoritettua $O(n)$-ajassa. Hakiessaan palvelutarjouksia CORBA-meklarilta Pilarcos-meklari suorittaa käänteisen muunnoksen.

Itsenäisessä toimintatilassa Pilarcos-meklari ei käytä CORBA-meklaria lainkaan, vaan tallentaa palvelutarjoukset palvelutyypeittäin omiin tietorakenteisiinsa.

5.4.2 Tarjousyhdistelmien etsintä

Palvelutarjousyhdistelmien etsintä on toteutettu luvussa 4.6 hahmotellulla tavalla, mutta CORBA-meklaria käytettäessä algoritmi muuttuu hiukan. Etsintäprosessi käynnistyy, kun meklari vastaanottaa palvelukyselyn eli `populate`-kutsun, ja se voidaan jakaa viiteen päävaiheeseen:

1. Kyselyssä annettujen palvelutarjousten ominaisuuksien leikkausten laskeminen.
2. Rajoitelausekkeen muodostaminen CORBA-meklaria varten (jokaiselle roolille erikseen).
3. Palvelukysely CORBA-meklarille (jokaiselle roolille erikseen).
4. Tarjousyhdistelmien etsintä.
5. Löytyneiden tarjousyhdistelmien ominaisuuksien kaventaminen yhteen arvoon.

Vaiheita käsitellään seuraavaksi tarkemmin.

Jos palvelukyselyssä parametrina annettu yhteisötarjous sisältää esitäytettyjä rooleja, meklari laskee niiden ominaisuuksien leikkaukset ensin, ja tallentaa leikkaukset verkkokohtaisiin ominaisuuskehikoihin.

Pilarcos-meklari hakee kuhunkin tyhjään rooliin palvelutarjouksia tekemällä palvelukyselyn CORBA-meklarille. Rajoitelausekkeeseen otetaan mukaan verkkokohtaisista ominaisuuskehikoista rooliin liittyvät perustyyppiset ominaisuudet. Rajoitelauseke on muotoa

`(not exist` *ominaisuus_1* `or` *ominaisuus_1* `=` *arvo*`) and`
`(not exist` *ominaisuus_2* `or` *ominaisuus_2* `=` *arvo*`) and ...`,

jossa kutakin ominaisuutta kohti on olemassaolo- ja yhtäsuuruustesti. Rajoitelausekkeen käyttö palvelukyselyssä estää CORBA-meklaria palauttamasta tarjouksia, jotka eivät ole yhteensopivia alkuperäisessä vajaassa yhteisötarjouksessa annettujen palvelutarjousten kanssa.

Koska rajoitelausekkeen ehdot täyttäviä palvelutarjouksia voi olla CORBA-meklarilla paljonkin, niitä ei siirretä Pilarcos-meklarille yhdellä kertaa. Pilarcos-meklari noutaa CORBA-meklarilta aluksi vain yhden erän, prototyypissä viisi kappaletta, palvelutarjouksia kullekin roolille. Kun kaikki rooliin noudetut palvelutarjoukset on käyty läpi, Pilarcos-meklari noutaa seuraavan viiden tarjouksen erän käyttäen CORBA-meklarin tarjousiteraattoria (`OfferIterator`).

CORBA-standardin mukaan tarjousiteraattoria käytettäessä tarjousten kokonaismäärää ei välttämättä aluksi tiedetä. Jos näin on, Pilarcos-meklari arvaa roolien järjestämistä varten määräksi kaksi erää eli kymmenen tarjousta. Erä kerrallaan hakemisen tehokkuus perustuu oletukseen, että CORBA-meklari palauttaa federoidussa tilanteessa paikalliset palvelutarjoukset ensin. Tätä ei kuitenkaan ole taattu CORBA-meklauspalvelustandardissa.

Itsenäisessä toimintatilassa mitään CORBA-meklarikyselyjä ei tehdä, vaan Pilarcos-meklari käyttää suoraan omaa palvelutarjoustietorakennettaan, josta tarjoukset ovat suoraan saatavilla palvelutyypeittäin.

Tarjousyhdistelmien etsintä tapahtuu luvun 4.6 mukaisella algoritmilla. Suositummuusjärjestystä prototyyppiin vielä ole toteutettu, mutta itsenäisessä toimintatilassa meklari sekoittaa kunkin palvelutyypin tarjoukset satunnaiseen järjestykseen aina uuden yhteisötarjouksen löydyttyä, jolloin kysyjä saa vastauksena toisistaan poikkeavia tarjousyhdistelmiä.

Kun kokonainen yhteisötarjous löytyy, se voitaisiin sinällään lisätä palautettavien yhteisötarjousten listaan, mutta Pilarcos-meklarin prototyyppitoteutus suorittaa tätä ennen vielä yhden vaiheen: arvoväli- ja joukkomuotoisten ominaisuuksien kaventamisen yhteen arvoon. Laajemmassa Pilarcos II-prototyypissä meklarin palauttaman yhteisötarjouksen pohjalta nimittäin muodostetaan federaatiosopimus, jota yhteisön osapuolet sitoutuvat noudattamaan keskinäisessä yhteistoiminnassaan. Lopullisessa sopimuksessa arvoväli- ja vaihtoehtojoukkomuotoisille politiikoille on sovittu yksittäiset arvot. Lopullinen sopimus voitaisiin muodostaa esimerkiksi erillisellä neuvottelukierroksella yhteisötarjouksen pohjalta. Pilarcos-prototyypin tapauksessa federaatiopolitiikat ovat kuitenkin niin yksinkertaisia, että ominaisuuksiin liitettäviä painokertoimia käyttäen meklari kykenee itse muodostamaan lopullisen sopimustarjokkaan.

Prototyypissä arvovälityyppisiin ominaisuuksiin sisältyy toivottu arvo, jota lähimpiä arvoja suositaan. `StringSet`-tyyppisten ominaisuuksien merkkijonoihin liittyy painokerroin; suuremman kertoimen omaavia vaihtoehtoja suositaan. Ennen yhteisötarjouksen palauttamista kysyjälle Pilarcos-meklari asettaa arvovälien lopulliseksi arvoksi suosittujen arvojen keskiarvon. `StringSet`-tyyppisten ominaisuuksien lopulliseksi arvoksi Pilarcos-meklari valitsee vaihtoehdoista sen, jolle eri palvelutarjouksissa annettujen painokerrointen summa on suurin. Tämä ratkaisu liittyy kiinteästi Pilarcos II-prototyyppiin, eikä välttämättä ole mielekäs muissa ympäristöissä.

5.5 Suorituskykymittaukset

Meklarin suorituskyky on tärkeä tekijä, joka vaikuttaa koko sitä käyttävän järjestelmän skaalautuvuuteen. Pilarcos-meklariprototyypillä tehtiin suorituskykymittauksia, joiden tavoitteena oli selvittää Pilarcos-meklauksen tehokkuutta ja skaalautuvuutta CORBA-ympäristössä. Vertailukohtana käytettiin tavanomaisten CORBA-meklauspalvelutoteutusten suorituskykyä. Seuraavissa luvuissa käsitellään mittausparametrit, mittausjärjestelyt, tulokset ja niistä kootut johtopäätökset.

5.5.1 Mittausparametrit

Vaikka meklauspalvelun käyttöskenaarioiden tyypillisiin piirteisiin kuuluu tarjoustietokannan tiheät päivitykset (luku 2.1), palvelutarjousten rekisteröinti tai muokkaaminen ovat tavallisesti palvelukyselyjä harvinaisempia tapahtumia. Näin on erityisesti avoimilla palvelumarkkinoilla, joita varten Pilarcos-prototyyppi on suunniteltu. Lisäksi palvelukysely on Pilarcos-meklauksen laskennallisesti raskain operaatio. Näistä syistä mittauksissa tutkittiin ainoastaan palvelukyselyjen tehokkuutta.

Koska Pilarcos-meklauspalvelu on avoin yhden palvelimen palvelu, vastausajat erilaisilla asiakasmäärillä voidaan laskea jonoteorian avulla palvelukyselyjen suoritusajoista. Niinpä mittauksissa käytettiin vain yhtä palvelukyselyjä suorittavaa mittausasiakasta. Meklarien federoinnin vaikutusta ei kokeiltu; se kuuluu jatkotyökohteisiin.

Mittauksia varten generoitiin palvelutyyppejä, arkkitehtuureja ja palvelutarjouksia erillisellä työkalulla. Generoidut arkkitehtuurit ovat ketjun muotoisia: ketjun päitä lukuunottamatta kukin rooli liittyy kahteen vierusrooliinsa yhdellä tai useammalla rajapinnalla. Jokaista roolia kohti on erillinen palvelutyyppi. Arkkitehtuurin muoto ei vaikuta etsinnän laskennalliseen vaativuuteen; ainoastaan roolien, rajapintojen ja ominaisuuskehikkojen määrällä sekä ominaisuuskehikkoverkkojen laajuudella on merkitystä. Generoiduissa arkkitehtuureissa ominaisuuskehikkoverkot ovat vain kahden roolin laajuisia, mikä on etsintäalgoritmin tehokkuuden kannalta huonoin mahdollinen tapaus.

Varsinaiset mittausparametrit olivat seuraavat:

- roolien (ja samalla palvelutyyppien) lukumäärä,
- palvelutarjousten kokonaismäärä ja jakauma,
- palvelutarjousten yhteensopivuusaste eli yhteisötarjousten osuus kaikista mahdollisista tarjousyhdistelmistä,
- rajapintojen määrä palvelutyyppiä kohti,
- ominaisuustyyppien määrä palvelutyyppiä kohti ja
- palautettavien yhteisötarjousten enimmäismäärä.

Palvelutarjousten jakaumaa säädeltiin kahdella parametrilla: tarjousten vähimmäis- ja enimmäismäärällä palvelutyyppiä kohti. Palvelutarjousten määrät eri palvelutyypeille (ja siten myös rooleille) saatiin interpoloimalla lineaarisesti vähimmäis- ja enimmäismäärän välillä.

Generoidut palvelutarjoukset olivat kaikki keskenään yhteensopivia yhtä roolia lukuunottamatta. Tässä niin sanotussa etsinnän katkaisevassa roolissa vain yhteensopivuusasteen määrittelemä osuus tarjouksista oli yhteensopivia. Kaikissa mittaussarjoissa etsinnän katkaiseva rooli sijoittui etsintäpuussa tasolle 3/4 puun korkeudesta eli roolien kokonaismäärästä. Etsittäessä vain muutamia yhteisötarjouksia etsinnän katkaisutasolla ei ole suorituskyvylle lainkaan merkitystä, koska meklarin etsintäalgoritmi peruuttaa vain yhdellä tasolla kerrallaan törmätessään epäyhteensopivaan tarjousyhdistelmään. Tämä on varmistettu mittauksin.

Generoiduissa arkkitehtuureissa jokaisen roolin välillä on vähintään kaksi rajapintaa. Kuhunkin rajapintaan liittyy yksi ominaisuuskehikko, jonka ominaisuustyypeistä puolet on `LongRange`- ja `StringSet`-tyyppisiä, toinen puolet `long`- ja `boolean`-tyyppisiä. Ominaisuuksien määrää säädeltiin ominaisuuskehikkojen kokoa muuttamalla.

Kutakin mittausparametria varioitiin erikseen perustapauksesta siten, että kussakin mittaussarjassa vaihdettiin vain yhden parametrin arvoa. *Perustapauksen* parametrien arvot olivat:

- neljä roolia,
- 40, 70 ja 100 palvelutarjousta kolmea roolia kohti (yhteensä 210),
- 20 prosentin yhteensopivuusaste,
- neljä rajapintaa palvelutyyppiä kohti,
- 32 ominaisuustyyppiä palvelutyyppiä kohti ja
- yksi palautettava federaatiotarjous (eli yhteisötarjous).

Perustapaus vastaa vaativaa, mutta realistista käyttötapausta Pilarcos-meklarille.

Mittausasiakas teki palvelukyselyjä käyttäen vajaata yhteisötarjousta, jossa yksi rooli oli valmiiksi täytetty. Pilarcos-meklari etsi yhteensopivat palvelutarjoukset muihin rooleihin. Perustapauksessa mahdollisia yhdistelmiä oli siis kaikkiaan 280 000, joista 56 000 oli kelvollisia yhteisötarjouksia.

5.5.2 Mittausjärjestelyt

Palvelukyselyn suoritus Pilarcos-meklarissa jaettiin vaiheisiin, joiden kestot mitattiin erikseen. Vaiheet olivat palvelukyselyn alustus Pilarcos-meklarissa, palvelukyselyt CORBA-meklarille, ensimmäisen yhteisötarjouksen löytyminen, palvelukyselyn suorituksen päättyminen ja löytyneiden yhteisötarjousten palauttaminen kysyjälle. Kaikki mittaukset suoritettiin sekä Pilarcos-meklarin itsenäisessä toimintatilassa että CORBA-meklaria tarjoustietokantana käyttävässä toimintatilassa.

Mittaukset tehtiin kahdella 1 GHz Pentium III-työasemalla, joissa oli 512 MB RAM-muistia. Käyttöjärjestelmänä oli Linuxin ytimen versio 2.2. Toisessa työasemassa ajettiin mittausasiakasta ja toisessa Pilarcos-meklaria. CORBA-meklarina käytettiin ORBacus-meklarin Java-pohjaista versiota 2.0.0 [9], jota ajettiin IBM Java2 1.4.0-virtuaalikoneella kääntävässä tilassa samassa työasemassa kuin Pilarcos-meklaria. Ajat mitattiin Linuxin `gettimeofday`-käyttöjärjestelmäkutsua käyttäen millisekunnin tarkkuudella.

Generointityökalulla tuotettiin 40 järjestykseltään satunnaistettua palvelutarjoustietokantaa jokaiselle eri mittausparametrin arvolle. Kaikki palvelukyselyt tehtiin vielä kahteen kertaan, jolloin saatiin yhteensä 80 mittaustulosta mittauspistettä kohti. Palvelutarjoustietokanta tyhjennettiin ja palvelutarjoukset rekisteröitiin uudelleen meklariin palvelukyselyjen välissä. Mittaussarjojen välillä sekä Pilarcos-meklari että ORBacus-meklari ajettiin alas ja käynnistettiin uudelleen. ORBacus-meklaria käytettäessä järjestelmää lämmitettiin kolmella palvelukyselyllä ennen varsinaisia mittauksia.

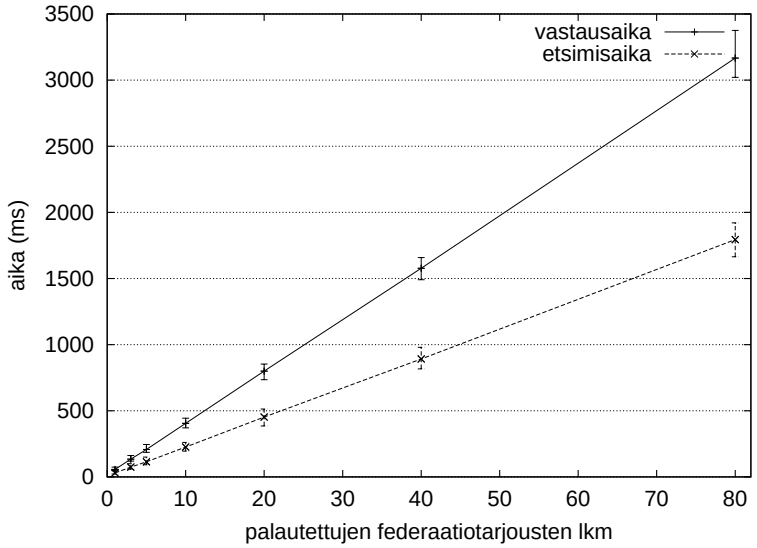
5.5.3 Mittaustulokset

Vaikka kaikki mittaukset suoritettiin sekä itsenäisessä että CORBA-meklaria hyödyntävässä toimintatilassa, tässä luvussa esitellään vain itsenäistä toimintatilaa käyttäen saatuja tuloksia. Näissä tuloksissa Pilarcos-meklarin algoritmin ominaisuudet ovat selvimmin esillä. Luvun lopussa esitellään vertailun vuoksi muutamia CORBA-meklaria hyödyntävässä tilassa saatuja mittaustuloksia.

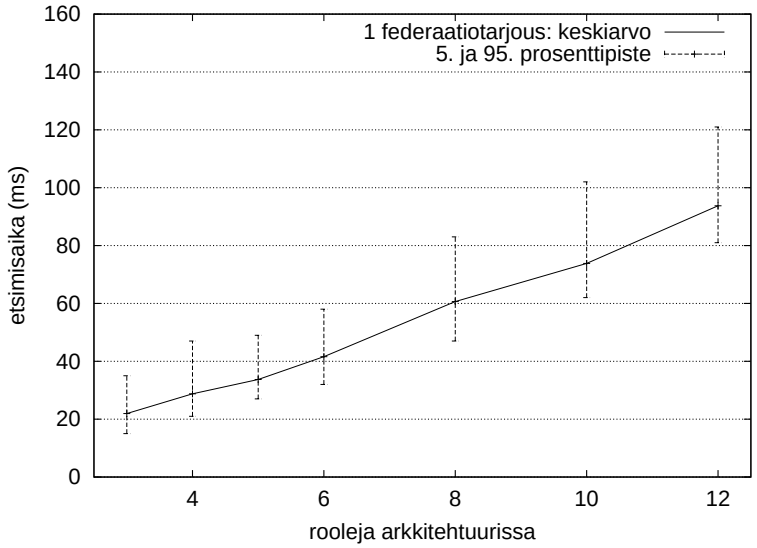
Useimpiin mittaustuloksia esittäviin kuviin on merkitty 80 havaintoarvon keskiarvo yhtenäisellä viivalla. Lisäksi hajontaa on havainnollistettu merkitsemällä havaintoarvojen jakauman 5. ja 95. prosenttipisteen välinen osuus kunkin mittauspisteen kohdalle. Tähän osuuteen sisältyy siis 90 prosenttia havaintoarvoista eli 72 arvoa. Yleisesti havaintoarvojen jakauma ei ole normaali eikä edes symmetrinen, vaan riippuu mittauspisteestä.

Kuva 5.2 esittää palautettavien yhteisötarjousten määrän vaikutusta palvelukyselyn kestoon. Yleensä kysyjä rajoittanee yhteisötarjousten enimmäismäärän muutamaan, ehkä korkeintaan kymmeneen. Mittaussarjassa määrät ovat epätavallisen suuria, jotta parametrin vaikutus saadaan selvästi näkyviin. Kuten kuvasta ilmenee, odotusten mukaisesti palvelukyselyn kesto on suoraan verrannollinen palautettavien yhteisötarjousten määrään. Hiukan yllättäen palvelukyselyn kokonaiskesto on kuitenkin yli puolitoistakertainen etsintäalgoritmin käyttämään aikaan nähden.

Perustapauksessa palvelukyselyn keskimääräinen kesto jakaantuu seuraavasti: palvelukyselyn alustus 6 millisekuntia, varsinainen yhteisötarjouksen etsintä 24 millisekuntia ja yhteisötarjousten siirto 25 millisekuntia. Jälkimmäinen koostuu vajaan yhteisötarjouksen siirrosta palvelukyselyn parametrina ja löydetyn yhteisötarjouksen siirrosta palvelukyselyn vastauksena. Selvästi yhteisötarjoustietorakenteen siirto CORBA-yhteyden yli on raskasta, ja vie suuren osan palvelukyselyn kokonaisajasta. Tämä johtuu siitä, että yhteisötarjoustietueen IDL-määritelmässä pyrittiin joustavuuteen ja selkeyteen siirtotehokkuuden kustannuksella. Tietue sisältää runsaasti



Kuva 5.2: Palautettavien yhteisötarjousten määrän vaikutus palvelukyselyn kestoon.

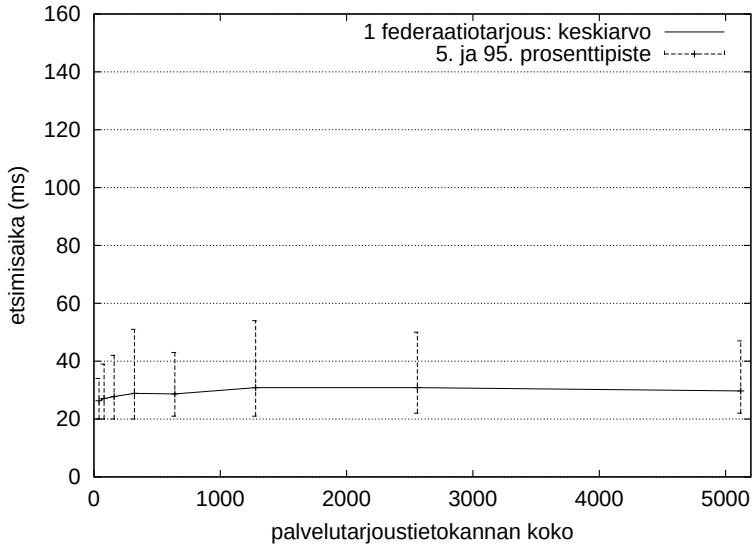


Kuva 5.3: Roolien määrän vaikutus etsintäaikaan.

sisäkkäisiä tietueita ja listoja sekä `Any`-tyyppisiä kenttiä.

Koska siirtoviive tunnetaan ja se on vakio, se on jätetty pois jäljempänä esitettävistä tuloksista, jolloin etsintäalgoritmin ominaisuudet ovat paremmin nähtävissä.

Roolien määrän vaikutusta etsintäaikaan on tutkittu kuvan 5.3 mittaussarjassa. Käytetyillä arkkitehtuurikuvauksilla myös ominaisuuskehikkojen määrä muuttuu suorassa suhteessa roolien määrään. Tässä mittaussarjassa jokaiselle roolille oli poikkeuksellisesti yhtä monta palvelutarjousta, ja niiden kokonaismäärä pidettiin vakiona (210). Etsintäajan riippuvuus roolien määrästä on lineaarinen ja nousu tar-



Kuva 5.4: Palvelutarjousten kokonaismäärän vaikutus etsintäaikaan.

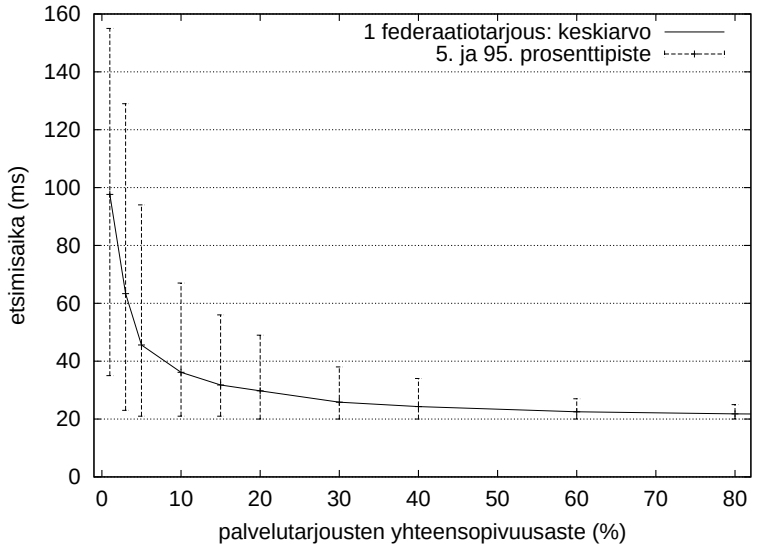
peeksi loiva, jotta jopa kymmenen roolin kokoiset arkkitehtuurit ovat käyttökelpoisia. Lisäksi on huomattava, että reaalitilanteessa ei välttämättä ole tarpeen täyttää kaikkia arkkitehtuurin rooleja yhdellä palvelukyselyllä.

Kuvassa 5.4 on kuvattu palvelutarjoustietokannan koon vaikutusta etsintäaikaan. Koska algoritmi on syvyyssuuntainen roolien suhteen ja pysähtyy löydettyään yhteisötarjouksen, ja koska kaikki palvelutarjoukset mahtuvat keskusmuistiin, ei palvelutarjousten määrä vaikuta lainkaan. Pilarcos-meklari on suunniteltu nimenomaan keskusmuistipohjaiseksi; levymuistia on tarkoitus käyttää lähinnä vain tilan säilyttämiseen poikkeustapauksista toipumista varten.

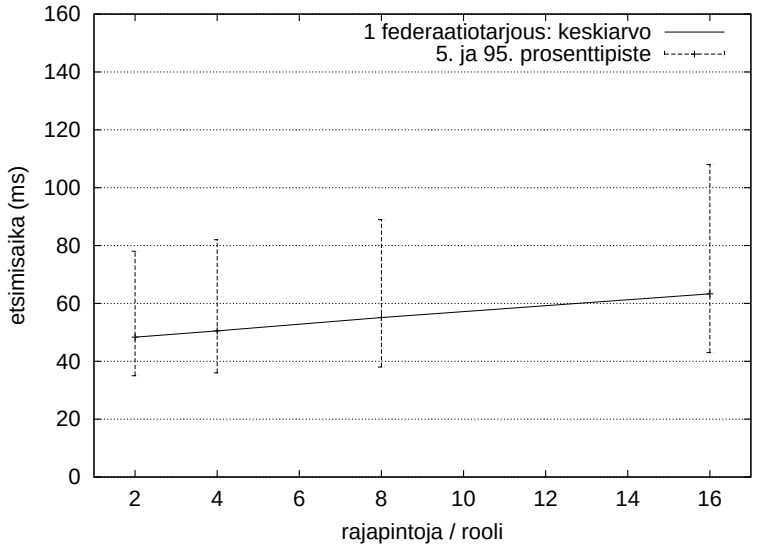
Palvelutarjousten keskinäinen yhteensopivuusaste on tärkeimpiä etsintäaikaan vaikuttavia tekijöitä. Mittaustulokset sen vaikutuksesta on esitetty kuvassa 5.5. Tulosten mukaan etsintäaika on keskimäärin kääntäen verrannollinen yhteensopivuusasteeseen, kuten algoritmin toiminnastakin voidaan päätellä. Käytännössä yhteensopivuusaste ei yleensä ole riippumaton parametri, vaan esimerkiksi arkkitehtuurin roolien määrä vaikuttaa siihen.

Keskimääräinen ja maksimaalinen etsintäaika nousevat siis jyrkästi yhteensopivuusasteen laskiessa. Silti vielä yhden prosentin yhteensopivuusasteellakin palvelukyselyn kesto pysyy kohtuullisena, keskimäärin alle 100 millisekunnissa, ja viidestä prosentista alkaen keskimäärin alle 50 millisekunnissa. Yhteensopivuusaste koskee vain kyselyn arkkitehtuurikuvauksessa esiintyviä palvelutyyppejä; arkkitehtuurit ja palvelutarjoukset on mahdollista laatia siten, että yhteensopivuusaste yleisimmissä käyttötapauksissa pysyy riittävän korkeana. Esimerkiksi erilaisiin käyttötilanteisiin tarkoitetuille samankaltaisille palveluille on järkevää rekisteröidä erilliset palvelutyypit. Myös kysyjä voi vaikuttaa palvelukyselyn kestoon asettamalla etsinnälle aikarajan ja väljentämällä tarvittaessa kyselyn ehtoja.

Kuvassa 5.6 on esitetty palvelutyyppien rajapintojen määrän vaikutus etsintäai-

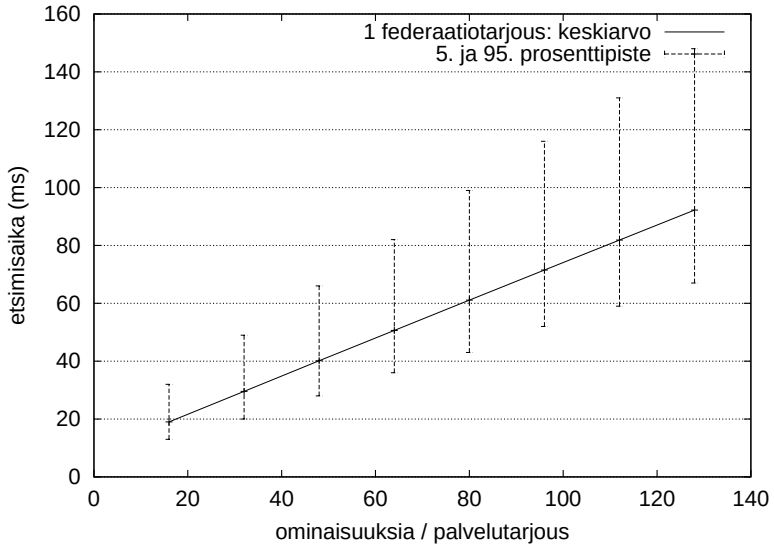


Kuva 5.5: Yhteensopivuusasteen vaikutus etsintäaikaan.

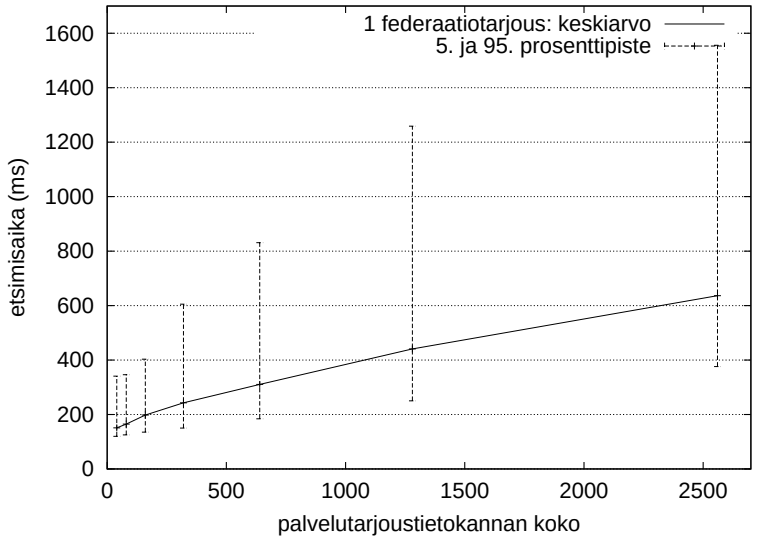


Kuva 5.6: Rajapintojen määrän vaikutus etsintäaikaan.

kaan. Kaikissa palvelutyypeissä oli sama määrä rajapintoja. Odotetusti riippuvuus on lineaarinen ja melko heikko. Tämä johtuu siitä, että etsintäalgoritmi tekee yhden `isCompatibleInterface`-kutsun tyyppivarastolle sidosta kohti, ja nykytoteutuksessa kyseinen operaatio ainoastaan vertailee merkkijonoja. Vaikutus olisi hiukan suurempi tilanteessa, jossa tyyppivarasto ja meklari olisi hajautettu erilleen tai jossa tyyppivarasto suorittaisi monimutkaisempia yhteensopivuustestejä. Rajapintojen määrä riippuu palvelutyypistä, ja lienee yleensä alle 10, joten parametrin merkitys on käytännössä vähäinen.



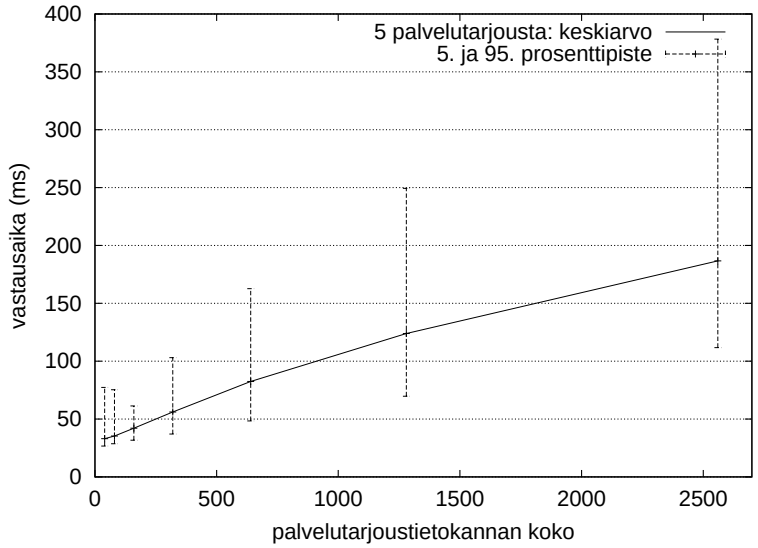
Kuva 5.7: Ominaisuuksien määrän vaikutus etsintäaikaan.



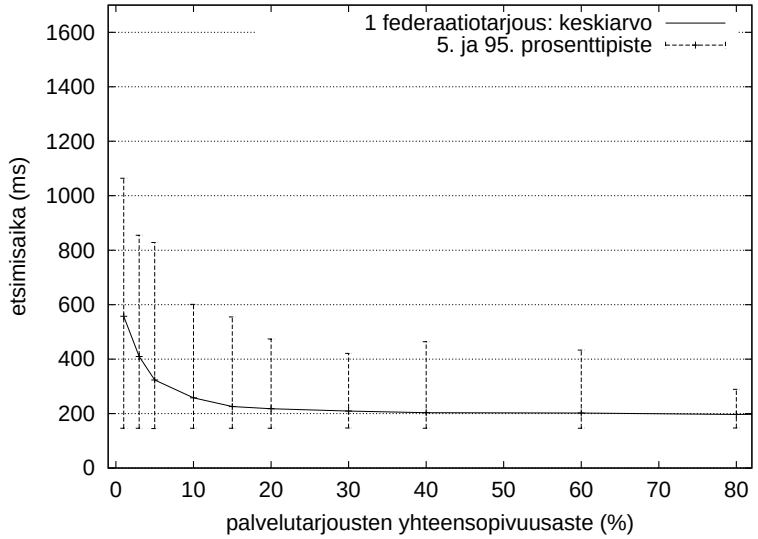
Kuva 5.8: Palvelutarjousten määrän vaikutus ORBacus-meklaria käytettäessä.

Ominaisuuksien määrä palvelutarjousta kohti vaikuttaa etsintäaikaan selvästi, kuten nähdään kuvasta 5.7. Sekä etsintäajan keskiarvo että hajonta kasvavat suorassa suhteessa ominaisuuksien määrään. Pilarcos-meklarin skaalautuvuus on erittäin hyvä: 32 ominaisuutta palvelutarjousta kohti on melko paljon, 64 poikkeuksellisen paljon, mutta keskimääräinen etsintäaika pysyy jälkimmäisessäkin tapauksessa noin 50 millisekunnissa.

Kuvassa 5.8 on esitetty palvelutarjousten määrän vaikutus etsintäaikaan käytettäessä ORBacus-meklaria palvelutarjoustietokantana. Itsenäiseen toimintatilaan



Kuva 5.9: ORBacus-meklarin vastausaika viiden tarjouksen palvelukyselyyn.



Kuva 5.10: Yhteensopivuusasteen vaikutus etsintäaikaan ORBacus-meklaria käytettäessä.

(kuva 5.4) verrattuna ero on huomattava: etsintäaika riippuu ORBacus-meklaria käytettäessä voimakkaasti palvelutarjousten määrästä. Lisäksi keskimääräinen etsintäaika on moninkertainen.

Kuva 5.9 esittää ORBacus-meklarin vastausaikaa palvelukyselyyn, jossa pyydetään viisi palvelutarjousta. Kuvan tulokset on eristetty Pilarcos-meklarin mittauksista, joten niissä on käytetty samoja palvelutarjouksia ja -kyselyjä kuin Pilarcos-meklarin mittaussarjassa. Tuloksista nähdään, että ero itsenäisen ja ORBacus-mek-

laria käyttävän toimintatilan välillä johtuu lähes yksinomaan ORBacus-meklarin käyttäytymisestä.

ORBacus-meklarin suhteellisen huono suorituskyky näissä mittauksissa johtuu palvelutarjousten ominaisuuksien suuresta määrästä. Palvelukyselyssä pyydetään vastauksena kaikki palvelutarjousten ominaisuusarvot, ja niiden siirtäminen CORBA-kanavan yli on `Any`-tyypin käytön vuoksi raskasta. Myös palvelukyselyssä annettava rajoitelauseke on pitkä ja monimutkainen (luku 5.4.2) ja siksi hidas tulkittava. CORBA-meklaria ei ilmeisesti ole suunniteltu tällaiseen käyttötapaukseen, vaan on oletettu, että rajoitelausekkeet ovat suhteellisen yksinkertaisia ja että vastauksessa ei tarvitse siirtää mukana kaikkia ominaisuusarvoja.

Kuvassa 5.10 esitettyä yhteensopivuusasteen vaikutusta etsintäaikaan ORBacus-meklaria käytettäessä voidaan verrata itsenäisen toimintatilan tuloksiin kuvassa 5.5. Riippuvuus on samankaltainen, mutta ORBacus-meklaria käytettäessä etsintäajat ovat lähes kymmenkertaisia. Pilarcos-meklarin CORBA-meklaria käyttävästä algoritmista johtuen palvelukyselyn peruskustannus on suhteellisen suuri, koska alustuksen yhteydessä Pilarcos-meklari hakee CORBA-meklarilta viisi tarjousta jokaiseen tyhjään rooliin. ORBacus-meklarin Java-pohjaisuudesta aiheutuvat roskankeruuviiveet aiheuttavat tuloksiin hajontaa myös suurilla yhteensopivuusasteilla.

5.5.4 Johtopäätökset

Mittauksissa käytetyssä perustapauksessa Pilarcos-meklarin keskimääräinen vastausaika palvelukyselyyn oli 55 millisekuntia, kun CORBA-meklaria ei käytetty. Sadan prosentin käyttöasteella tämä vastaa noin 18 palvelukyselyä sekunnissa. Vastausajasta 25 millisekuntia eli lähes puolet kului kyselyn ja vastauksen siirtoon CORBA-yhteyden yli. Ottaen tämä vakioviive huomioon voidaan varsinaista etsintäalgoritmia pitää suorituskyvyltään hyvänä.

Mittausten perusteella palvelutarjousten yhteensopivuusaste on Pilarcos-meklauksen kannalta kriittisin tekijä. Etsintäalgoritmi on käyttökelpoinen vielä yhden prosentin yhteensopivuusasteella, mutta keskimäärin vaaditaan vähintään viiden prosentin yhteensopivuusaste hyvän suorituskyvyn varmistamiseksi. Tähän voidaan pyrkiä suunnittelemalla palvelutyypit ja arkkitehtuurikuvaukset oikein. Pilarcos-meklariin voitaisiin myös lisätä välimuisti vaikeimmin löydettäviä kokonaisia tai osittaisia palvelutarjousyhdistelmiä varten, mikä nopeuttaisi palvelukyselyjä alhaisen yhteensopivuusasteen tapauksissa.

Pilarcos-meklarin prototyyppitoteutus ei ole kaikilta osin optimoitu. Meklari esimerkiksi säilyttää ominaisuusarvot sisäisesti `Any`-tyyppisinä olioina, mikä aiheuttaa turhaa työtä etsintäalgoritmissa. Todennäköisesti optimoinnilla absoluuttiset etsintäajat voitaisiin saada pienenemään puoleen tai allekin, mutta etsintäalgoritmin asymptoottista käyttäytymistä eri parametrien suhteen tämä ei muuttaisi.

Yhteensopivuusastetta lukuunottamatta etsintäaika riippuu korkeintaan lineaarisesti eri mittausparametreista. Tämä merkitsee, että algoritmi ja ominaisuustyypit on valittu oikein, ja että kokonaisuus on niin skaalautuva kuin luonteeltaan kombinatorisen ongelman tapauksessa voidaan olettaa mahdolliseksi.

Käytännön kannalta ongelmallisin mittauksissa paljastunut tekijä on palvelutarjousten siirtämisen raskaus. Yhteisötarjoukset koostuvat palvelutarjouksista, joten ongelma koskee myös niitä. Erityisen selvästi ongelma tulee esiin CORBA-meklaria hyödyntävässä toimintatilassa, joka on itsenäistä toimintatilaa selvästi hitaampi paljolti juuri siirtoviiveen vuoksi — palvelutarjoukset on aina siirrettävä CORBA-meklarilta Pilarcos-meklarille ennen kuin Pilarcos-meklari voi kokeilla niiden yhdistelmiä.

CORBA-meklaria hyödyntävää toimintatilaa voitaisiin huomattavasti nopeuttaa sijoittamalla CORBA-meklari ja Pilarcos-meklari samaan prosessiin. Nykyisessä prototyyppitoteutuksessa näin ei voitu tehdä, koska ORBacus-meklari ja Pilarcos-meklari toimivat eri CORBA-alustoilla. Samaan prosessiin sijoittaminen ei kuitenkaan ratkaise vakavampaa ongelmaa: tarjousten siirtoa federoitujen meklarien välillä. Kun CORBA-meklarit kytketään toisiinsa meklarifederaatioksi, sekä palvelukyselyt että vastaukset saattavat kulkea usean perättäisen linkin yli. Vaikka prosessi olisi osittain rinnakkainenkin, ilman välimuistiratkaisuja se on hyvin raskas, erityisesti, jos palvelutarjouksien kaikki ominaisuudet joudutaan siirtämään vastausten mukana. CORBA-meklauspalvelua ei ilmeisesti ole suunniteltu tähän; tavallisissa käyttötilanteissa riittää usein pelkän rajapintaviitteen siirtäminen.

Sekä Pilarcos- että CORBA-meklarin palvelutarjoustietueissa ominaisuudet voivat olla mitä tahansa OMG IDL-tyyppiä, joten ne koodataan `Any`-tyyppisiksi. `Any`-tyyppisten arvojen siirtäminen on tunnetusti raskasta erityisesti, jos arvot ovat muita kuin perus-IDL-tyyppejä. Rajoittamalla ominaisuustyyppien joukkoa ja luopumalla `Any`-tyypin käytöstä palvelutarjousten siirtoa voitaisiin tehostaa huomattavasti. CORBA-meklarin osalta tämä edellyttäisi meklauspalvelustandardista poikkeamista tai sen muuttamista.

Pilarcos-ympäristössä, jossa palvelut voivat toimia monimutkaisessakin yhteistyösuhteessa keskenään, palvelujen kuvaamiseen tarvitaan melko paljon ominaisuuksia palvelutarjousta kohti. Palvelutarjousten siirtäminen verkon yli on siten optimoitunakin kohtuullisen raskasta. Pilarcos-meklarissa käytetty suunnitteluratkaisu, jossa palvelutarjoukset ja etsintäalgoritmi keskitetään samaan paikkaan, on siten tehokkuusnäkökulmasta oikea. Pilarcos-meklarien federointi edellyttää palvelutarjousten siirtämisen minimointia esimerkiksi alueellisen jaon ja välimuistiratkaisujen avulla.

Luku 6

Jatkokehityskohteita

Kehitetty Pilarcos-meklauskäsitteistö ja prototyyppitoteutus tarjoavat useita mahdollisia jatkokehityskohteita. Tässä luvussa käydään läpi joitakin niistä lähtien prototyyppitoteutuksen parantamisesta kohti suurempia, käyttöaluetta laajentavia kehityskohteita.

6.1 Prototyyppitoteutuksen parantaminen

Pilarcos-meklariprototyypin ominaisuustyyppien valikoima on nykyisellään melko suppea; sitä kannattaisi laajentaa sovelluskokeilujen perusteella tehtävän vaatimusanalyysin perusteella. Valikoima kannattaa silti pitää rajoitettuna ominaisuuskehikkojen uudelleenkäytettävyyden ja meklarin suorituskyvyn takia. Prototyypin suorituskykyä voitaisiin tehostaa useilla optimoinneilla, joita käsiteltiin luvussa 5.5.4. Selvimpänä näistä nousee esiin CORBAn `Any`-tyyppien käytön vähentäminen.

Laaja-alaista käyttöä varten toteutus tulisi laajentaa tukemaan meklarien federointia. Verkkoliikenteen kasvua ja tarjousten siirrosta aiheutuvia suorituskykyongelmia voitaisiin vähentää tarjousvälimuistin avulla. CORBA-meklarille on kehitetty tarjousvälimuistiin ja kyselyjen tehokkaaseen reitittämiseen perustuvia suuriin, dynaamisiin verkkoympäristöihin tarkoitettuja laajennoksia [4, 5], joita voitaisiin soveltaa myös Pilarcos-meklariin. Kyselyvälimuistin hyöty Pilarcos-meklarille olisi vähäisempi kuin tarjousvälimuistin, koska oletettavasti Pilarcos-meklarille tehdään vain hyvin vähän identtisiä tai lähes identtisiä palvelukyselyjä.

6.2 Käsitteellinen kehittäminen

Luvussa 3.4.3 ehdotettiin, että ominaisuuskehikot voitaisiin liittää kiinteästi osaksi abstraktin rajapintatyypin määritelmää. Vaikka tämä muutos on käsitteellisesti melko suoraviivainen, se edellyttäisi runsaasti muutoksia Pilarcos-meklarin prototyyppitoteutukseen. Jos Pilarcos-meklari siirretään tulevaisuudessa uuteen ympäristöön (esimerkiksi web-palveluympäristöön), muutos kannattaa tehdä.

Pilarcos-meklari on nykymuodossaan melko suoraviivaista laajentaa käsittelemään monenvälisiä sidoksia, kuten luvussa 3.4.3 todettiin. Käytännön toteutuksessa ongelmaksi muodostunee, millä tavoin monenväliset sidokset teknisesti toteutetaan ja miten sovellukset saadaan ottamaan huomioon esimerkiksi ryhmälähetyksen semantiikka.

Vaikka Pilarcos-arkkitehtuuri organisaatioiden välisen luonteensa vuoksi ei voi perustua yhteiseen, yhtenäiseen tyyppihierarkiaan (luku 4.1), käytännön ohjelmistosuunnittelussa palvelutyyppien perintä olisi käyttökelpoinen väline. Perinnän semantiikka vaatisi huolellista harkintaa. Vielä tärkeämpää olisi lisätä Pilarcos-meklariin ominaisuuskehikkojen koostamismahdollisuus. Suurissa sovelluksissa tämä on tarpeen ominaisuuskehikkojen määrän ja laajuuden hallitsemiseksi.

Jotta automaattinen palvelujen löytäminen avoimilta markkinoilta olisi käytännössä realistista, Pilarcos-meklarin täytyisi ottaa huomioon ainakin kysyjän ilmaisemat suositummuudet. Suositummuuslausekkeen käyttö samaan tapaan kuin CORBA-meklarissa — hakemalla ensin suuri joukko yhteisötarjouksia ja järjestämällä ne sitten lausekkeen mukaan — on suorituskyvyn kannalta huono vaihtoehto, koska kokonaisten yhteisötarjousten muodostaminen on verraten hidasta. Tehokkaampaa olisi järjestää kunkin palvelutyypin tarjoukset ennen etsinnän aloittamista suositummuusjärjestykseen kysyjän kriteerien mukaan. Koska ominaisuuksien lopulliset arvot ratkeavat vasta yhteisötarjousta muodostettaessa, ratkaisu ei ole täydellinen, mutta käytännössä luultavasti riittävä.

6.3 Käyttöalueen laajentaminen

Kuten johdannossa (luku 1) mainittiin, web-palvelujen löytämistä varten on kehitteillä erilaisia hakemistoja, joista vahvimmin on noussut esille UDDI. Nykymuodossaan UDDI-hakemistosta etsitään vain yksittäisiä palveluja ja palveluntarjoajia, ja haku on tarkoitettu tehtäväksi käsin tai työkalujen avulla. UDDI ei siis ole tarkoitettu suoritusaikaiseen, automaattiseen sidontaan kuten meklauspalvelut.

Pilarcos-meklarin yhteydessä esitettyjä periaatteita — arkkitehtuurikuvauksia ja ominaisuuskehikkoja — on mahdollista soveltaa web-palveluihin. UDDI-hakemiston yhteyteen voitaisiin esimerkiksi rakentaa Pilarcos-meklauspalvelin, jolta voisi tehdä suoritusaikaisia hakuja rajatusta joukosta web-palveluja arkkitehtuurikuvauksen perusteella.

Ehkä suurin ongelma julkisessa palveluhakemistossa ja etenkin sen automatisoidussa hyödyntämisessä on ontologioiden yhteensopivuus. Palvelujen kuvaamiseen käytettyjen käsitteiden joukon täytyy olla yhteisesti sovittu ja niiden semantiikan täsmällisesti määritelty, jotta automaattisten hakujen tuloksiin voitaisiin luottaa. Tällaiset määritelmät eivät kuitenkaan jousta tekniikan ja palvelujen kehittyessä. Tämä lienee estänyt meklauspalvelujen käytön yleistymisen itsenäisesti hallittujen järjestelmien välillä.

Pilarcos-arkkitehtuuri tarjoaa osittaiseksi ratkaisuksi ongelmaan palvelujen kuvaamisessa käytettyjen käsitteiden määrittelyn ja tallentamisen julkiseen tyyppiva-

rastoon, johon tallennetaan myös palvelukuvausten väliset muunnossuhteet. Muunnossuhteet toteutetaan käytännössä ohjelmistokomponentteina, jotka asennetaan suoritusaikana tarvittaessa palvelujen väliseen viestintäkanavaan. Yleensä muunnokset edellyttävät abstraktien tyyppien eli tyyppijärjestelmän peruskäsitteiden (luku 4.1) olevan samat muunnettavien tyyppijärjestelmien välillä.

Muita tarpeita julkisessa, automaattisessa palveluhakemistossa ovat luotettavuus ja turvallisuus sekä ajantasaisuus. Ajantasaisuutta voidaan ylläpitää Jini-tekniikan tapaan määrittelemällä palvelutarjouksille voimassaoloaika, jonka jälkeen ne poistetaan automaattisesti, jollei palveluntarjoaja ole uudistanut niitä.

Luku 7

Yhteenveto

Perinteinen meklaus perustuu kahdenväliseen asiakas-palvelin-malliin: meklauspalvelusta on mahdollista etsiä vain yhden tyyppistä palvelua kerrallaan. Tilanteessa, jossa tarvitaan useita yhteistyössä toimivia palveluja, palvelut on kiinnitettävä etukäteen tai asiakassovelluksen on itse huolehdittava eri palvelujen yhteensopivuudesta.

Pilarcos-meklaus laajentaa perinteistä meklausta monenvälisiin hajautettuihin sovellusarkkitehtuureihin. Palveluja tarjoavien ja käyttävien komponenttien muodostama yhteisö kuvataan eksplisiittisellä arkkitehtuurikuvauksella, joka koostuu rooleista ja niiden välisistä sidoksista. Pilarcos-meklari etsii palvelukyselyssä annetun arkkitehtuurikuvauksen tyhjiin rooleihin sopivat palvelut yhdellä kertaa. Meklarin muodostamia palvelutarjousyhdistelmiä kutsutaan yhteisötarjouksiksi. Pilarcos-meklari ottaa etsinnässään huomioon kysyjän vaatimusten lisäksi myös palvelujen keskinäisen yhteensopivuuden.

Palvelujen kuvaaminen perustuu palvelutyyppeihin, jotka rekisteröidään julkiseen tyyppivarastoon. Jokaiseen arkkitehtuurikuvauksen rooliin, myös asiakkaan, liittyy jokin palvelutyyppi. Palvelutyypillä kuvataan paitsi toteutuskomponentin tarjoamat, myös sen käyttämät rajapinnat. Palvelu voi siis määritellä tarvitsevansa toimiakseen toisia palveluja. Palvelun kysyjä kuvaa ominaisuutensa ja vaatimuksensa Pilarcos-meklauksessa samalla tavoin kuin palvelun tarjoajat, joten asiakkaat ja palvelimet ovat symmetrisessä asemassa.

Yhteisötarjousten muodostaminen palvelutarjoustietokannasta on kombinatorinen ongelma ja siten laskennallisesti vaativa. Sen ratkaisemiseksi tutkielmassa on esitetty algoritmi, joka suorittaa pienimmän mahdollisen määrän vertailuja palvelutarjousten ominaisuuksien välillä.

Pilarcos-meklarista on toteutettu prototyyppi CORBA-alustalla. Meklariprototyypillä on kaksi toimintatilaa: itsenäinen ja CORBA-meklaria hyödyntävä. Suorituskykymittausten perusteella Pilarcos-meklari on itsenäisessä toimintatilassa riittävän tehokas moniin sovelluksiin, ja skaalautuu hyvin arkkitehtuurin koon ja palvelutarjousten määrän kasvaessa. CORBA-meklaria hyödyntävässä toimintatilassa suorituskyvyn pullonkaulaksi muodostuu palvelutarjousten siirtäminen prosessien välillä. Sama ongelma tulisi esiin Pilarcos-meklarifederaatioissa, joten palvelutar-

jouksien siirtoa pitäisi tehostaa ja siirtojen määrää vähentää välimuistiratkaisujen avulla, jotta skaalautuvuus säilyisi.

Pilarcos-meklariprototyypissä ei ole mahdollista ilmaista palveluille niiden ominaisuuksien mukaista suositummuusjärjestystä. Tutkielmassa on kuitenkin hahmoteltu, millä tavoin suositummuudet voitaisiin ottaa mukaan ilman suuria suorituskykyvaikutuksia. Pilarcos-meklauksen pääperiaatteet ovat sovellettavissa muuallekin kuin CORBA-ympäristöön, ja tulevaisuudessa näitä periaatteita hyödyntävää palvelujen löytämistä kokeiltaneen web-palvelujen yhteydessä.

Kirjallisuutta

[1] Allen, R. J. *A Formal Approach to Software Architecture.* Ph.D. thesis, School of Computer Science, Carnegie Mellon University, Pittsburgh, May 1997. Myös `http://www-2.cs.cmu.edu/afs/cs/project/able/www/paper_abstracts/rallen_thesis.htm` [23.2.2003].

[2] Austin, D., Barbir, A., Ferris, C., and Garg, S. Working draft on Web Services Architecture Requirements. Tech. rep., W3C, Aug. 2002. `http://www.w3.org/TR/2002/WD-wsa-reqs-20020819` [23.2.2003].

[3] Bearman, M. *Tutorial on ODP Trading Function.* DSTC, Faculty of Information Sciences and Engineering, University of Canberra, Australia, 1997. Myös `http://citeseer.nj.nec.com/bearman97tutorial.html` [23.2.2003].

[4] Belaid, D., Provenzano, N., and Taconet, C. Dynamic management of CORBA trader federation. In *4th USENIX Conference of Object-Oriented Technologies and Systems (COOTS)* (Santa Fe, New Mexico, 1998), USENIX Association, California, USA. Myös `http://www.usenix.org/publications/library/proceedings/coots98/full_papers/belaid/belaid.pdf` [23.2.2003].

[5] Craske, G., Tari, Z., and Kumar, K. R. DOK-trader: A CORBA persistent trader with query routing facilities. In *International Symposium on Distributed Objects and Applications (DOA'99)* (1999), IEEE Computer Society, pp. 230–240. Myös `http://www.cs.rmit.edu.au/~zahirt/Papers/doa99.pdf` [23.2.2003].

[6] Deschrevel, J.-P. The ANSA Model for Trading and Federation. Tech. Rep. APM.1005.01, Architecture Projects Management Limited, July 1993. Myös `http://www.ansa.co.uk/ANSATech/93/Primary/100501.pdf` [23.2.2003].

[7] Grosse, A., Keller, L., and Kottmann, D. Agora: Value-added services in trading-based systems. In *IFIP/IEEE International Conference on Distributed Platforms (ICDP'96)* (Dresden, Germany, Feb. 1996), Chapmann & Hall. Myös `http://citeseer.nj.nec.com/grosse96agora.html` [23.2.2003].

[8] Hoare, C. A. R. *Communicating Sequential Processes.* Prentice-Hall International, Englewood Cliffs, 1990.

[9] IONA TECHNOLOGIES. *ORBacus Trader, version 2.0.0*, 2001. `http://www.iona.com/products/orbacus/trader.htm` [23.2.2003].

[10] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Trading Function*, 1997. IS13235.

[11] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Type Repository Function*, 1997. IS14746.

[12] ISO/IEC JTC1. *Information Technology – Open Systems Interconnection, Data Management and Open Distributed Processing. ODP Enterprise Language*, June 2000. CD13235.

[13] KRZYSZTOF, M. S. Some performance aspects of trading service implementations. In *ACM SIGPLAN Conference On Object-Oriented Programming Systems, Languages and Applications (OOPSLA 1997)* (Atlanta, Georgia, Oct. 1997), ACM Press. Myös `http://citeseer.nj.nec.com/242961.html` [23.2.2003].

[14] KUTVONEN, L. *Trading services in open distributed environments*. PhD thesis, Department of Computer Science, University of Helsinki, 1998. Report A-1998-2.

[15] KUTVONEN, L. Automated management of interorganisational applications. In *6th IEEE International Enterprise Distributed Object Computing Conference (EDOC2002)* (2002), IEEE Computer Society, pp. 27–38.

[16] MARVIE, R., MERLE, P., GEIB, J.-M., AND LEBLANC, S. Type-safe trading proxies using TORBA. In *Fifth International Symposium on Autonomous Decentralized Systems (ISADS 2001)* (March 2001), IEEE Computer Society, pp. 303–310. Myös `http://citeseer.nj.nec.com/401104.html` [23.2.2003].

[17] MEDVIDOVIC, N., AND TAYLOR, R. N. A framework for classifying and comparing architecture description languages. In *ESEC/FSE '97* (1997), M. Jazayeri and H. Schauer, Eds., vol. 1301 of *Lecture Notes in Computer Science*, Springer / ACM Press, pp. 60–76. Myös `http://citeseer.nj.nec.com/medvidovic97framework.html` [23.2.2003].

[18] *MicoCCM-projektin web-sivut*, helmikuu 2003. `http://www.fpx.de/MicoCCM/`.

[19] OASIS CONSORTIUM. *UDDI version 3.0 published specification*, July 2002. `http://www.uddi.org/specification.html` [23.2.2003].

[20] OBJECT MANAGEMENT GROUP. *OMG Trading Object Service Specification*, June 2000. OMG Document formal/2000-06-27. Myös `http://www.omg.org/cgi-bin/doc?formal/2000-06-27` [23.2.2003].

[21] Object Management Group. *Model Driven Architecture – A Technical Perspective*, July 2001. OMG Document ormsc/01-07-01. Myös `http://www.omg.org/cgi-bin/doc?ormsc/01-07-01.pdf` [23.2.2003].

[22] Object Management Group. *Common Object Request Broker Architecture: Core Specification 3.0.2*, Dec. 2002. OMG Document formal/2002-12-06. Myös `http://www.omg.org/cgi-bin/doc?formal/02-12-06` [23.2.2003].

[23] Object Management Group. *CORBA Component Model v3.0*. Framingham, MA, USA, 2002. OMG Document formal/02-06-65. Myös `http://www.omg.org/cgi-bin/doc?formal/02-06-65` [23.2.2003].

[24] ObjectWeb Consortium. *OpenCCM-projektin web-sivut*, helmikuu 2003. `http://www.objectweb.org/openccm/`.

[25] Orponen, P. *Laskennan teoria*. Tietojenkäsittelytieteen laitos, Helsingin yliopisto, elokuu 1993. Luentomoniste, 2.painos.

[26] *PIMEA-projektin web-sivut*, kesäkuu 2002. `http://pimea.sourceforge.net`.

[27] Puder, A., and Römer, K. Generic trading service in telecommunication platforms. In *5th International Conference on Conceptual Structures (ICCS'97)* (August 1997), Springer Verlag, pp. 551–565. Myös `http://citeseer.nj.nec.com/puder97generic.html` [23.2.2003].

[28] Rafaelsen, H. O., and Eliassen, F. Trading and negotiating stream bindings. In *Proceedings of IFIP/ACM International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware 2000)* (New York, NY, USA, Apr. 2000), Springer Verlag, pp. 289–307. Myös `http://citeseer.nj.nec.com/rafaelsen00trading.html` [23.2.2003].

[29] Szyperski, C. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 1998.

[30] Terzis, S., and Nixon, P. Towards a semantically enhanced component trader architecture. Tech. rep., Computer Science Department, Trinity College Dublin, 2000. TCD-CS-2000-23. Myös `http://citeseer.nj.nec.com/terzis00towards.html` [23.2.2003].

[31] Vähäaho, M., Haataja, J.-P., Metso, J., Suoranta, T., and Kutvonen, L. Pilarcos prototype II. Tech. rep., Department of Computer Science, University of Helsinki, Jan. 2003. Draft, to be published as C-2003-NN.

[32] Vähäaho, M., Silfver, E., Haataja, J., Kutvonen, L., and Alanko, T. Pilarcos demonstration prototype – design and performance. Tech. rep., Department of Computer Science, University of Helsinki, Dec. 2001. C-2001-64.

[33] VÄHÄKANGAS, A. Meklauksen vaativuudesta. Henkilökohtainen sähköposti Antti Vähäkankaalta Markku Vähäaholle 31.3.2002.

[34] WALDO, J. The Jini architecture for network-centric computing. *Communications of the ACM 42*, 7 (July 1999), 76–82. Myös `http://citeseer.nj.nec.com/waldo99jini.html` [23.2.2003].

Liite A

Suomi-englanti -sanasto

Alla oleva sanasto perustuu englanti-suomi -ODP-sanastoon [14, liite A], yleisesti käytettyyn termistöön ja kirjoittajan omiin suomennoksiin.

aikainen sidonta	early binding
alityyppi	subtype
alusta	platform
arkkitehtuurikieli	architecture description language
arkkitehtuurikuvaus	architecture description
asiakas	client
komponentti	component
(palvelun) kysyjä	importer
käyttäjä	user
liitin	connector
meklari	trader
meklaus	trading
meklauspalvelu	trading service
muunnin	interceptor
myöhäinen sidonta	late binding
nimipalvelu	name service
näkökulma	viewpoint
olioviite	object reference
ominaisuus	property
ominaisuuskehikko	property framework
operaatio	operation
palvelin	server
palvelu	service
palvelukysely	import
palvelupyyntö	service request
palvelutarjous	service offer
palvelutyyppi	service type

perintä	inheritance
politiikka	policy
politiikkakehikko	policy framework
rajapinta	interface
rajapintatyyppi	interface type
rajapintaviite	interface reference
rajapintavarasto	interface repository
rajoitelauseke	constraint expression
rooli	role
ryhmälähetys	multicast
sidonta	binding
sidos	binding
signaali	signal
suositummuus	preference
suositummuuslauseke	preference expression
(palvelun) tarjoaja	exporter
tarjoaminen	export(ing)
tavoitenäkökulma	enterprise viewpoint
tietue	struct (C-kielessä)
toimintonäkökulma	computational viewpoint
toteutusnäkökulma	engineering viewpoint
tyyppikuvaus	type description
tyyppivarasto	type repository
viitemalli	reference model
vuo	stream
vuorovaikutus	interaction
väliohjelmisto	middleware
välitystarjous	proxy offer
yhteisö	community
ylityyppi	supertype

Liite B

CORBA-meklauspalvelun rajapintojen IDL-kuvaukset

Alla on CORBA-meklauspalvelun ja sen palvelutyyppivaraston keskeisimpien rajapintojen OMG IDL-kuvaukset [20]. Poikkeukset on selvyyden ja lyhyyden vuoksi jätetty pois kuvauksista.

```
//
// ***** Trading Service *****
//

module CosTrading {
    // ----- type definitions used in more than one interface -----

    typedef Istring PropertyName;
    typedef sequence<PropertyName> PropertyNameSeq;
    typedef any PropertyValue;
    struct Property {
            PropertyName name;
            PropertyValue value;
    };
    typedef sequence<Property> PropertySeq;

    struct Offer {
            Object reference;
            PropertySeq properties;
    };
    typedef sequence<Offer> OfferSeq;
    typedef string OfferId;
    typedef sequence<OfferId> OfferIdSeq;

    typedef Istring ServiceTypeName;  // similar structure to IR::Identifier
    typedef Istring Constraint;

    enum FollowOption {
        local_only,
        if_no_local,
        always
```

```
};

typedef string PolicyName;  // policy names restricted to Latin1
typedef sequence<PolicyName> PolicyNameSeq;
typedef any PolicyValue;
struct Policy {
    PolicyName name;
    PolicyValue value;
};
typedef sequence<Policy> PolicySeq;

// ----- interfaces -----

interface OfferIterator {
    unsigned long max_left (
    );

    boolean next_n (
        in unsigned long n,
        out OfferSeq offers
    );

    void destroy ();
};

interface Lookup : TraderComponents, SupportAttributes, ImportAttributes {
    typedef Istring Preference;

    enum HowManyProps { none, some, all };

    union SpecifiedProps switch ( HowManyProps ) {
        case some: PropertyNameSeq prop_names;
    };

    void query (
        in ServiceTypeName type,
        in Constraint constr,
        in Preference pref,
        in PolicySeq policies,
        in SpecifiedProps desired_props,
        in unsigned long how_many,
        out OfferSeq offers,
        out OfferIterator offer_itr,
        out PolicyNameSeq limits_applied
    );
};

interface Register : TraderComponents, SupportAttributes {
    struct OfferInfo {
        Object reference;
        ServiceTypeName type;
        PropertySeq properties;
    };
```

```
        OfferId export (
            in Object reference,
            in ServiceTypeName type,
            in PropertySeq properties
        );

        void withdraw (
            in OfferId id
        );

        OfferInfo describe (
            in OfferId id
        );

        void modify (
            in OfferId id,
            in PropertyNameSeq del_list,
            in PropertySeq modify_list
        );

        void withdraw_using_constraint (
            in ServiceTypeName type,
            in Constraint constr
        );

        Register resolve (
            in TraderName name
        );
    };

}; /* end module CosTrading */

//
// ***** Service Type Repository *****
//

module CosTradingRepos {
    interface ServiceTypeRepository {
        // ----- local types -----

        typedef sequence<CosTrading::ServiceTypeName> ServiceTypeNameSeq;
        enum PropertyMode {
            PROP_NORMAL, PROP_READONLY,
            PROP_MANDATORY, PROP_MANDATORY_READONLY
        };
        struct PropStruct {
            CosTrading::PropertyName name;
            CORBA::TypeCode value_type;
            PropertyMode mode;
        };
        typedef sequence<PropStruct> PropStructSeq;

        typedef CosTrading::Istring Identifier;  // IR::Identifier
        struct IncarnationNumber {
```

```
            unsigned long high;
            unsigned long low;
        };
        struct TypeStruct {
            Identifier if_name;
            PropStructSeq props;
            ServiceTypeNameSeq super_types;
            boolean masked;
            IncarnationNumber incarnation;
        };

        enum ListOption { all, since };
        union SpecifiedServiceTypes switch ( ListOption ) {
            case since: IncarnationNumber incarnation;
        };

        // ----- attributes -----

        readonly attribute IncarnationNumber incarnation;

        // ----- operation signatures -----

        IncarnationNumber add_type (
            in CosTrading::ServiceTypeName name,
            in Identifier if_name,
            in PropStructSeq props,
            in ServiceTypeNameSeq super_types
        );

        void remove_type (
            in CosTrading::ServiceTypeName name
        );

        ServiceTypeNameSeq list_types (
            in SpecifiedServiceTypes which_types
        );

        TypeStruct describe_type (
            in CosTrading::ServiceTypeName name
        );

        TypeStruct fully_describe_type (
            in CosTrading::ServiceTypeName name
        );

        void mask_type (
            in CosTrading::ServiceTypeName name
        );

        void unmask_type (
            in CosTrading::ServiceTypeName name
        );
    };
}; /* end module CosTradingRepos */
```

Liite C

Pilarcos-meklauspalvelun rajapintojen IDL-kuvaukset

Alla on Pilarcos-meklarin keskeisimpien tietorakenteiden ja rajapintojen OMG IDL-kuvaukset [31]. Poikkeukset on selvyyden ja lyhyyden vuoksi jätetty pois kuvauksista.

```
//
// ***** Common data structures *****
//

module Pilarcos {
    // ----- Policies -----

    typedef string                          PolicyTypeName;
    typedef string                          PolicyFrameworkTypeName;
    typedef string                          PolicyFrameworkName;
    typedef sequence<PolicyFrameworkName>   PolicyFrameworkNameSeq;

    typedef any                             PolicyValue;
    typedef sequence<PolicyValue>           PolicyValues;
    typedef sequence<PolicyValues>          PolicyValuesSeq;

    struct LongRange {
        long        low;
        long        high;
        long        preferred;
    };

    struct DoubleRange {
        double      low;
        double      high;
        double      preferred;
    };

    enum ConstraintType {
        some_of,
        one_of,
```

```
    exactly
};

typedef sequence<string>            StringList;
typedef sequence<double>            WeightFactors;

struct StringSet {
    StringList                      strings;
    WeightFactors                   weight_factors;
    ConstraintType                  constraint_type;
};

// ----- Application interface references -----

typedef string                      FederationContractID;
typedef string                      NameServiceRef;
typedef string                      PSIType;
typedef string                      PIIName;
typedef string                      FederatedName;
typedef string                      NamingFormat; // CORBA, JNDI, ...

typedef sequence<PSIType>           PSITypeSeq;
typedef sequence<PIIName>           PIINameSeq;

struct IndirectRef {
    string         name;
    NameServiceRef ns_loc;
};

interface FederatedNaming;
struct InterfaceRef {
    PSIType                         type;
    // name of the interface in the service type (e.g. "billing")
    PIIName                         pii_name;

    FederatedNaming                 federated_ns;
    FederatedName                   federated_name;
    boolean                         direct_naming;
};
typedef sequence<InterfaceRef>      InterfaceRefSeq;

// ----- Service contract structure -----

typedef string                      ServiceTypeName;
typedef string                      RoleName;
typedef string                      ArchitectureName;
typedef string                      ServiceContractID;

typedef sequence<ServiceTypeName>   ServiceTypeNameSeq;

struct ServiceContract {
    ServiceContractID               service_contract_id;
    ServiceTypeName                 service_type;
    PolicyValuesSeq                 policy_values;
```

```
        InterfaceRefSeq                 interfaces;
        Object                             federation_manager;
    };

    // ----- Federation contract structure -----

    struct ServiceForRole {
        RoleName                        role;
        ServiceContract                 service_contract;
    };
    typedef sequence<ServiceForRole>    ServiceForRoleSeq;

    struct FederationContract {
        FederationContractID            federation_contract_id;
        ArchitectureName                architecture;
        ServiceForRoleSeq               service_contracts;
    };
    typedef sequence<FederationContract> FederationContractSeq;

}; // module Pilarcos

//
// ***** Pilarcos trading service *****
//

module Pilarcos {
    module Trading {
        // note that ServiceOfferID != ServiceContractID
        typedef string ServiceOfferID;

        // ----- OfferRegistration interface -----

        interface OfferRegistration {
            ServiceOfferID export(
                in ServiceContract service_offer
            );
            void withdraw(
                in ServiceOfferID service_offer_id
            );
            ServiceContract describeServiceOffer(
                in ServiceOfferID service_offer_id
            );
            void emptyTrader();
        };

        // ----- Lookup interface -----

        interface Lookup {
            FederationContractSeq populate (
                in FederationContract  initial_federation_offer,
                in unsigned long    max_offers,
                in unsigned long    max_time_ms
            );
        };
```

```
        // ----- Trader component -----

        component TraderComponent {
            provides OfferRegistration               offer_registration;
            provides Lookup                          lookup;
            uses     TypeRepository::TypeRegistration type_registration;
            uses     TypeRepository::TypeMatching     type_matching;
            uses     CosTrading::TraderComponents     corba_trader;

        };
        home TraderHome manages TraderComponent {
        };

    }; // module Trading
}; // module Pilarcos

//
// ***** Type repository *****
//

module Pilarcos {
    module TypeRepository {
        // ----- Interface types -----

        typedef string InterfaceType;
        typedef string InterfaceTypeName;

        // ----- Policies -----

        struct PolicyType {
            PolicyTypeName  name;
            CORBA::TypeCode type;
        };
        typedef sequence<PolicyType> PolicyTypeSeq;

        struct PolicyFrameworkType {
            PolicyFrameworkTypeName name;
            PolicyTypeSeq           policy_types;
        };

        struct PolicyFramework {
            PolicyFrameworkName     name;
            PolicyFrameworkTypeName policy_framework_type;
        };
        typedef sequence<PolicyFramework> PolicyFrameworkSeq;

        // ----- Service type -----

        typedef string ServiceInterfaceTypeName;

        enum InterfaceRole {
            provider,
            user
```

```
};

struct ServiceInterfaceType {
    InterfaceRole            interface_role;
    ServiceInterfaceTypeName name;
    InterfaceType            interface_type;
    PolicyFrameworkNameSeq   service_policy_framework_names;
};
typedef sequence<ServiceInterfaceType> ServiceInterfaceTypeSeq;

struct ServiceType {
    ServiceTypeName         name;
    PolicyFrameworkSeq      policy_frameworks;
    ServiceInterfaceTypeSeq service_interface_types;
 };

// ----- Architecture -----

typedef string RoleName;
typedef string ArchitectureName;

struct Role {
    RoleName        name;
    ServiceTypeName service_type;
};
typedef sequence<Role> RoleSeq;

struct Binding {
    RoleName                role_1;
    ServiceInterfaceTypeName role_1_interface;
    RoleName                role_2;
    ServiceInterfaceTypeName role_2_interface;
};
typedef sequence<Binding> BindingSeq;

struct Architecture {
    ArchitectureName name;
    RoleSeq          roles;
    BindingSeq       bindings;
};

// ----- Architecture w/ policy framework graphs -----

// These are used only between type repository and trader.

struct RolePolicyFramework {
    long architecture_role_index;     // index of role in architecture
    long role_policy_framework_index; // index of p.framework in role
};
typedef sequence<RolePolicyFramework> RolePolicyFrameworkSeq;

struct PolicyFrameworkGraph {
    PolicyFrameworkType    policy_framework_type;
    RolePolicyFrameworkSeq policy_frameworks_in_roles;
```

```
};
typedef sequence<PolicyFrameworkGraph> PolicyFrameworkGraphSeq;

// index = index of policy framework in role (service type)
typedef sequence<long> PolicyFrameworkGraphIndexSeq;

// index = index of role in architecture
typedef sequence<PolicyFrameworkGraphIndexSeq> RoleIndexSeq;

struct ArchitectureWithGraphs {
    Architecture            architecture;
    PolicyFrameworkGraphSeq policy_framework_graphs;
    RoleIndexSeq            architecture_role_index;
};

// ----- TypeRegistration interface -----

interface TypeRegistration {
    void registerPolicyFrameworkType (
        in PolicyFrameworkType policy_framework_type
     );
    void registerServiceType (
        in ServiceType service_type
    );
    void registerArchitecture (
        in Architecture architecture_description
    );
    void emptyTypeRepository ();
    PolicyFrameWorkType describePolicyFrameworkType (
        in PolicyFrameworkTypeName name
    );
    ServiceType describeServiceType (
        in serviceTypeName name
    );
    Architecture describeArchitecture (
        in ArchitectureName name
    );
    ArchitectureWithGraphs describeArchitectureWithGraphs (
        in ArchitectureName name
    );
};

// ----- TypeMatching interface -----

interface TypeMatching {
    boolean isCompatibleInterface (
        in PSIType first_end,
        in PSIType second_end
    );
};

// ----- Type repository component -----

component TypeRepositoryComponent {
```

```
            provides TypeRegistration type_registration;
            provides TypeMatching type_matching;
            uses CosTrading::TraderComponents corba_trader;
        };
        home TypeRepositoryHome manages TypeRepositoryComponent {
        };

    }; // module TypeRepository
}; // module Pilarcos
```