Väliohjelmistojen hyödyntäminen adaptiivisissa mobiilisovelluksissa

Jouni Ojanen

Helsinki 23.11.2003

Reflektiiviset väliohjelmistot -seminaari

HELSINGIN YLIOPISTO

Tietojenkäsittelytieteen laitos

Sisältö

1. Johdanto

Mobiiliympäristöille on ominaista jatkuva dynaamisuus. Hajautetut järjestelmät ovat luonteeltaan varsin heterogeenisiä. Jo tarkasteltaessa suhteellisen pientä maantieteellistä aluetta havaitaan, että verkon eri osissa käytetään useita erilaisia teknologioita ja protokollia. Lisäksi aliverkkojen topologiat eroavat toisistaan huomattavasti. Mobiilikäyttäjien päätelaitteet ovat niinikään ominaisuuksiltaan varsin vaihtelevia unohtamatta itse käyttäjiäkään, joiden vaatimukset palvelujen sisällön, laadun yms. suhteen voivat olla hyvinkin yksityiskohtaisia. Mobiilisovellusten kehittäjille nämä seikat ovat haasteellisia. Sovellusten tulisi selvitä ympäristön muutoksista mahdollisimman tuntumattomasti ja dynaamisesti. Paras vaihtoehto olisi, että sovellus pystyisi täysin ajonaikaisesti sopeutumaan muutoksiin. Tällainen sovellus on **sopeutuva l. adaptiivinen** . Katz [Ka94] määrittelee mobiilisovellusten adaptiivisuustarpeen ytimekkäästi seuraavasti:

“Mobility requires adaptability. By this we mean that systems must be location- and situation-aware, and must take advantage of this information to dynamically configure themselves in a distributed fashion.”

Tämä esitys koostuu seuraavista osista. Ensin käsitellään adaptoitumistarpeita mobiiliympäristöissä yleisesti kappaleessa 2. Kappaleessa 3 siirrytään tarkastelemaan eri tasoja, joilla adaptaatio voidaan toteuttaa. Adaptaatio voidaan toteuttaa periaatteessa joko palvelin- tai asiakaspäässä ja edelleen ohjelmiston eri tasoilla. Seuraavaksi tarkastellaan muutamia yleisiä väliohjelmistoratkaisuja, jotka mahdollistavat sovellusten adaptaation kappaleessa 4. On kuitenkin tärkeää havaita, että nämä esitettävät ratkaisumallit ovat sovellettavissa periaatteessa mihin sovellusalueeseen tahansa. Mobiiliympäristössä malleista saadaan vain huomattavan suuri hyöty.

2. Adaptoituvuustarpeet mobiiliympäristössä

Seuraavassa käsitellään joitain tekijöitä, jotka aikaansaavat mobiilisovellusten adaptoitumistarpeen [BlCo00]. Lista ei ole mitenkään kaikenkattava, mutta antaa käsityksen siitä, miksi adaptiivisuus on mobiiliympäristössä tärkeää.

2.1. Käyttäjätiedot

Mobiilisovellusten asiakaskunnat vaihtelevat suuresti. Joissain tapauksissa sovellus on räätälöity ainoastaan tietyn yrityksen sisäiseen käyttöön. Tällaisia ovat esim. sovellukset, jotka ottavat yhteyden yrityksen intranet:iin. Toisaalta sovellus voi olla täysin maailmanlaajuisestikin tavoitettavissa. Ellei kyse ole täysin avoimesta ja ilmaisesta internet-palvelusta, tarvitaan yleensä ainakin jonkinlaista käyttäjän tunnistamista ja **autentikointia**. Tarvittavan autentikoinnin taso voi edelleen vaihdella paljon. Autentikoinnin taso voidaan määritellä esimerkiksi koko sovelluksen tasolla, jolloin autentikoitu käyttäjä saa oikeuden kaikkiin sovelluksen toimintoihin. Monimutkaisemmissa tapauksissa oikeuksia rajoitetaan myös yksittäisten toimintojen osalta.

Käyttäjillä on erilaisia sovelluksen **käyttötarpeita**. Tarpeet voivat liittyä esimerkiksi sovelluksen käsittelemän datan määrään. Jotkin käyttäjäryhmät voivat tarvita enemmän dataa kuin toiset. Hyvä esimerkki on sovellus, jolla on sekä tavallisia että viranomaiskäyttäjiä. Viranomaiskäyttäjällä on luonnollisesti oltava mahdollisuus hakea enemmän tietoja kuin tavallisella käyttäjällä esimerkiksi päätösten tekoa varten. Viranomainen voi myös tarvita sellaisia toimintoja, joita tavallinen käyttäjä ei ollenkaan tarvitse.

On myös mahdollista, että koko sovelluksen logiikan täytyy mukautua käyttäjän tarpeiden mukaan. Monien sovellusten täytyy esimerkiksi tukea sekä online- että offline-tyyppistä työskentelyä esim. käyttäjän kulloisenkin työtilanteen mukaan. Jos käyttäjä vaikkapa kiertää asiakkaalta toiselle ja kirjaa kustakin asiakkaasta muistiinpanoja, ei välttämättä yksinkertaisesti ole aikaa ottaa joka kerta yhteyttä verkkoon, vaan tietojen synkronointi täytyy tehdä yhdessä isommassa sessiossa työpäivän päätteeksi.

Tarpeet nivoutuvat usein varsin tiukasti käyttöoikeuksiin. Niinpä kunkin tarpeen täyttyminen edellyttää usein tiettyä käyttöoikeusprofiilia.

Jokaisella käyttäjällä on edelleen oma vakiintunut tapansa työskennellä tietokonesovellusten parissa. Niinpä on luontevaa, että myös mobiilisovellus mukautuu käyttäjän mieltymysten mukaiseksi. Tässä on kuitenkin oltava erittäin varovainen, sillä “automaattinen” adaptointi eli käyttäjän mieltymysten olettaminen ei tässä kovinkaan usein toimi. Parhainkaan sovellus ei pysty olemaan ajatustenlukija. Niinpä voikin olla viisainta jättää mieltymysten mukaisten ohjelma-asetusten tekeminen käyttäjälähtöiseksi.

2.2. Laitteiston ominaisuudet

PDA-laitteiden **prosessorit** ovat kehittyneet huomattavasti viime vuosina. Tämän tutkielman kirjoitushetkellä parhaiten varusteltujen PocketPC - PDA-laitteiden prosessorikellotaajuudet olivat jo 400 Mhz. Tämä vastaa suunnilleen pöytäkoneiden prosessoritehoja noin neljä vuotta sitten. Prosessoritehosta ei oikeastaan enää voidakaan puhua varsinaisena suorituskykypullonkaulana. Laskentaintensiivisiin tehtäviin PDA-laite ei yleensäkään liene kaikkein luontevin laitealusta.
Muistin määrä rajoittaa vielä toistaiseksi huomattavasti enemmän kämmenmikrojen käyttömahdollisuuksia kuin prosessoriteho. Kämmenmikroissa käytettävissä oleva muisti jakaantuu **ROM**- ja **RAM**-muisteihin. ROM-muisti on luonteeltaan pysyvä eli sen sisältö säilyy myös silloin, jos kämmenmikrolle tehdään ns. hard-reset. Tämä tarkoittaa sitä , että laite palautetaan tehdasasetuksiinsa. ROM-muisti sisältää tyypillisesti ainakin käyttöjärjestelmän sekä esimerkiksi mahdolliset väliohjelmistot, esim. PocketPC-laitteen ollessa kyseessä .NET Compact Framework-ohjelmiston. ROM-muistia on uusimmissa PocketPC-laitteissa yleensä 32 megatavua. RAM-muisti sisältää kämmenmikrossa käytettävät ohjelmat. Se on väliaikaismuisti ja sen sisältö tuhoutuu hard-reset:in yhteydessä. Uusimmissa laitteissa RAM-muistia on yleensä jo vähintään 64 megatavua. Jokseenkin kaikissa laitteissa muistia voi myös lisätä erillisillä muistikorteilla. Näin RAM-muistin maksimimäärä voi olla jopa 256 megatavua.

Käytännössä sovellusten käytössä olevan RAM-muistin määrä on jonkin verran pienempi. Käyttöjärjestelmä vie aina osan muistista niin että esimerkiksi 64 megatavun muistista jää sovellusten käyttöön noin 56 megatavua. Tämähän on tilanne toki myös PC-koneissa, mutta merkittävä ero on se, että kämmenmikroissa ei ole kiintolevyjä lainkaan. Kaikki data, jonka sovellus tallentaa persistentisti, tallentuu RAM-muistiin.

Näytön koko on todennäköisesti kaikkein rajoittavin tekijä kämmenmikrosovelluksia kehitettäessä. Siinä missä pöytäkoneiden näytöt suurenevat kaiken aikaa ollen jo keskimäärin vähintään 17-tuumaisia, kämmenmikrojen näyttöjen koot ovat pysyneet suunnilleen samoina tai ehkä jopa pienentyneet laitteiden koon pienentyessä. Näyttöjen kokoahan on hyvin hankalaa nykytekniikoilla kasvattaa ilman laitteen koon kasvattamista.

Näytön pieni koko voi pakottaa kehittämään kokonaan oman **käyttöliittymämallin** (UI-model) kämmenmikrosovelluksia varten.

Akkukesto on erittäin tärkeä profiilitekijä, joka tulisi ottaa aina huomioon kämmenmikrosovellusten suunnittelussa. Akun käyttöaika on luonnollisesti kääntäen verrannollinen virrankulutukseen. PDA-laitteiden

virrankulutusta on tutkittu melko paljon aiheen tärkeyden vuoksi, esim. [Fli01]. Virrankulutus riippuu monesta tekijästä. Jotkin ovat puhtaasti laitteistoarkkitehtuurikohtaisia eivätkä siis kovin helposti sovelluskehittäjän hallittavissa. Esimerkiksi eri prosessorimallien virrankulutuksessa on havaittu huomattaviakin eroja. Toisiin tekijöihin taas voidaan vaikuttaa sovelluksen arkkitehtuurisuunnittelulla. Yksi tällainen tekijä on sovelluksen yhteydellisen toiminnallisuuden määrä. Langattomat lähiverkkoyhteydet (WLAN) ovat tällä hetkellä vasta tekemässä tuloaan PDA-laitteisiin ja toteutuksissa tuntuu olevan vielä parantamisen varaa ainakin virrankulutuksen osalta. Paljon WLAN-yhteyksiä käytettäessä akun käyttöaika saattaa vähentyä jopa kymmeniä prosentteja. Käyttöaikaa voidaankin selkeästi parantaa minimoimalla ensinnäkin aika, joka ollaan yhteydessä verkkoon sekä toisaalta siirrettävän tiedon määrä.

2.3. Ympäristö

Mobiilisovelluksen tulee monissa tapauksissa olla tietoinen myös ympäristöstä, missä se kulloinkin toimii. Seuraavassa käsitellään lyhyesti joitain kyseeseen tulevia ympäristötekijöitä.

Monien mobiilisovellusten toimintalogiikka riippuu käyttäjän kulloisestakin **sijainnista**. Tarvittava tarkkuus voi edelleen vaihdella sovelluksen tarpeista riippuen. Jos sovellus tarjoaa maakohtaisesti lokalisoituja palveluja, voi riittää tietää pelkästään valtio, jonka alueella mobiililaite on. Mikäli sovelluksen tarkoituksena on esimerkiksi tarjota ajo-ohjeita kaupungissa, on sijainti tiedettävä jo ainakin kadun nimen tarkkuudella. Jopa sijainnin tietäminen rakennuksen huoneen tarkkuudella voi tulla kyseeseen. Paitsi käyttäjän myös itse ohjelmiston sijainti tai paremminkin konteksti voi edellyttää adaptoituvuutta. Tilanne voi olla esimerkiksi se, että sovellusta haluttaisiin ajaa sekä asiakas- että palvelinympäristöissä ja näin välttää mahdollisimman paljon “tuplakoodia”.

Verkkoyhteys ja sen laatu on yksi tärkeimmistä ympäristöprofiilitekijöistä. Yhteyden merkitys on luonnollisesti sitä suurempi mitä enemmän sovelluksen logiikka on online-tyyppistä eli siis jatkuvan verkkoyhteyden vaativaa. Jotta palvelun laatu pystyttäisiin pitämään vaaditulla tasolla, sovellus (ja laite) joutuvat mukautumaan muuttuviin olosuhteisiin useilla tietoliikenneprotokollan tasoilla. Esimerkiksi verkkokerrostasolla (network layer) tapahtuva muutos on mobiililaitteen siirtyminen verkon solusta toiseen. Siirtoyhteystasolla (transport layer) voidaan joutua vaihtamaan esim. TCP/IP-protokollan parametreja eli pistokeoptioita. On myös mahdollista, että joudutaan vaihtamaan yhteydellinen protokolla yhteydettömään tai päinvastoin. Tilannehan saattaa olla esimerkiksi sellainen, että haluttaisiin preferoida UDP-protokollaa TCP/IP:n asemesta, mutta epäluotettavan verkon takia joudutaan ottamaan TCP/IP käyttöön.

Mukautumista voidaan tarvita myös sovellustasolla. Jos yhteyden laatu huononee, voi olla hyödyllistä ottaa

käyttöön vaikkapa tiedon pakkaualgoritmi tai tehokkaampi algoritmi, jos jokin on jo ollut käytössä. Joskus sovelluslogiikkaa voidaan joutua mukauttamaan vieläkin ylemmällä tasolla, esim. ottamalla käyttöön paikallisia välimuisteja (cache) tms.

3. Toteutusarkkitehtuuri

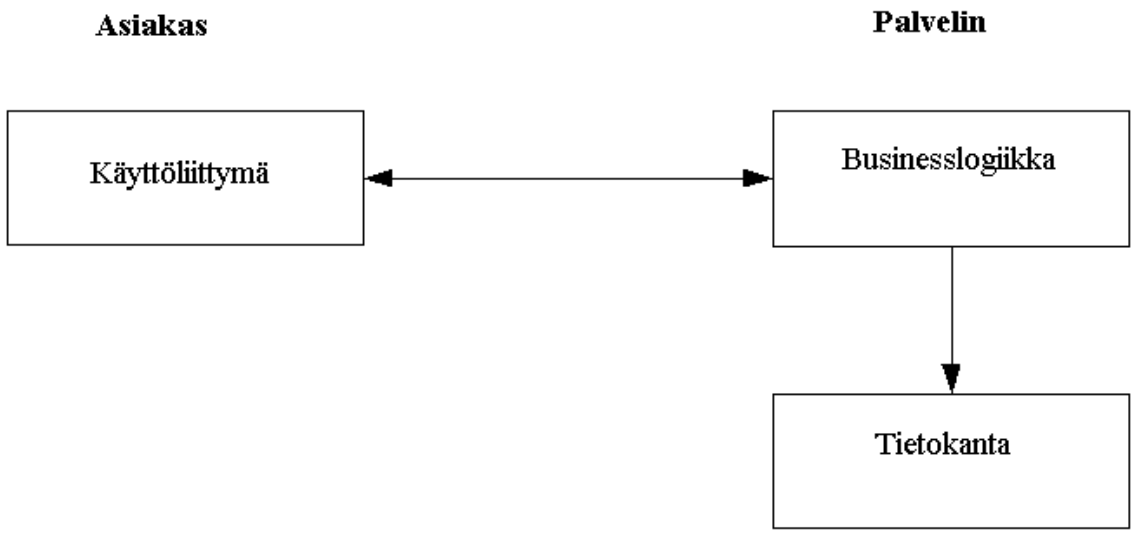
Adaptoituvuusvaatimukset on otettava huomioon jo mobiilisovelluksen arkkitehtuuria suunniteltaessa. Adaptoituvuus voidaan toteuttaa monessa paikassa ja edelleen monella eri tasolla. Adaptointi voidaan tehdä ensiksikin joko asiakas- tai palvelinpäässä. Tähän liittyviä seikkoja käsitellään kappaleessa 3.1. Toiseksi on tehtävä päätös siitä, millä sovelluksen tasolla adaptointi toteutetaan. Tätä käsitellään lyhyesti kappaleessa 3.2.

3.1. Perusarkkitehtuurivaihtoehdot

Mobiilisovellusta rakennettaessa on tehtävä heti selkeä työnjako asiakkaan eli päätelaitteen ja palvelimen välillä. Käytännössä on valittavissa kaksi peruslähestymistapaa, joilla kummallakin on omat etunsa ja haittansa [JHE99]. Näitä lähestymistapoja käsitellään seuraavassa.

3.1.1. Thin client

Thin client –arkkitehtuuriratkaisussa (kuva 1) lähes koko sovelluksen toimintalogiikka on sijoitettu palvelimelle. Mobiililaite käyttää sovellusta selaimen kautta. Sovellus on tällöin useimmiten luonteeltaan tilaton ja sen käyttöä voisi luonnehtia sessiopohjaiseksi. Thin Client -ratkaisulla saavutetaan useita hyötyjä. Koska varsinainen sovellus sijaitsee palvelimella, sovelluksen ylläpito ei vaadi mitään muutoksia asiakkaisiin, vaan ainoastaan palvelinkomponenttien päivityksen. Erityisesti PC-sovelluksia kehittävät yritykset ymmärtävät, kuinka paljon kustannuksia ja työtä asiakasohjelmien lähetyksestä, asennuksesta ja tukipalveluista aiheutuu. Toiseksi, koska Thin Client -sovellus toimii selaimen kautta, sovellus tukee periaatteessa kaikkia laitteita, joissa on asianmukainen selain asennettuna. Laitteistoheterogeenisyys hoituu siis erittäin hyvin.



Kuva 1. Thin Client

Ratkaisussa on kuitenkin myös merkittäviä puutteita. Merkittävin lienee nimenomaan tilallisen datakäsittelyn puute. Tästä seuraa, että sovellusta käytettäessä laitteen olisi oltava jatkuvasti online-tilassa, siis yhteydessä verkkoon. Tämä on monesti ylitsepääsemätön ongelma, koska ensinnäkin sovellus voi olla perusluonteeltaan offline-tyyppinen eli siis suurin osa työstä tehdään offline-tilassa ja toiseksi internet-yhteyttä ei välttämättä voida muodostaa ollenkaan joka paikassa. Sovellusten käyttöliittymä on myös luotava selaimen rajoitusten mukaan. Tämä voi johtaa kankeasti käytettävään sovellukseen, jos näyttöjä on paljon.

Thin Client -lähestymistapa soveltuu parhaiten yksinkertaisiin, lähinnä tietoja hakeviin sovelluksiin. Esimerkki tällaisesta sovelluksesta voisi olla vaikkapa tuotteen saatavuustiedot myyntimiehelle noutava sovellus. Myös yksittäisiä tietoja lähettävät sovellukset voivat olla lähestymistapaan soveltuvia.

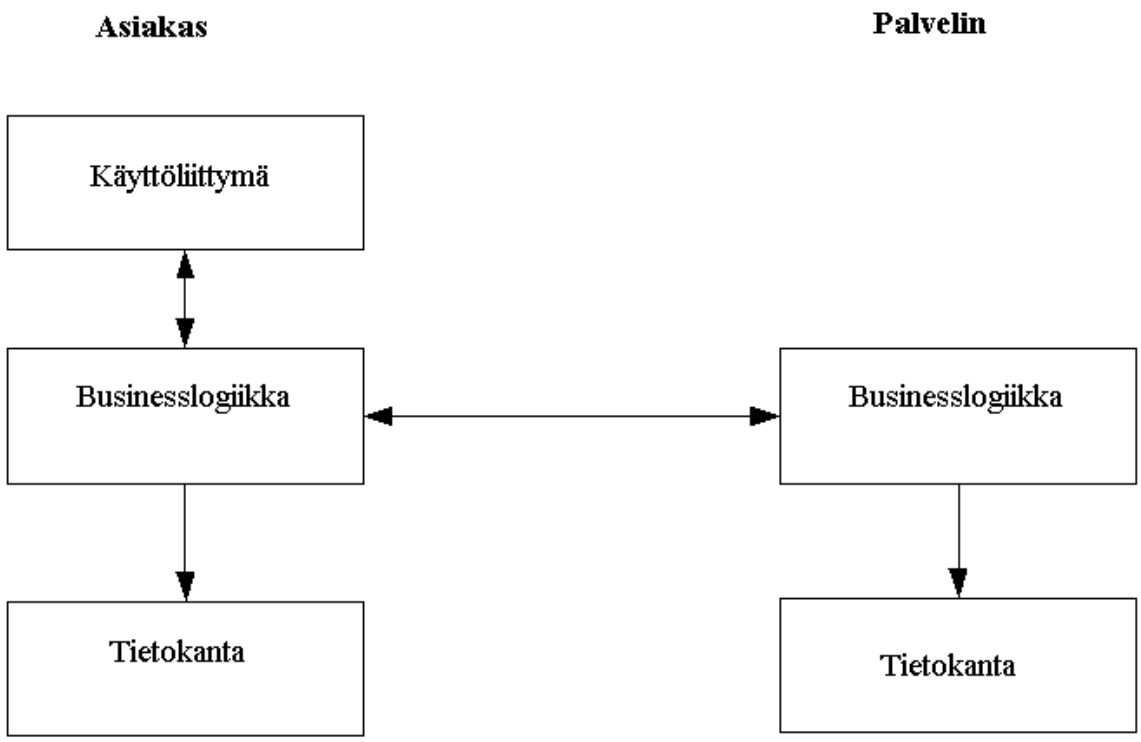
Mobiilisovellusten adaptoituvuuteen liittyvän tutkimuksen trendi on ollut selvästi painottunut selainpohjaisiin thin client -ratkaisuihin. Erityisesti voidaan mainita tässä yhteydessä **yhdistetyt kyvykkyys- ja asetusprofiilit,** engl. **Composite Capability / Preference Profiles** [WWW00], joita käytetään välittämään mobiililaitteen tekniset tiedot sekä mahdolliset käyttäjän personointitiedot palvelimelle. Palvelinsovellus voi adaptoida tuottamansa sisällön profiilien mukaisiksi. Tässä esityksessä asiaa ei tarkastella lähemmin.

3.1.2. Rich client

Rich Client -lähestymistavassa (kuva 2) ainakin osa businesslogiikkaa (BL) sekä mahdollisesti myös osa datakerroksen toiminnoista on siirretty palvelimelta asiakassovellukseen. Asiakassovellusta voidaan käyttää periaatteessa hyvin samalla tavalla kuin tilallista työpöytäsovellusta. Poikkeuksena on luonnollisesti mm. näppäimistön ja hiiren käyttö, jotka PDA-laitteissa hoidetaan tyypillisesti tarkoitukseen kehitetyn kynän avulla. Lähestymistavan selvimpänä hyötynä on nimenomaan joustavuus ja monipuolisuus. Käyttöliittymä voidaan suunnitella mahdollisimman käyttäjäystävälliseksi ja sovellus voi tukea monenlaisia käyttöskenaarioita ml. ne, jotka myös thin client -ratkaisulla saavutettaisiin.

Haittoina ovat monimutkaistuva asiakassovellusten päivitys sekä heterogeenisyyden hallinta. Koska sovelluslogiikkaa ajetaan mobiililaitteessa, laitteen ominaisuudet vaikuttavat sovelluksen toimintaan aivan eri tavalla kuin selainsovelluksissa. Ennenkuin Rich Client -ratkaisua voidaan harkita, on varmistuttava, että käyttäjillä on riitävän suorituskykyinen laitekanta.

Adaptaatio voidaan rich client -mallissa toteuttaa joko asiakas- tai palvelinpäässä tai molemmissa. Esimerkiksi käyttöliittymän personointi käyttäjäprofiilin mukaan on luonnollisesti asiakassovelluksen adaptointia, kun taas tiedonhakulogiikkaan liittyvä adaptointi lienee parhainta toteuttaa palvelimella.

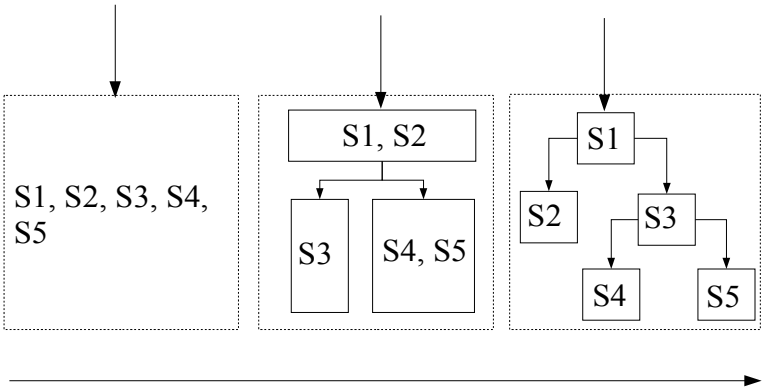


Kuva 2. Rich client

3.2. Sovelluksen jako komponentteihin

Nykyaikainen sovellus on lähes aina enemmän tai vähemmän komponenttipohjainen. Komponenttien käyttö ohjelmistokehityksessä parantaa ohjelmistojen joustavuutta, laajennettavuutta sekä ohjelmiston osien uusiokäyttömahdollisuuksia. Nämä seikat ovat erittäin tärkeitä myös kehitettäessä adaptoituvia mobiilisovelluksia. Seuraavassa käsitellään erilaisia seikkoja, jotka tällaisten sovellusten suunnittelussa täytyisi ottaa huomioon.

Sovellusta voidaan jakaa komponentteihin useilla eri tavoilla. Kuvassa 3 on esitetty muutamia mahdollisia komponentisoinnin asteita. Komponentisointiaste suurenee kuvassa vasemmalta oikealle.



Kuva 3. Sovelluksen komponentisointi

Mikäli sovellus olisi täysin monoliittinen, esimerkiksi yksi ajettava binääri, se täytyisi siirtää jokaiseen tarvittavaan ympäristöön samanlaisena. Tämä yksinkertaistaisi huomattavasti erityisesti sovellusten lataus- ja asennuslogiikkaa, mutta aiheuttaisi myös useita ongelmia. Ensiksikin kyseinen sovellus voisi olla kooltaan aivan liian suuri ympäristöihin, joissa muistitilaa on vähän. Toiseksi mukaan tulisi ehkä runsaasti ohjelmaosia, joita joko ei ollenkaan tarvita tai ikinä voida käyttää resursseiltaan rajoittuneissa ympäristöissä. Kuvassa 3 vasemmanpuoleinen kuva esittää monoliittista sovellusta, joka sisältää palvelut S1,S2,S3,S4 sekä S5.

Toinen äärivaihtoehto olisi jakaa koko sovellus komponentteihin siten, että itse ajettava (tai käynnistävä) binääri on pelkkä kokoelma komponenttikutsuja. Binääri voidaan mieltää tässä tapauksessa eräänlaiseksi proxy:ksi, joka siis itsessään sisältää hyvin vähän toimintalogiikkaa kaiken logiikan sijaitessa komponenteissa. Tämä ratkaisuvaihtoehto tarjoaa kaikkein parhaimman joustavuuden. Jos komponentit ovat hyvin suunniteltuja, eri sovellusversioiden toteuttaminen on periaatteessa vain sopivien komponenttien valitsemista ja yhdistelyä. Esimerkiksi sovellusversioiden tilantarvetta voidaan säädellä optimaalisesti. Haittana on mm. sovelluksen yleisen suorituskyvyn heikentyminen. Komponenttien luominen ja niiden väliset kutsut ovat johtavat ehkä jonkinverran hitaampaan suoritukseen kuin vastaavassa monoliittisessa sovelluksessa. Lisäksi ratkaisumalli edellyttää erittäin vahvaa ohjelmistoarkkitehtuuriosaamista. Huonot suunnitteluratkaisut kostautuvat mm. hankalana ja kalliina ylläpitona.

Kaikkein ihanteellisinta sovelluksen kannalta olisi luonnollisesti se, että adaptaatio olisi sovelluskerrokselle täysin tuntumatonta. Tämä edellyttää nimenomaan reflektiivisen väliohjelmiston hyödyntämistä. Itse väliohjelmistonkin komponentisointi on erittäin oleellista, mikäli mobiililaitteessa halutaan väliohjelmistoa hyödyntää. Tähän palataan myöhemmin kappaleessa 4.

4. Esimerkkiarkkitehtuureja

Seuraavassa käsitellään ensin muutamia esimerkkejä väliohjelmistoratkaisuista, jotka tukevat reflektiivisyyttä kaikkein alimmalla tasolla. Tämä tarkoittaa siis sitä, että väliohjelmiston tasolla voidaan dynaamisesti konfiguroida järjestelmän ydinosia kuten säikeistys- ja kommunikointistrategioita ilman oleellisia muutoksia sovellustasoon. Käsiteltävät järjestelmät eivät UIC:ta lukuunottamatta ole erityisesti mobiililaitteisiin suunniteltuja, mutta antavat tärkeitä suuntaviivoja adaptiivisten mobiilisovellusten ja -sovelluskerrosten suunnitteluun. Lopuksi käydään läpi lyhyesti mahdollisia ratkaisumalleja Microsoftin .NET-alustalla, joka ei itsessään varsinaisesti ole dynaamisesti konfiguroitava väliohjelmisto.

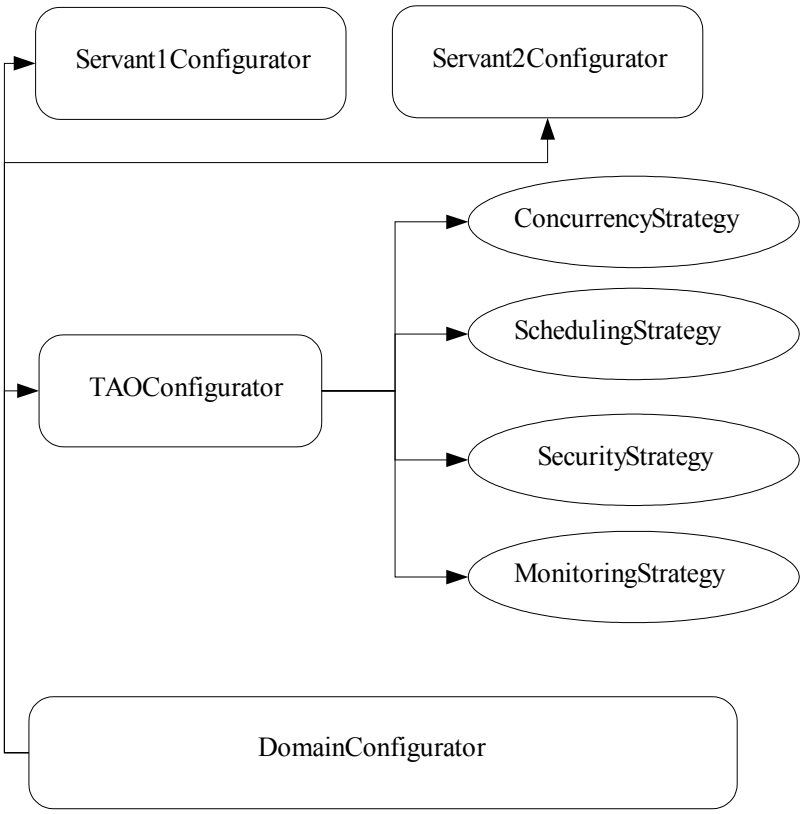
4.1. DynamicTAO

DynamicTAO [RKC01, Ko00a] on reflektiivinen CORBA ORB (Object Request Broker). Se tarjoaa mahdollisuuden sisäisen rakenteensa tarkasteluun (inspection) sekä muuttamiseen (reconfiguration). DynamicTAO sisältää rajapinnat, joilla komponentteja voi siirtää hajautetun järjestelmän osien välillä sekä ladata ja poistaa muistista kulloisenkin tilanteen mukaan. DynamicTAO:a voidaan pitää edeltäjänsä TAO:n laajennuksena. TAO kehitettiin alunperin reaaliaikaisten sovellusten ajoympäristöksi, joten se oli luonteeltaan varsin staattinen. Sen ajonaikaiset strategiat olivat kyllä konfiguroitavissa, mutta eivät ajonaikaisesti. DynamicTAO mahdollistaa myös ajonaikaisen konfiguroinnin, mikä tekee sitä käsitteellisesti reflektiivisen ajoympäristön.

4.1.1. Perusrakenne

Ajonaikainen konfigurointi tapahtuu nk. *komponenttikonfiguroijien* (component configurators) avulla [Ko00b]. Yksittäinen komponenttikonfiguroija ylläpitää riippuvuussuhdetta kyseisen komponentin ja muiden komponenttien välillä. Jokainen prosessi, joka käyttää DynamicTAO ORB:tä, sisältää DomainConfigurator-konfiguroijan. Tämän lisäksi kuhunkin ORB-ilmentymään liittyy yksi TAOConfigurator-konfiguroija, jonka

tehtävä on tarjota liittymäkohta järjestelmän ydintoimintojen vaihtoehtoisiin toteutusstrategioihin. Malli on esitetty kuvassa 4.



Kuva 4. DynamicTAO

DynamicTAO:n konfigurointi tehdään DynamicConfigurator- objektin rajapintaa käyttäen. Rajapinta sisältää monentyyppisiä palveluja. **Kyselypalvelujen** avulla voidaan kysyä esimerkiksi, mitä järjestelmätason palveluja voidaan konfiguroida. Konfiguroitavia palveluja nimitetään DynamicTAO:ssa kategorioiksi (category). **Lataupalvelujen** avulla puolestaan voidaan ladata haluttuja toteutusstrategioita käyttöön ja liittää (hook) niitä haluttuihin komponentteihin. Strategioita voidaan edelleen myös muokata tai poistaa käytöstä ajonaikaisesti. Tällainen strategioiden dynaaminen lataaminen ja poistaminen ei toki onnistu ilman ongelmia. Strategian vaihto ei välttämättä onnistu noin vain, mikäli aktiivisia järjestelmän asiakkaita palvellaan edelleen vanhaa strategiaa käyttäen samalla kun uutta ollaan lataamassa. Esimerkkinä tällaisesta tilanteesta on säikeistysstrategian vaihto “säieallas” -strategiasta (thread pool) tavalliseen strategiaan. Nyt “allas-strategian” suorittavaa koodia ei voida

ladata suoraan pois järjestelmästä, koska ko. strategiaa käyttävät pyynnöt vaativat pääsyn siihen aina yhteyden päättyessä. Tässä tapauksessa allasstrategiasta vastaavan konfiguroijan täytyy pitää kirjaa aktiivisista säikeistä ja tuhota ne sitä mukaa kuin niihin liitetyt yhteydet (client connections) päättyvät. Kun yhtään aktiivista säiettä ei ole, konfiguroija generoi tapahtuman, joka kertoo sitä vastaavan strategian olevan valmis poistettavaksi.

4.1.2 Monitorointi

DynamicTAO mahdollistaa erittäin dynaamisen tavan palvelustrategioiden ajonaikaiseen vaihtamiseen. Oleellisena ratkaistavana ongelmana on kuitenkin myös sen havaitseminen, **koska** strategiaa pitäisi vaihtaa. DynamicTAO:ssa asia on hoidettu erillisellä monitoripalvelulla, joka kykenee keräämään tietoa komponenttien välisestä vuorovaikutuksesta. Tämä tieto voidaan välittää sovelluksille, jotka kykenevät adaptoitumaan tilanteen mukaan. Myös järjestelmävalvojille monitoripalvelun tarjoama tieto voi olla hyödyllistä, koska se auttaa havaitsemaan mm. mahdollisia järjestelmän pullonkauloja.

Monitoripalvelun keskeisenä osana on **väliprosessoija** (interceptor), joka kerää tiedot CORBA ORB:hen kohdistuvista komponenttikutsuista. Palvelua voidaan konfiguroida DynamicConfigurator-konfiguroijan avulla samaan tapaan kuin strategioita vaihdettaessa. Palvelu kirjaa keräämänsä tiedot **tallennuspalvelimelle** (storage server), josta niitä voidaan kysellä.

4.2 Universaalisti yhteistoiminnallinen ydin

DynamicTAO:a tarkasteltaessa on helppo havaita, että se ei sellaisenaan vielä sovellu kovinkaan hyvin mobiililaitteeseen asennettavaksi väliohjelmistoksi. Syynä on yksinkertaisesti ohjelmiston yleinen raskaus ja koko. Vaikka DynamicTAO:n toiminnallisuus on itsessään erittäin hyvin konfiguroitavissa, se on kuitenkin siinä mielessä staattinen, että se sisältää kaiken toiminnallisuuden yhdessä ohjelmakirjastossa. Mobiililaitteiden näkökulmasta katsottuna tarvittaisiin mikroydinpohjainen ratkaisu, joka sallisi ottaa mukaan vain ja ainoastaan ne ohjelmiston osat, joita tarvitaan ja joihin resurssit riittävät. **Universaalisti yhteistoiminnallinen ydin**, engl. **Universally Interoperable Core (UIC)** [RKC01] on DynamicTAO:sta edelleen kehitetty versio, jossa tätä lähestymistapaa on käytetty. Huomautettakoon, että UIC tunnettiin itse asiassa aikaisemmin nimellä LegORB [Ro00]. UIC sisältää vain kaikkein oleellisimmat ydinpalvelut antaen ohjelmoijalle muuten täyden vapauden valita tarvittavat lisäkomponentit ja mahdollisesti ohjelmoida lisätoiminnallisuutta niitä hyödyntäen. Suunnitteluperiaatteena on ollut WYNIWYG eli suomeksi “mitä tarvitset sen saat”.

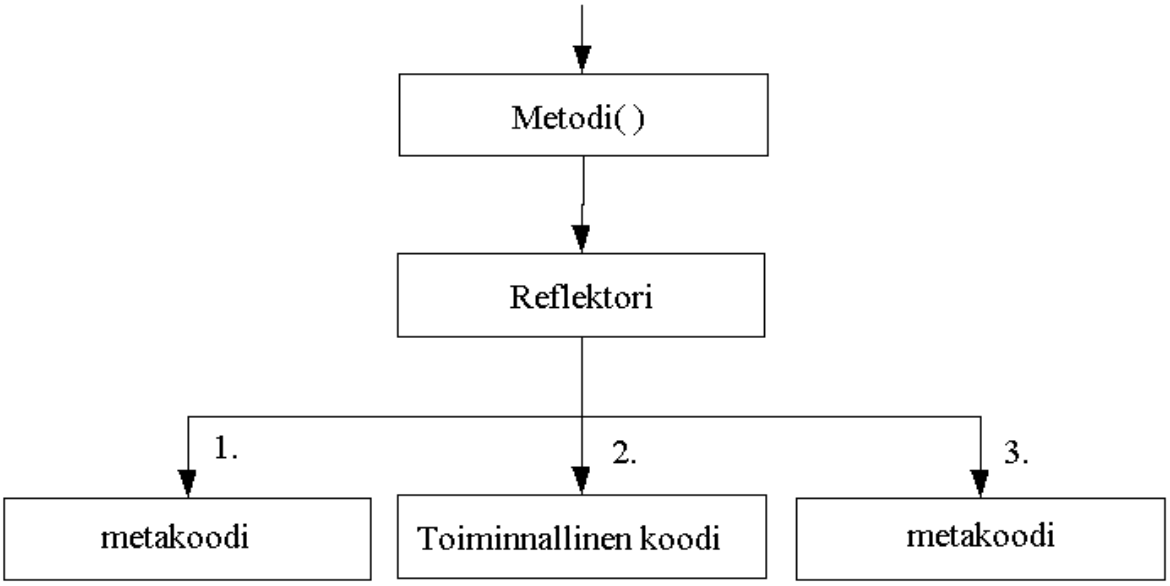
UIC on pohjimmiltaan abstrakti olio ja se täytyy **erikoistaa** (specialize) ennen käyttöönottoa. Tämä tarkoittaa sitä, että on määriteltävä, minkä väliohjelmistojen kanssa se kykenee kommunikoimaan. Erikoistetusta UIC-oliosta käytetään nimitystä **persoona** (personality). Persoona voi olla joko single- tai multi-tyyppinen. Single-tyyppinen persoona kykenee kommunikoimaan ainoastaan yhden väliohjelmiston kanssa, kun taas multi-tyyppisellä persoonalla ei tätä rajoitusta ole. Persoona voi edelleen olla joko **asiakaspersoona** (client-side) , **palvelinpersoona** (server-side) tai se voi edustaa molempia rooleja. Näin voidaan määritellä tarkasti, mitä rooleja laitteen täytyy edustaa.

UIC-persoonat voidaan konfiguroida joko staattisesti, dynaamisesti tai hybridisesti. Staattisessa konfiguroinnissa persoonat käännetään staattiseksi kirjastoksi, jota ei ajonaikaisesti enää voi konfiguroida. Dynaamisessa konfiguroinnissa kirjastot ovat dynaamisesti ladattavia ja konfiguroitavia. Staattisen konfiguraation merkittävin hyöty on persoonan pieni koko, kun taas dynaaminen konfiguraatio on tietenkin joustavampi. Hybridikonfiguraatio pyrkii yhdistämään staattisen ja dynaamisen asennuksen hyvät puolet antamalla vapauden valita, mitä asennetaan staattisesti ja mitä dynaamisesti.

4.3. Distributed Adaptive Runtime

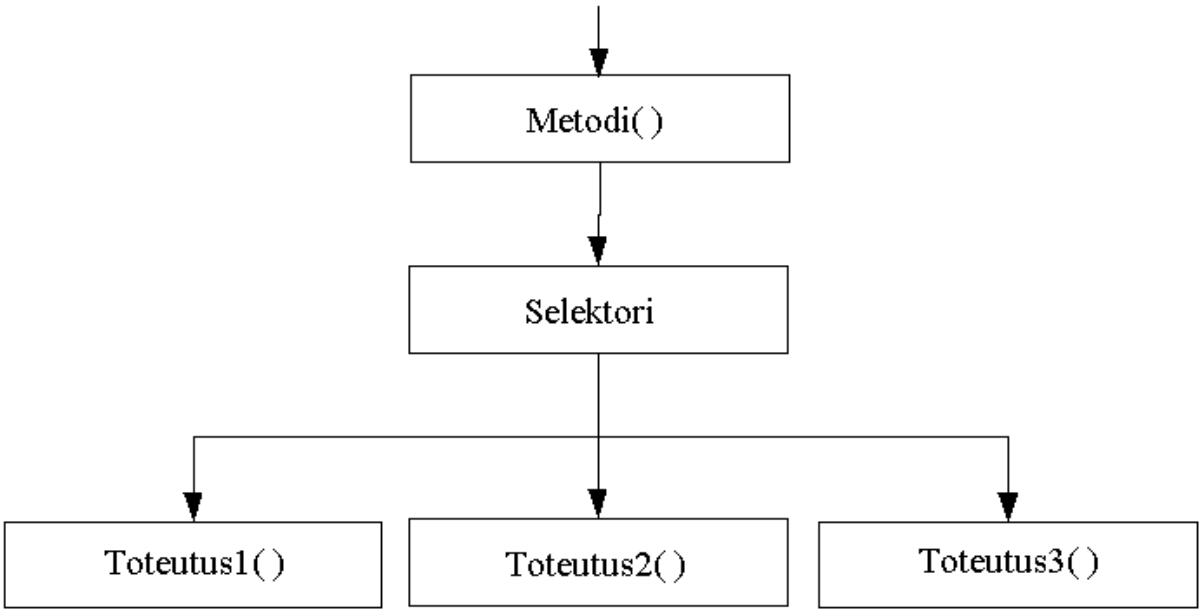
Distributed Adaptive Runtime (**DART**) [RaLe98] on toinen esimerkki reflektiivisestä väliohjelmistosta. DART tekee selvän eron sovelluksen sisäisen sekä järjestelmätason adaptiivisuuden välillä. DART:ssa on joitain yhtäläisyyksiä myös DynamicTAO:n kanssa. Tässä DART esitellään kuitenkin erityisesti siitä syystä, että se toimii eräänlaisena siltana Microsoft .NET -ratkaisulle.

Luokkien metodit voidaan määritellä joko reflektiivisiksi tai adaptiivisiksi, luonnollisesti metodin ei tarvitse kuulua kumpaankaan luokkaan. Reflektiiviset metodit mahdollistavat sovelluksen ajoympäristön dynaamisen uudelleenkonfiguroinnin. DART määrittelee joukon metaobjekteja, jotka kuvaavat yleisiä järjestelmätason palveluja sovelluksista riipumatta. Kuhunkin reflektiiviseen metodiin liittyy yksi tai useampi tällainen metaobjekti. Reflektiivisen metodin kutsu uudelleenohjautuu siihen liitetyille metaobjekteille. Metaobjektien metodeja ei kuitenkaan kutsuta suoraan vaan erityisten **reflektorien** (reflectors) kautta. Reflektori määrittelee funktiot, jotka suoritetaan ennen ja jälkeen reflektiivisen metodin funktionaalisen koodin suoritusta. Nämä funktiot antavat siis mahdollisuuden ympäristön adaptoimiseksi. Kuvassa 5 on esimerkki reflektiivisen metodin kutsusta.



Kuva 5. Reflektiivinen metodikutsu

Adaptiivisia metodeja käytetään sovelluksen toiminnallisen koodin ajonaikaiseen valintaan. Ideana on se, että luokan suunnitteluvaiheessa määritellään kutakin mahdollista strategiaa varten joukko vaihtoehtoisia metodeja. Nämä metodit merkitään adaptiivisiksi ja ajonaikana niihin kohdistuvat kutsut uudelleenohjautuvat oikealle metodille **selektorin** (selector) kautta. Tämä on esitetty kuvassa 6.



Kuva 6. Adaptiivisen metodin kutsu

Samalla kun DART-sovellus käynnistyy, luodaan erityinen *DARTManager*-objekti. Se sisältää kaiken sovelluksen adaptiivisuuteen liittyvän tiedon. Adaptiivisia ja/tai reflektiivisiä metodeja sisältävät luokat rekisteröivät itsensä DARTManager-objektille. DARTManager- objektia vastaa DynamicTAO:ssa käsitteellisesti DomainConfigurator-objekti.

4.4 Microsoft .NET

Seuraavassa käsitellään adaptiivisuuden ratkaisumalleja Microsoftin uutta .NET Framework -ohjelmistoa [Mic00] hyödyntävissä mobiilisovelluksissa.

4.4.1 Reflektio .NET Framework:ssa

.NET Framework sisältää suhteellisen helppokäyttöisen reflektiokirjaston. **System.Reflection** -nimiavaruudessa on määritelty joukko kirjastoluokkia, joita käyttäen voidaan kysellä esim. mitä luokkia ajettava binääri (assembly) sisältää ja edelleen mitä metodeja luokat sisältävät. Myös arkkitehtuurisia seikkoja voidaan kysellä, kuten esimerkiksi onko luokka jonkin toisen luokan aliluokka tai kenties abstrakti luokka.

Mielenkiintoinen piirre .NET :ssä on mahdollisuus laajentaa ohjelmistokomponentin metadataa nk. **laajennusattribuuteilla** (custom attributes). Metodiin tai koko luokkaan voidaan liittää kuvaavia attribuutteja, joita väliohjelmisto itse sekä mahdolliset työkaluohjelmistot voivat hyödyntää. Esimerkiksi C# -metodimäärittelyssä

```
[WebMethod(TransactionOption=TransactionOption.RequiresNew)]
public int ProcessOrder(DataSet dsOrder)
{
}
```

WebMethod-laajennusattribuutti kertoo .NET -ajoympäristölle (Common Language Runtime l. CLR), että kyseessä on ns. **Web Services** -metodi. Web Services [WWW03] on .NET -alustan keskeinen hajautusmalli ja erityisesti PocketPC-mobiililaitteet kommunikoivat palvelinten kanssa nimenomaan

tätä mallia käyttäen. WebMethod-attribuutti saa esimerkissä vielä parametrinaan TransactionOption.RequiresNew -arvon, joka kertoo edelleen ajoympäristölle, että metodi aloittaa uuden transaktion.

Laajennusattribuutit ovat erittäin tervetullut lisä Microsoft-teknologioihin. Haittapuolina on kuitenkin ensinnäkin se, että attribuutin muuttaminen vaatii aina uuden käännöksen ja toisaalta koodin selkeys helposti kärsii.

4.4.2 .NET Compact Framework

.NET Compact Framework, jatkossa .NET CF, on kämmentietokoneita varten luotu "miniversio" .NET Framework:sta. Täydessä .NET Framework:ssa on osia, joita ei resursseiltaan rajoittuneissa laitteissa voi ajaa ja siksi tällaisen karsitun version tekeminen on ollut välttämätöntä.

Siinä missä esimerkiksi UIC antoi ohjelmoijille mahdollisuuden täydelliseen väliohjelmistokonfiguraation dynaamiseen valintaan, Microsoftin ratkaisu ongelmaan on luonteeltaan staattisempi. .NET CF -väliohjelmiston ydinkomponenttien konfiguraatio on aina vakio. Ainoastaan käytettävien luokkakirjastojen määrää voi kontrolloida. Ratkaisun ilmeisenä etuna on kaikkien .NET CF:n sisältämien laitteiden yhtenevä konfiguraatio. Kaikilla ohjelmoijilla on myös yhtenevä käsitys siitä, mitkä väliohjelmiston peruspalvelut ovat käytettävissä ja mitkä eivät. Tasokkaimmat laitteet luonnollisesti kärsivät tästä ajattelumallista. Ne voisivat ehkä hyödyntää hyvinkin joitain puuttuvia palveluja, mutta joutuvat odottamaan, kunnes kaikki laitteet edistyvät sille tasolle, että uusi sellainen versio .NET CF:stä voidaan julkaista, joka sisältää ko. palvelut. .NET CF -väliohjelmisto ei siis itsessään istu kovinkaan hyvin adaptiiviseen suunnitteluun, vaan adaptiivisuuden toteutus jää pitkälti sovellustasolle.

4.4.3 Esimerkkiratkaisu

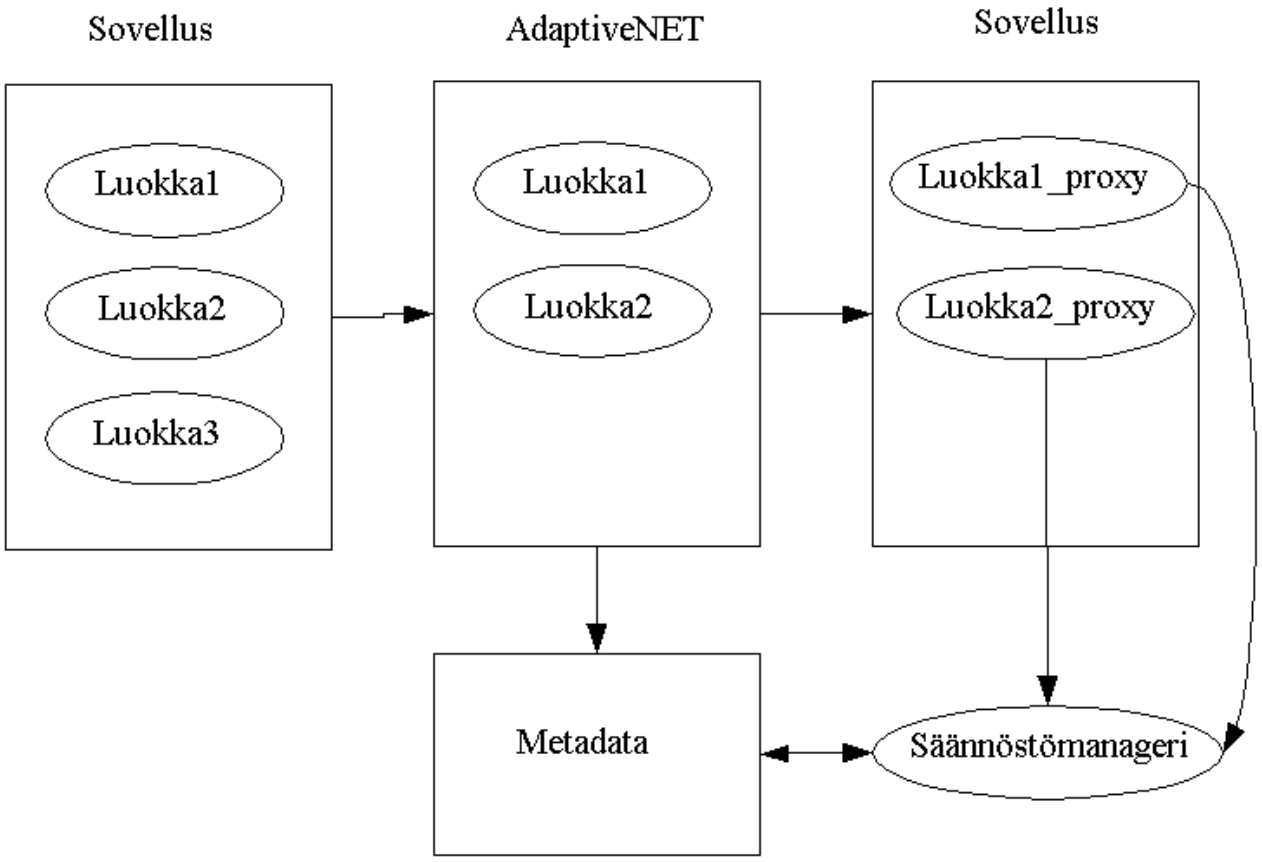
Vaikka itse .NET CF -alusta ei tarjoakaan mainittavaa adaptointitukea, voidaan .NET CF:n reflektio-ominaisuuksien avulla kuitenkin toteuttaa kohtuullisella vaivalla sovellustason adaptaatio. Seuraavassa käsitellään hyvin lyhyesti yksi mahdollinen lähestymistapa. Menetelmä liittyy kirjoittajan omaan työhön ja pohjautuu etäisesti DART:n metamalliin

Menetelmän lähtökohtana on se, että joko sovelluksen sisältämiä luokkia tai niiden yksittäisiä metodeja voidaan

merkitä adaptiivisiksi. Tätä merkitsemistä ei kuitenkaan varsinaisesti tehdä vielä sovellusta ohjelmoidessa, vaan ideana on generoida valmiista sovelluksesta adaptiivinen versio erillistä **AdaptiveNET**-työkalua käyttäen. Haluttaessa tietystä luokasta adaptiivinen sille tehdään kaksi tai useampia vaihtoehtoisia toteutuksia l. strategioita, jotka periytyvät yhteisestä (abstraktista) kantaluokasta. Jos taas halutaan yksittäisestä metodista adaptiivinen, luokkaan määritellään kaksi tai useampia loogisesti saman asian tekeviä metodeja, joiden nimeäminen voi olla vapaata.

Ajonaikaisesti valittavat adaptiiviset luokka- ja metodiversiot valitaan erityistä **säännöstömanageri**-objektia (policy manager) käyttäen. Säännöstömanageri käsittelee sovelluskohtaista metadataa, jonka avulla oikea strategia voidaan valita. Itse metadata on yksi XML-dokumentti, joka määrittelee reunaehdot kunkin strategian käytölle.

Työkalu generoi **proxy**-luokat ohjelmoijan valitsemista adaptiivisista luokista sekä luokista, jotka sisältävät adaptiivisia metodeja. Proxy-luokat vastaavat DART:in selektoreja ja reflektoreja. Ne lisäävät luokan alkuperäiseen koodiin kutsut säännöstömanageriobjektiin strategian selvittämiseksi. Työkalu voidaan toteuttaa suhteellisen yksinkertaisesti System.Reflection-luokkien avulla. Työkalu sisältää graafisen käyttöliittymän, jossa valitun komponentin rajapintakuvausta voidaan tarkastella ja siitä valita adaptiiviset metodit. Valituille metodeille voidaan sitten edelleen määritellä reunaehdot työkalua käyttäen. Kuvassa 7 on esitetty koko prosessi.



Kuva 7. AdaptiveNET-työkalu

Koska metadata on toteutustavaltaan XML-dokumentti, on sen ajonaikainen muokkaaminen yksinkertaista. Muokkaaminen voidaan tehdä joko DOM-dokumenttia suoraan päivittämällä tai sitten korvaamalla XML-tiedosto, josta dokumentti sitten uudelleen ladataan. Edellinen tapa soveltuu ajoympäristössä tapahtuvien profiilimuutosten kirjaamiseen. Jälkimmäistä tapaa voidaan käyttää esimerkiksi mobiililaitteessa olevan sovelluksen etäkonfigurointiin palvelimelta käsin.

Seuraavassa on vielä yksinkertainen esimerkki XML-dokumentista. Dokumentti määrittelee reunaehdot sovelluksen Validate-luokan adaptiivisten (vaihdannaisten) metodien ValidateComplete( ) ja ValidatePartial( ) ajonaikaiseen valintaan.

```
<AdaptationPolicy Class="Validate">
    <AdaptiveMethodPolicy MethodName="ValidateComplete">
        <Environment type="Server"/>
    </AdaptiveMethodPolicy>
    <AdaptiveMethodPolicy MethodName="ValidatePartial">
        <Environment type="MobileClient"/>
    </AdaptiveMethodPolicy>
</AdaptationPolicy>
```

Jos siis komponenttia ajetaan palvelinympäristössä, valitaan ValidateComplete-metodi (täydellinen validointi). Jos taas ajoympäristönä on mobiililaite, valitaan ValidatePartial-metodi (osittainen validointi). Tässä tapauksessa adaptointitarve voi aiheutua esim. siitä, että mobiilisovelluksessa ei ole riittävästi dataa, jotta täydellinen validointi voitaisiin tehdä.

Lähteet

Ka94 Katz, R.H. ,Adaptation and Mobility in Wireless Information Systems, IEEE Personal Communications, Vol. 1, No. 1, pp. 6-17, First Quarter 1994.

BlCo00 G.S. Blair et al, A Principled Approach to Supporting Adaptation In Distributed Mobile Environments, International Symposium on Software Engineering for Parallel and Distributed Systems, Limerick, Ireland, 2000.

Fli01 J.Flinn, Extending Mobile Computer Battery Life through Energy-Aware Adaptation, PhD Thesis, Carnegie Mellon University, December 2001.

JHE99 J. Jing, A. Helal, A. Elmagarmid, Client-Server Computing in Mobile Environments, ACM Computing Surveys, Vol. 31, No. 2, June 1999.

WWW00 Composite Capability / Preference Profiles (CC/PP) , http://www.w3.org/Mobile/CCPP

RKC01 M. Roman, F. Kon and R. Campbell, Reflective Middleware: From your Desk to your Hand, IEEE Communications Surveys, 2(5), 2001.

Ko00a F. Kon et al. , Monitoring, Security and Dynamic Configuration with the DynamicTAO reflective ORB, proc. IFIP/ACM Int'l Conf. Distibuted Systems Platforms and Open Distributed Processing (Middleware2000), Springer-Verlag, New York, 2000.

Ko00b F.Kon, Automatic Configuration of Component-Based Distributed Systems, PHD Thesis, Department of Computer Science, University of Illinois at Urbana-Champaign, May 2000.

Ro00 M. Roman, D. Mickunas, F. Kon, R.H. Campbell, LegORB and Ubiquitous CORBA, Workshop on Reflective Middleware 2000, Palisades, NY, April 2000.

RaLe98 P.-G. Raverdy, R.Lea, DART: A Distributed Adaptive Run-Time. Proceedings of the IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware'98), September 1998

Mic00 Microsoft .NET , http://www.msdn.microsoft.com/netframework

WWW03 Web Services Architecture, W3C Working Draft 8 August 2003, http://www.w3.org/TR/2003/WD-ws-arch-20030808/