Mobiilin MIDP-sovelluksen toteutus ja suorituskyky XML-pohjaisilla yhteyskäytännöillä

Heikki Holopainen

6.5.2005

Joensuun yliopisto
Tietojenkäsittelytiede
Pro gradu -tutkielma

TIIVISTELMÄ

Ohjelmistojen kehitys mobiililaitteille tuo mukanaan uusia haasteita pienen näytön ja vähäisen käytettävissä olevan muistin määrän takia. Javan *J2ME*-kehitysympäristö on tarkoitettu mobiililaitteilla toimiville sovelluksille ja tarjoaa sovelluskehittäjille käyttöjärjestelmä- ja alustariippumattoman kehitystyökalun. Laitteiden minimikokoonpanon määrittelee *CLDC*-määritys, jonka piiriin kuuluvia laitteita ovat esimerkiksi matkapuhelimet. Mobiilisovelluksille on ominaista, että ne käyttävät omien ominaisuuksiensa lisäksi myös verkosta saatavilla olevaa tietoa. Tällöin järjestelmien keskinäinen kommunikointi edellyttää sovellusten välistä yhteistoimintaa hajautetussa ympäristössä, johon *XML*-pohjaiset yhteyskäytännöt tarjoavat laitteisto- ja ohjelmistoriippumattoman tiedonsiirtomenetelmän.

Tässä tutkielmassa arvioidaan empiirisesti esimerkkisovelluksen avulla *XML*-pohjaisia yhteyskäytäntöjä tiedonsiirron suorituskyvyn osalta ja ohjelmointipanosta koodirivien lukumäärän ja ohjelmoijalta vaadittavan osaamisen perusteella. Lisäksi arvioidaan paikallisen *RMS*-tietuevaraston suorituskykyä tiedon tallennuksen osalta eri suurilla tietomäärillä. Tutkielman tulokset osoittavat, että *XML*-pohjaisten viestien koot ovat moninkertaisia verrattuna *HTTP*-yhteyskäytännön mukaisiin binaarimuotoisiin viesteihin. *XML*-pohjaisista yhteyskäytännöistä *SOAP*-viestissä siirrettävä kuorma on suurin, yli kaksinkertainen verrattuna *XML-RPC* -viestiin. Tutkielmassa saatujen tulosten mukaan *XML-RPC* on varsin kilpailukykyinen yhteyskäytäntö suorituskyvyn ja ohjelmointipanoksen osalta verrattaessa sitä *HTTP*-yhteyskäytäntöön. *RMS*-tietuevaraston suorituskyky tiedon tallennuksen osalta riippuu lineaarisesti tallennettavan tiedon määrästä. Suorituskyky on tietysti myös riippuvainen päätelaitteen prosessorin suorituskyvystä.

ACM-luokat: (ACM Compuing Classsification System, 1998 version): C.2.1, C.2.4, C.2.5, C.4, D.2.3, D.2.6
Avainsanat: J2ME, CLDC, HTTP, XML, SOAP, XML-RPC, mobiili

SISÄLTÖ

1 JOHDANTO

Matkapuhelimiin tarvitaan yhä enemmän sovelluksia, jotka laajentavat laitteen perusominaisuuksia. Nykyaikaisille mobiililaitteille on ominaista laajennettavuus ja käyttäjän tarpeisiin mukautuminen. Käyttäjille pyritään tarjoamaan mahdollisuus täydentää mobiililaitteen toiminnallisuutta siten, että järjestelmä palvelee jokaista käyttäjää yksilöllisemmin. Tekniseltä kannalta kyse on useimmiten uuden sovelluksen asentamisesta, samaan tapaan kuin perinteiseen työasemaan jokainen voi asentaa uusia ohjelmistoja. Mobiililaitteiden nykyiset kasvunäkymät merkitsevät valtavaa kasvupotentiaalia niissä käytetyille teknologioille. Ihmiset käyttävät yhä enemmän matkapuhelimia, langatonta teknologiaa rakennetaan autoihin, langattomia tiedonsiirtopalveluja on saatavana entistä useammilla alueilla ja langattomia lähiverkkoja käytetään yhä useammissa paikoissa.

1.1 Mobiililaitteiden ominaisuuksia

Internet on muuttumassa mobiiliksi ja langattomaksi samalla tavalla kuin puhelinliikenne muuttui aikanaan. Tätä muutosta ajaa tarve päästä käsiksi kaikkeen informaatioon mistä ja milloin tahansa, jolloin lähtökohtana tarvittavien palveluiden toteuttamiseen on langattomuus. Schillerin (2001) mukaan mobiililaitteella tarkoitetaan kaikkia sellaisia laitteita, joiden avulla voidaan olla liikkeessä ja langattomassa yhteydessä tietoverkkoihin tai muihin mobiiliviestinnän käyttäjiin.

Mobiililaitteet ja langattomuus asettavat kuitenkin uusia vaatimuksia niin verkko- kuin sovellusteknologioillekin. Nykyisten verkkojen kapasiteetti ja kaistanleveys eivät riitä tulevaisuuden multimediapalveluiden käyttämiseen. Toisaalta myöskään Internetiä ei ole alun perin suunniteltu vastaamaan verkossa liikkuvien laitteiden asettamia vaatimuksia. Näiden ongelmia ratkovien uusien standardien ja tekniikoiden kehitys on kuitenkin vielä kesken. Uusien palveluiden käyttöönotossa on käyttäjien kannalta tärkeää palveluiden nopeus, toimivuus ja turvallisuus. Niinpä palveluiden suunnittelussa on kiinnitettävä erityistä huomiota langattomuuden ja mobiiliuden aiheuttamiin erityis-

piirteisiin. Yhteyden katkeamiset ja kaistanleveyden muutokset sekä hyvinkin erilaiset palveluiden käyttölaitteet vaativat palveluilta mukautumista vallitseviin olosuhteisiin. Erittäin kriittiseksi asiaksi nousee myös korkean tietoturvan säilyttäminen kaikkialla, otettiin yhteys mistä tahansa verkosta mistä päin maailmaa tahansa (Schiller, 2001).

1.2 Mobiiliohjelmien toteutus Java-kielellä

Mobiililaitteiden ohjelmistot toteutetaan useimmiten mobiiliohjelmistoalustojen avulla. Näiden ohjelmien tuottaminen kaikille saatavana oleville laitteille on huomattavasti helpompaa, jos ohjelmat ovat laitteiden kesken yhteensopivia. *Java*-ohjelmat tehdään prosessorista riippumattomalla ohjelmointikielellä. Täten *Java* tarjoaa helpon siirrettävyyden laitteesta toiseen ja on siksi varsin luonnollinen valinta myös matkaviestinten ohjelmointikieleksi. Vuonna 1999 *Sun* julkisti *J2ME* (Java 2, Micro Edition) -alustan, jolla voidaan toteuttaa mobiileihin ja muihin sulautettuihin laitteisiin soveltuvia langattomasti siirrettäviä ohjelmistoja (Muchow, 2002).

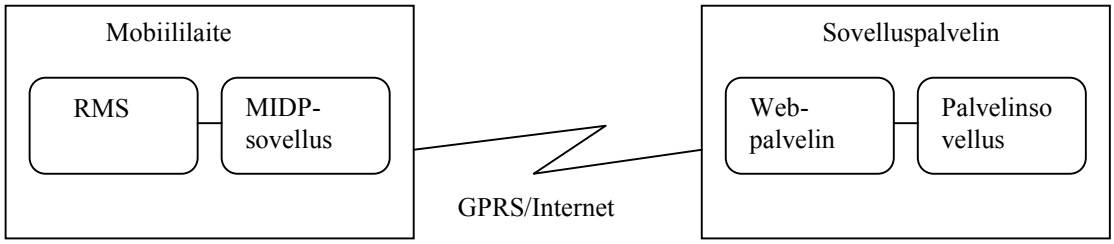
J2ME:n kehityksen tavoitteena oli kehittää pieneen tilaan mahtuva virtuaalikone ja määritellä tarkat rajat erilaisten kämmenlaitteiden *Java*-toteutuksille. Työn tuloksena syntyi *KVM* (Kilobyte Virtual Machine), jonka muistintarve on vain 128 kilotavua. Sitä käytetään monien *J2ME*-toteutusten pohjana. Lisäksi pöytätietokoneissa käytettävien kehitysympäristöjen luokkien lukumäärä on liian suuri käytettäväksi matkapuhelimissa. Tämän vuoksi *Sun* ja laitevalmistajat määrittelivät tarkat rajat erilaisten kämmenlaitteiden *Java*-toteutuksille. Näin syntyi kaksi erilaista *J2ME*-lajia, nimeltään *CDC* ja *CLDC*. Konfiguraatio *CLDC* (Connected, Limited Decice Configuration) määrittelee erityisesti matkapuhelimien ja kämmentietokoneiden toteutuksien rajat. Näille laitteille on ominaista, että niiden käyttömuistin koko on tyypillisesti 128-512 kilotavua, näyttö on kooltaan varsin pieni ja tehonlähteenä käytetään akkua (White ja Hemphill, 2002).

Konfiguraatio ja *profiili* yhdessä muodostavat *J2ME*-ympäristön. Konfiguraatio määrittelee systeemitason ominaisuuden, jotka ovat periaatteessa piilotettu ohjelmoijalta. *Profiili* on määritys, jossa kuvataan yksityiskohtaisesti konfiguraation päälle rakentuva *Java*-rajapinta, joka tarjoaa tuen tietyntyyppiselle laitteelle. Tietylle profiilille tehdyt oh-

jelmat toimivat jokaisessa saman profiilin tarjoavassa laitteessa ilman lisävaatimuksia. Profiilit laajentavat konfiguraatiota ja niiden avulla voidaan hyödyntää laitteiden erityisominaisuuksia. Erikoisesti matkapuhelimia varten konfiguraatiolle *CLDC* on määritelty profiili *MIDP* (Mobile Information Defice Profile). *MIDP*-määritys antaa laite- ja ohjelmistovalmistajille varsin vapaat kädet tehdä minimivaatimukset täyttävä toteutus. Vähintään pitää löytyä verkkoyhteys, tietueille perustuva paikallinen tallennusjärjestelmä sekä *lcdui*-määritystä noudattava käyttöliittymä (White ja Hemphill, 2002).

1.3 Mobiilisovellus osana laajempaa järjestelmää

Mobiilisovellusten toteutukseen liittyy paikallisen tallennusjärjestelmän lisäksi olennaisena osana verkkoyhteyden kautta tapahtuva palvelinten kapasiteetin hyödyntäminen. Matkapuhelinten liittäminen laajempaan järjestelmään voidaan *MIDP*-profiilia tukevissa laitteissa toteuttaa *TCP/IP*-yhteyskäytännön päälle *HTTP*-yhteyden avulla.



Kuva 1.1: *J2ME*-arkkitehtuuri mobiiliympäristössä.

Kuvassa 1.1 kuvataan *J2ME*-arkkitehtuuri mobiiliympäristössä. Mobiililaitteilla on yleensä jokin pysyvä muisti, johon tietoa voidaan tallentaa. *J2ME*-ympäristössä on käytettävissä paikallinen tietueille perustuva tallennusjärjestelmä eli tietuevarasto (Record Management System, RMS), johon *MIDP*-sovellukset voivat tallentaa tietoa pysyvään muotoon. *MIDP*-sovellukset voivat kommunikoida verkossa olevien palvelinsovellusten kanssa käyttäen *GCF*-rajapintaa (Generic Connection Framework). *MIDP*-määrityksen mukaan kaikkien *MIDP*-sovellusten on pystyttävä kommunikoimaan verkossa olevien

laitteiden kanssa vähintään *HTTP*-yhteyttä käyttäen. Chu & al. (2004) kuvaavat mobiililaitteen langattoman ja työasemassa käytetyn kiinteän verkkoyhteyden erot seuraavalla kolmella attribuutilla.

Ennustettavuus: Mobiililaitteilla on käytettävissä kapeakaistainen verkkoyhteys, jonka toiminnan ennustettavuus ei ole samalla tasolla kuin pöytäkoneissa käytettävällä kiinteäkaistaisella langallisella verkkoyhteydellä. Langattomissa verkkoyhteyksissä voi esiintyä verkkoyhteyksien katkeamisia ja vaihtelua kaistanleveydessä. Langattomat tietoliikenneyhteydet eivät ole käytettävissä joka paikassa. Maaston muodot, rakennukset ja muut vastaavat esteet haittaavat verkon käyttöä. Signaalin puuttuminen tai haalistuminen voi aiheuttaa vaihtelua kaistanleveydessä. Kaistanleveyden vaihtelun johdosta vasteaikojen ennustettavuus huononee, joka on epämiellyttävää sovelluksen käyttäjän näkökulmasta.

Resurssit: Mobiililaitteissa on usein vain rajoitettu määrä muistia ja prosessorina käytetään suhteellisen heikkotehoisia prosessoreita verrattuna työasemapäätelaitteisiin. Lisäksi mobiililaitteiden energian lähteenä käytetään akkua. Energian kulutuksen tulee olla mahdollisimman pieni, jotta energian lähde voisi olla mahdollisimman pieni ja sen käyttöikä olisi riittävän pitkä.

Saatavuus: Mobiililaitteet ovat tyypillisesti kulutuslaitteita. Tämä tarkoittaa sitä, että mobiilisovelluksilla on erilaisia vaatimuksia kuin työasemasovelluksilla. Yksi mobiilisovelluksille tyypillinen ominaisuus on nopea saatavuus. Tämä tarkoittaa sitä, että mobiilisovelluksen täytyy käynnistyksen jälkeen olla mahdollisimman nopeasti toimintakunnossa.

Mobiilisovelluksille on ominaista, että ne käyttävät omien ominaisuuksiensa lisäksi myös verkosta saatavilla olevaa tietoa. Eri koneilla sijaitsevien sovellusten yhteistoimintaa varten tarvitaan tietoliikenneyhteyksien lisäksi menetelmiä sovellusten välisen vuorovaikutuksen toteuttamiseksi. *MIDP*-pohjaisissa mobiilisovelluksissa verkkoyhteyksien käyttö perustuu useimmiten *HTTP*-yhteyskäytäntöön. *MIDP*-määrityksen mukaan kaikkien *MIDP*-toteutusten on pystyttävä kommunikoimaan verkossa olevien laitteiden kanssa vähintään *HTTP*-yhteyttä käyttäen (White ja Hemphill, 2002).

Järjestelmien keskinäinen kommunikointi edellyttää tietojärjestelmien välistä yhteistoimintaa hajautetussa ympäristössä. Merkittävä uusi tekniikka, jolla tulee olemaan tärkeä rooli tietojärjestelmien yhteistoiminnallisuuden edistämisessä on *XML* (eXtensible Markup Language). *XML* on W3C:n (World Wide Web Consortium) suositus rakenteisten dokumenttien ja tiedon esitysmuodoksi tietoverkoissa. Hanslon & al. (2004) mukaan rakenteiset *XML*-pohjaiset dokumentit ovat laitteisto- ja ohjelmistoriippumattomia, koska tekstipohjainen esitysmuoto ei aseta erityisvaatimuksia dokumentteja käsittelevälle ympäristölle. Riippumattomuuden ansiosta *XML*-pohjaiset dokumentit ovat siirrettäviä eri ympäristöjen välillä.

XML-dokumentin rakenne saadaan aikaan merkitsemällä teksti sisältöä kuvaavilla merkitsemistunnisteilla. Rakenteisten dokumenttien tietomalli on puumainen hierarkia, joka on ilmaisuvoimainen esitystapa sekä tiedon määrittelyyn että käsittelyyn. Hanslon & al. (2004) mukaan rakenteisia dokumentteja kutsutaan itsekuvautuviksi, koska ne sisältävät tekstisisällön joukkoon upotettua rakennetta sekä sisällön merkitystä kuvaavaa metatietoa. Rakenteisessa dokumentoinnissa tiedon ulkoasua on erotettu sisällöstä ja sisältöä kuvaavasta rakenteesta. Tämä mahdollistaa dokumenttien tuottamisen, hallinnan ja siirron julkaisutavasta riippumattomalla tavalla. Itsekuvautuvista dokumenteista on mahdollista tuottaa useita erilaisia julkaisuja määrittelemällä ulkoasutieto käyttötarpeen ja julkaisuvälineen mukaan.

Rakenteisten dokumenttien rooli sovellusten välisessä kommunikoinnissa on tarjota standardi tiedon esitysmuoto sanomien välitykseen. Hanslon & al. (2004) mukaan *XML*-pohjaisen viestin ominaisuuksiin kuuluu, että se kykenee kulkemaan verkossa *HTTP*-yhteyskäytännön yhteydessä. Tämän ominaisuuden ansiosta *XML*-pohjainen yhteyskäytäntö kykenee tarvittaessa läpäisemään tietoverkkojen liikennettä valvovat palomuurit, jotka ovat aiheuttaneet suuria ongelmia muille proseduurien etäkutsuista huolehtiville mekanismeille.

XML-RPC on *XML*-pohjainen toteutus *RPC* (Remote Procedure Call) -tekniikasta. Tekniikka mahdollistaa muiden hajautustekniikoiden tavoin hajautetuilla palvelimilla sijaitsevien hajautettujen komponenttien käyttämisen aivan kuin ne olisivat paikallisella palvelimella. Proseduurien etäkutsulla tarkoitetaan asiakas-palvelin -suhdetta, jossa kom-

munikointi järjestelmien välillä tapahtuu palvelupyyntöjen muodossa. Palvelupyyntö lähetetään määrättyyn *URL*-osoitteeseen, joka sijaitsee palvelinjärjestelmässä. Palvelinjärjestelmä käsittelee pyynnön ja toteuttaa vaaditun operaation mahdollisten mukana seuraavien parametrien arvoilla ja muodostaa sen jälkeen tuloksen, joka palautetaan asiakasjärjestelmälle. *XML-RPC* on hajautettujen sovellusten toteutustekniikka, jossa käytetään *HTTP*-yhteyskäytäntöä ja *XML*-dokumentteja komponenttien väliseen kommunikointiin. Lisäksi se on ohjelmointikielestä ja alustasta riippumaton ja yksinkertaisuutensa ansiosta erittäin helppo käyttää (Allman, 2003).

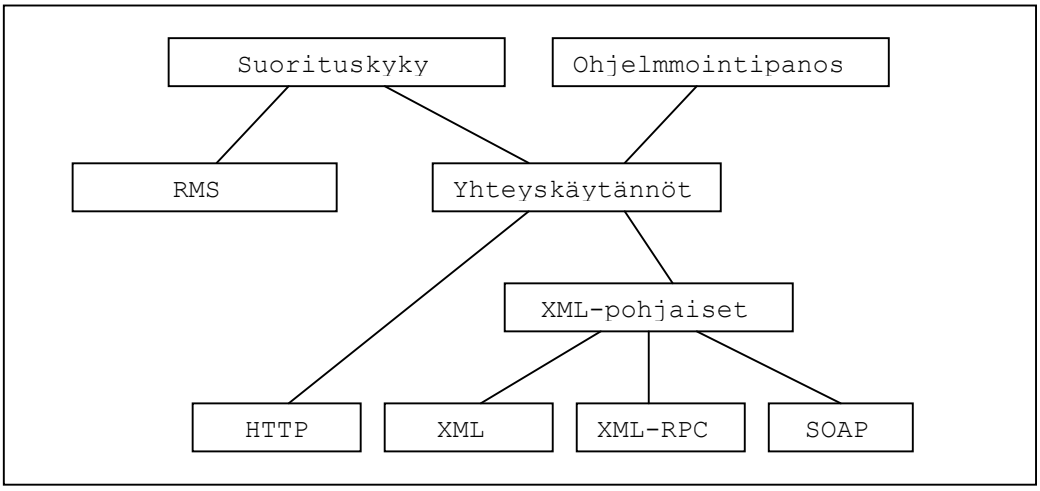
SOAP (Simple Object Access Protocol) on hajautetussa ja verkkoympäristössä tapahtuvaan informaation välitykseen tarkoitettu kevyt ja *XML*-kieleen pohjautuva tietoliikennemenetelmä. *SOAP* on tarkoitettu rakenteisen ja tyypitetyn tiedon välittämiseen sovellusten välillä. *SOAP* määrittelee vain viestirakenteen, eikä se kykene kulkemaan verkossa itsenäisesti. Mahdollisia sanomien kuljetuskäytäntöjä ovat esimerkiksi *TCP/IP*, *HTTP* ja *SMTP*. Tyypillisin tapa tällä hetkellä on käyttää *HTTP*-yhteyskäytäntöä SOAP-sanomien kuljetukseen (Tian & al., 2004).

XML-pohjaisilla yhteyskäytännöillä saavutetaan monia merkittäviä etuja hajautettujen järjestelmien välisessä tiedonsiirrossa. Yuan'n (2003) käsityksen mukaan *XML*-pohjaisilla yhteyskäytännöillä saavutetaan löyhä sidonta asiakas- ja palvelinsovellusten välille, josta seuraa hyvä siirrettävyys ja yhteiskäytettävyys. Toisaalta *XML*-pohjaisuus lisää siirrettävän tiedon määrä, joka voi johtaa huonoon suorituskykyyn langattomassa verkossa. Binaarimuodossa tapahtuva tiedonsiirto mahdollistaa tiiviin ja optimoidun tiedon esitysmuodon, josta seuraa hyvä tiedonsiirron tehokkuus. Toisaalta asiakas- ja palvelinpuoli sidotaan tiukasti toisiinsa, joka johtaa huonoon yhteiskäytettävyyteen.

1.4 Tutkielman tavoitteet ja tutkimusongelman asettelu

Tässä tutkielmassa keskitytään selvittämään mobiilin *MIDP*-pohjaisen sovelluksen toteutusta, jonka toimintaympäristönä on niukkaresurssiset päätelaitteet. Tällaisia laitteita ovat esimerkiksi matkapuhelimet ja kämmentietokoneet. Tutkimuksen kohteena on myös etälaitteen rooli osana laajempaa järjestelmää sekä viestinvälitys järjestelmän

komponenttien välillä. Viestinvälitys hajautetuissa ympäristöissä voidaan toteuttaa monella eri tavalla. Tässä tutkielmassa yleistavoitteena on selvittää empiirisesti esimerkkisovelluksen avulla *XML*-pohjaisten viestinvälitystekniikoiden soveltuvuutta käytettäväksi langattomassa *mobiili*-ympäristössä.



Kuva 1.2: Tutkielman tavoitteet

Kuvassa 1.2 tutkielman yleistavoite on tarkennettu suorituskyvyn ja ohjelmointipanoksien selvittämiseen yhteyskäytäntöjen osalta sekä suorituskyvyn selvittämiseen *RMS*-tietuevaraston osalta. Mobiilijärjestelmillä on käytettävissä vähän muistia ja suhteellisen heikkotehoinen prosessori. Toisaalta niillä on mahdollista saada yhteys verkossa oleviin tietokoneisiin, joiden suorituskyky ylittää mobiililaitteiden kapasiteetin moninkertaisesti. Siksi verkossa sijaitsevia palvelimia voidaan käyttää mobiililaitteiden toiminnan laajentamiseen. Verkkoyhteyden kautta tapahtuvaan palvelinten kapasiteetin hyödyntämiseen mobiililaitteilla on käytettävissä suhteellisen hidas ja epävarma kapeakaistainen verkkoyhteys. Tästä syystä siirrettävän tiedon määrällä ja esitysmuodolla voi odottaa olevan merkitystä siirron ja käsittelyn suorituskykyyn.

Tutkielmassa verrataan *XML*-pohjaisten yhteyskäytäntöjen suorituskykyä perinteiseen *HTTP*-asiakas/palvelin -mallin mukaiseen yhteyskäytäntöön, jossa tiedonsiirto asiakassovelluksen ja palvelinsovelluksen välillä tapahtuu *HTTP*-yhteyskäytännön mukaisesti binaarimuodossa. Lisäksi tutkielmassa verrataan *XML*-pohjaisten menetelmien ohjelmointityön tehokkuutta suhteessa perinteiseen *HTTP*-asiakas/palvelin -mallin mukai-

seen yhteyskäytäntöön. *RMS*-tietuevaraston suorituskykyä tutkielmassa selvitetään tiedon tallennuksen osalta eri suurilla tietomäärillä. Aiheen laajuuden vuoksi tutkielman ulkopuolelle on rajattu tietoturvan käsittely. Samoin käyttöliittymäohjelmoinnin esittely on rajattu tämän tutkielma ulkopuolelle. Käyttöliittymän rakennetta ja toteutusta ovat tarkastelleet esimerkiksi Riggs & al. (2001), Mahmoud (2001a), White ja Hemphill (2002) sekä Holopainen (2002; 2003).

1.5 Tutkielman rakenne

Tutkielma on jäsennetty luvuittain seuraavasti. Luvut 2 ja 3 esittelevät *J2ME*-ohjelmointiympäristön luvuissa 4 ja 5 esiteltävien yhteyskäytäntöjen ja esimerkkisovelluksen ja niiden analysoinnin ymmärtämiseksi. Luku 2 esittelee erilaiset *Java*-ohjelmointiympäristöt sekä erityisesti mobiiliympäristöön tarkoitetun *J2ME*-ohjelmointiympäristön. Luvun pääpainona on esitellä yleisellä tasolla *J2ME*-ohjelmointiympäristön ominaisuudet ja rajoitukset. Luvussa 3 käydään läpi *MIDP*-pohjaisten sovellusten elinkaarimalli, joka ohjaa kaikkien *MIDP*-sovellusten ajonaikaista suoritusta. Lisäksi luvussa esitellään *RMS*-tietuevarasto, jonka avulla *MIDP*-sovellukset voivat tallentaa tietoa suoraan päätelaitteen muistiin.

Luvussa 4 esitetään miten *MIDP*-sovellukset voivat hyödyntää verkossa olevia palveluita ja tietovarastoja. Luvussa käsitellään myös *XML*-pohjaisten yhteyskäytäntöjen perusteita toteuttajan näkökulmasta määritysten pohjalta. Luvussa 5 *MIDP*-pohjaisten sovellusten toteutusta tarkastellaan tutkielman tavoitteiden mukaisesti suorituskyvyn ja ohjelmoijan näkökulmasta. Luvussa esitellään tutkielmassa käytettävät esimerkkisovellukset sekä saadut tulokset. Lopuksi luvussa 6 luodaan yhteenveto luvuissa 4 ja 5 saaduista tuloksista sovelluksen suorituskyvyn ja ohjelmoijan työpanoksen osalta.

2 J2ME-OHJELMOINTIYMPÄRISTÖ

Java on tulkattava ohjelmointikieli, jonka alkuperäinen tarkoitus oli helpottaa sulautettujen järjestelmien ohjelmointia tarjoamalla C/C++-ohjelmointia enemmän tukea ohjelmoijalle erityisesti automatisoidun resurssien vapauttamisen muodossa. Kun kieli esiteltiin 1990-luvun alussa, sen käyttökohteeksi kuitenkin muodostui lähinnä verkko-ohjelmointi Internetissä, sillä monet *Javan* ominaisuudet, erityisesti siirrettävyyden mahdollistava ajoajan ympäristö, sopivat hyvin tähän tarkoitukseen. Mobiiliympäristöön siirtymisen myötä *Javan* voi ajatella palanneen juurilleen, resursseiltaan rajoittuneeseen ympäristöön. Sun Microsystems on jakanut *Java*-alustat (Java 2 Platform) kolmeen erikokoisille järjestelmille tarkoitettuun osaan, jotka esitellään kohdassa 2.1. Kohdassa 2.2 käydään läpi mobiililaitteille tarkoitetun *J2ME*-ohjelmointiympäristön ominaisuuksia ja rajoituksia.

2.1 Java-alustat

Koska *Java*-ympäristössä on alunperinkin otettu huomioon esimerkiksi tietoturva ja ohjelman lataaminen verkkoyhteyden ylitse, on *Java* hyvin luonnollinen valinta mobiiliympäristöönkin. Toisaalta erilaiset laiteympäristöstä johtuvat vaatimukset poikkeavat toisistaan huomattavasti laskentatehon ja muistimäärän suhteen. *Java*-standardissa on tämä otettu huomioon, joten se tarjoaa kokonaisen perheen erilaisia ympäristöjä. Vuonna 1999 julkistetun *Java* 2:n myötä *Java*-ympäristöt eriytyivät käyttötarpeiden mukaan. Standardiympäristön lisäksi tarjottiin työvälineitä sekä verkkopalvelimen että matkapuhelimen sovellusohjelmointiin (White ja Hemphill, 2002):

- *Java 2 Standard Edition* (J2SE) on *Javan* perusympäristö. Se on ensisijaisesti tarkoitettu *PC*- ja *WWW*-maailman *Javaksi*, joka edellyttää suuren suorituskyvyn omaavaa laitteistoa ja paljon muistia. Vastapainoksi standardi tarjoaa laajan kirjastovalikoiman eri tarkoituksiin.

- *Java 2 Enterprise Edition* (J2EE) on palvelinkäyttöön tarkoitettu *Java*-ratkaisu. Teknisesti *J2EE* on *J2SE* täydennettynä palvelinkäyttöön soveltuvilla kirjastoilla.
- *Java 2 Micro Edition* (J2ME) on mobiilikäyttöön tarkoitettu *Java*-tekniikka. Tarkkaan ottaen *J2ME* määrittelee kaksi erilaista ohjelmistokokoonpanoa, jotka eroavat toisistaan huomattavasti. Monimutkaisempi näistä kokoonpanoista on *Personal Java*. Sen virtuaalikone, samoin kuin ajoajan vaatimukset muistin ja suorituskyvyn osalta, ovat hyvin lähellä *J2SE*:ssä käytettyä virtuaalikonetta ja ohjelmistovaatimuksia. Tästä syystä *Personal Java* ei liene käyttökelpoinen kaikkein yksinkertaisimmissa mobiililaitteissa. Yksinkertaisempi mobiiliympäristöön tarkoitetuista *Java*-kokoonpanoista perustuu toisenlaiseen ratkaisuun, jossa käytetty virtuaalikone on nimeltään *Kilo*-virtuaalikone (KVM, Kilo virtual machine). Tätä kokoonpanoa suunniteltaessa on pyritty huomioimaan erityisesti heikkotehoisten päätelaitteiden vaatimukset ja rajoitukset.

Java-ympäristön ydin on tavukoodina annettua ohjelmaa suorittava virtuaalikone. Virtuaalikone on ohjelmistolla toteutettu kone, joka toimii laitteistoarkkitehtuurin ja käyttöjärjestelmän päällä hyödyntäen niiden resursseja. Virtuaalikone sallii saman ohjelman virtuaalisen suorituksen kaikissa samaa virtuaalikonetta tukevissa laitteisto- ja käyttöjärjestelmäkokoonpanoissa. Koska sama perusinfrastruktuuri voidaan toteuttaa moneen ympäristöön, virtuaalikoneiden avulla on mahdollista luoda perinteisiä ohjelmistoja siirrettävämpiä järjestelmiä. Lisäksi turvallisuusasioiden huomiointi on helpompaa, sillä tarkastukset voidaan kohdentaa niihin rajapintoihin, joihin on mahdollista päästä käsiksi virtuaalikoneen kautta. On myös mahdollista siirtää esimerkiksi resurssien varaamiseen ja vapauttamiseen liittyviä asioita ohjelmoijalta suoritusaikaisen ympäristön ja virtuaalikoneen vastuulle. Esimerkiksi resurssien vapautus ei *Java*-ympäristössä ole ohjelmoijan tehtävä, vaan ajoajan ympäristö huolehtii automaattisesti resurssien vapauttamisesta (White ja Hemphill, 2002).

2.2 J2ME-arkkitehtuuri

J2ME-ohjelmointiympäristön tavoitteena on tarjota *Java*-alusta niukkaresurssisille mobiilipäätelaitteille. Laitteet ovat erilaisia, joten arkkitehtuurin on oltava helposti mukautuva, mutta samalla yhteensopivuuden eri toteutusten välillä on säilyttävä. Sulautettujen järjestelmien laitteita ovat mm. seuraavat: matkapuhelimet, televisiot, pesukoneet, *PDA*-kämmentietokoneet ja autojen ajotietokoneet sekä paikannusjärjestelmät. Tässä kohdassa esitellään olennaiset osat, joista *J2ME*-ohjelmointiympäristö koostuu. Aluksi alakohdassa 2.2.1 tarkastellaan *J2ME*-arkkitehtuurin tavoitteita. Alakohdassa 2.2.2 määritellään termi *konfiguraatio*, joka määrittelee laitteiden fyysiset ominaisuudet. *Profiili*, jonka avulla määritellään luokkakirjastot ja rajapinnat tietylle konfiguraatiolle, esitellään alakohdassa 2.2.3. Lopuksi alakohdassa 2.2.4 tarkastellaan *J2ME*-ympäristön turvallisuusmallia.

2.2.1 J2ME-arkkitehtuurin tavoitteet

J2ME:n arkkitehtuuri poikkeaa olennaisesti *J2SE*:n ja *J2EE*:n arkkitehtuurista johtuen erilaisista tavoitteista. *J2ME*-ohjelmointiympäristön tavoitteena on tarjota *Java*-alusta sulautettuihin järjestelmiin. Laitteet ovat erilaisia, joten arkkitehtuurin on oltava helposti mukautuva, mutta samalla yhteensopivuuden eri toteutusten välillä on säilyttävä. White ja Hemphill (2002) esittävät seuraavia tavoitteita *J2ME*-arkkitehtuurille:

- Tarjota eri laitteisiin yhtenäinen alusta ottaen arkkitehtuurissa huomioon myös niiden väliset erot. Erilaiset prosessorit, virtalähteet, muistimäärä, tallennusmahdollisuudet ja käyttöliittymät vaativat paljon joustavuutta.
- Tarjota arkkitehtuuri, joka voidaan optimoida pieneen tilaan.
- Soveltua henkilökohtaisiin laitteisiin, joita ovat esimerkiksi matkapuhelimet, hakulaitteet ja *PDA*-laitteet.
- Soveltua laitteisiin, joissa voi olla langaton tai kiinteä verkkoyhteys.
- Tarjota mahdollisuus ladata sovellukset langatonta tiedonsiirtotietä pitkin laitteeseen.

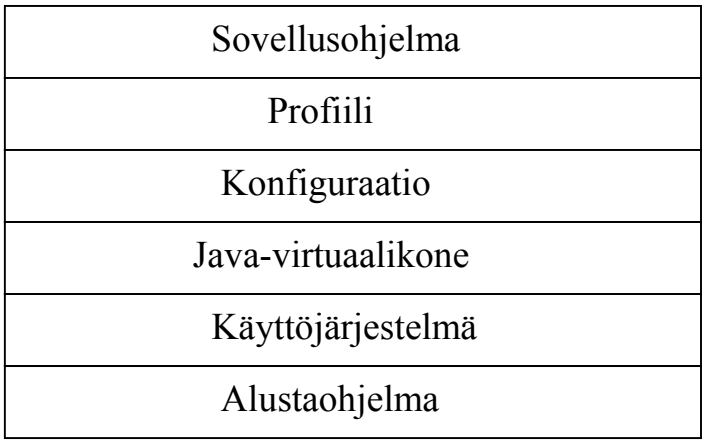
- Maksimoida *Java*-ympäristölle ominainen laitteistoriippumattomuus, samalla kuitenkin huomioiden yksittäisille laitteille ominainen kapasiteetti ja asetetut rajoitukset.
- Maksimoida joustavuus ja tarjota keino sopeutua nopeasti muuttuviin markkinoiden haasteisiin.
- Mahdollistaa kolmansien osapuolten tekemien sovellusten turvallisen asennuksen *J2ME*-tuetulle laitteelle laitevalmistajasta riippumatta.
- Tarjota keino sovittaa sovellukset erilaisille laitteille, joiden ominaisuudet poikkeavat mm. kapasiteetin ja resurssien suhteen.

Ohjelmistojen siirrettävyys on ollut *Java*-teknologioiden käytön suurimpia etuja työasema- ja palvelinympäristöissä. Koska *Javalla* toteutetut ohjelmat käännetään tavukoodiksi, ne voidaan suorittaa prosessoririippumattomasti ohjelmistoa hyväksi käyttäen. Koska laitteet ovat käyttötarkoituksiltaan, ominaisuuksiltaan ja kooltaan hyvin erilaisia, on *J2ME*-ohjelmien siirrettävyyden takaamiseksi määritelty *konfiguraatiot* ja *profiilit*, joiden mukaan toteutetut ohjelmat toimivat kaikissa niitä tukevissa laitteissa.

2.2.2 Konfiguraatiot

Sulautettujen järjestelmien ja pienten laitteiden kirjo on valtava. *J2ME* tarjoaa laitteisiin yhtenäisen rajapinnan ottaen huomioon myös niiden väliset erot, kuten muistin määrän ja prosessointitehon. Siksi *J2ME* päätettiin jakaa sopiviin osiin. Ensimmäisenä erotettiin laitteiden fyysiset ominaisuudet ja *J2ME*:n määriteltiin koostuvan konfiguraatioista. Konfiguraatio on kohdistettu fyysisiltä ominaisuuksiltaan tietynlaisille laitteille. Konfiguraatio voidaan ymmärtää laitevalmistajan ja *Java*-ympäristön toteuttajan väliseksi sopimukseksi, joka määrittelee tietyn laiteryhmän vähimmäisvaatimukset. Konfiguraation tulee tarjota ohjelmistoille mahdollisimman toimiva alusta ja riittävät palvelut, jotta ohjelmistokehitys saadaan tehokkaaksi ja ohjelmistot luetettaviksi. Se ei kuitenkaan saa olla liian raskas kyseiselle laitekategorialle, jottei se vie kaikkia resursseja laitteelta. Esimerkiksi muistikapasiteetti on sulautetuissa järjestelmissä lähes poikkeuksetta rajallinen (White ja Hemphill, 2002).

Kuva 2.1 havainnollistaa virtuaalikoneen, konfiguraation ja sen päälle rakennettujen profiilien välistä suhdetta (White ja Hemphill, 2002). Tällä hetkellä *J2ME*:n konfiguraatioita on määritelty kaksi: *Connected Device Configuration* (CDC) ja *Connected Limited Device Configuration* (CLDC). *CDC* on suunniteltu resursseiltaan pienille laitteille, joilla on kiinteä verkkoyhteys. *CDC* vaatii muistia vähintään 512 kilotavua. Tyypillisiä tähän konfiguraatioon kuuluvia laitteita ovat digi-tv, autojen navigointijärjestelmät ja kehittyneemmät kommunikointivälineet. *CLDC* on määritelty resursseiltaan hyvin pienille henkilökohtaisiin verkkoon kytkettyihin mobiililaitteisiin, joita ovat esimerkiksi matkapuhelimet, hakulaitteet ja *PDA*-laitteet (White ja Hemphill, 2002).



Kuva 2.1: *J2ME*-arkkitehtuuri.

Matkapuhelimet, hakulaitteet ja autojen navigointijärjestelmät eroavat ominaisuuksiltaan, ulkomuodoltaan ja toiminnoiltaan huomattavasti toisistaan. Tämän vuoksi *J2ME*-alusta tarjoaa vain minimaalisen *Java*-virtuaalikoneen ja sovelluskehittäjälle ohjelmointirajapinnan, joka tukee jokaisen laitteen erityisominaisuuksia. Toteutustasolla *J2ME*-konfiguraatio määrittelee joukon ohjelmointirajapintoja tuoteryhmille, joilla on samanlaiset vaatimukset muistimäärälle ja prosessointiteholle. *J2ME* määrittelee seuraavat ominaisuudet (Sun Microsystems, 2000a):

- *Java*-virtuaalikoneen ominaisuuksien tuen,
- *Java*-ohjelmointikielen tuen,
- tuettavat *Java*-kirjastot ja ohjelmointirajapinnat,

- minimilaitteistovaatimukset, kuten muistin, näytön koon ja prosessoritehon.

CLDC on *Java*-konfiguraatio, jossa mobiililaitteen erikoispiirteet on otettu huomioon. Tähän kategoriaan kuuluvilla laitteilla on yleensä heikkotehoinen prosessori, vähäinen muistin määrä ja kapeakaistainen verkkoyhteys. Konfiguraatio määrittelee minimitoiminnallisuuden, joka samaan kategoriaan kuuluvien laitteiden on toteutettava. *CLDC*-määritys asettaa laitteille muistin riittävyyteen liittyviä laitteistovaatimuksia. Määrityksen mukaan virtuaalikonetta ja *CLDC*:n kirjastoa varten on oltava vähintään 128 kilotavua pysyvää (non-volatile) muistia. *Java* ajoympäristölle ja olioille on puolestaan oltava vähintään 32 kilotavua käyttömuistia (volatile).

Johtuen rajoittuneesta suoritusympäristöstä, mobiiliympäristössä on joitakin *Javan* perusominaisuuksia jouduttu karsimaan. Osa ominaisuuksista on jätetty pois turvallisuussyistä ja osa resurssien niukkuuden vuoksi. Seuraavassa käydään läpi suurimmat poikkeamat White'n ja Hemphillin (2002) mukaan:

- *Liukulukuja ei tueta*: Liukulukujen poistaminen johtui alun perin siitä, että *CLDC*:tä tukevissa laitteissa ei ollut tuettu liukulukua laitetasolla, mistä syystä tuen toteuttaminen konfiguraatiossa katsottiin liian raskaaksi.
- *Finalizy-operaatita ei tueta*: *CLDC* API ei sisällä *Object.finalize*-metodia, vaan kaikki jätetään virtuaalikoneen roskien keruun huoleksi. Operaation toteuttaminen katsottiin liian raskaaksi ja käytännössä huonosti toimivaksi erityisesti mobiililaitteissa.
- *Virheenkäsittely on rajoitetusti tuettu*: *CLDC* määrittelee ajonaikaisten virheiden käsittelyyn kolme luokkaa. Muut kuin ajonaikaiset virheet käsitellään eri tavalla laitteesta riippuen.
- *Java-natiivirajapintaa ei tueta* (Java Native Interface, JNI): *JNI* mahdollistaisi kutsua *CLDC*:n alla olevan käyttöjärjestelmän kutsuja suoraan. *JNI*:n tuesta luovuttiin kahdesta syystä. Ensimmäinen syy oli se, että natiivirajapinnan toteuttaminen kuluttaisi liikaa muistia. Lisäksi rajapinnan toteuttaminen olisi vaarantanut mobiiliympäristön *Java*-turvallisuutta.
- *Käyttäjän määrittelemää luokkalataajaa (class loader) ei tueta*: Tuesta luovuttiin, koska sen katsottiin muodostavan turvallisuusriskin ja lisäksi sen toteutus olisi vaatinut liikaa resursseja.

- *Reflektointia (reflection) ei tueta*: Refleksiivisyys mahdollistaa olioiden tarkastelun suorituksen aikana sekä kenttien päivittämisen ja metodien kutsumisen dynaamisesti. Refleksiivisyyden toteutuksen katsottiin kuluttavan liikaa resursseja ja muodostavan turvallisuusriskin. Tuen puuttumisen seurauksena *RMI* ja olioiden sarjallistaminen ei ole mahdollista.
- *Heikkoa viittausta (weak reference) ei tueta*: Heikon viittauksen avulla ohjelma voi säilyttää viitteen olioon, jottei automaattinen roskienkeruu tuhoa sitä. Heikon viittauksen toteutusta pidettiin liian raskaana toimintona.
- *Säieryhmiä ei tueta*: Mobiiliympäristön *Java* toteuttaa säikeisiin perustavan moniajon, muttei säieryhmiä. Säieryhmien tuen poistamisen syynä olivat resurssien vähäisyys.

2.2.3 Profiilit

J2ME-ympäristö on jaettu kahteen *konfiguraation*. Silti sama *konfiguraatio* voi kattaa käyttötarkoitukseltaan aivan erityyppisiä laitteita, kuten esimerkiksi matkapuhelimia ja pesukoneita. Selvästikään ei ole tarkoituksenmukaista, että pesukoneen ohjelmia pystyttäisiin ajamaan matkapuhelimessa tai toisinpäin. Tästä syystä on tarpeen pystyä määrittelemään sellaisia palveluita ja ominaisuuksia, jotka ovat ominaisia kullekin laitteistotyypille. Erityyppisten laitteiden tukeminen vaatii lisätoimintoja, jotka toteutetaan *profiilien* avulla. *Profiili* on spesifikaatio, jossa kuvataan yksityiskohtaisesti *konfiguraation* päälle rakentuva *Java*-rajapinta, joka tarjoaa tuen tietyntyyppiselle laitteelle, kuten esimerkiksi matkapuhelimille (White ja Hemphill, 2002).

Tietylle *profiilille* tehdyt ohjelmat toimivat jokaisessa saman *profiilin* tarjoavassa laitteessa ilman lisävaatimuksia. *Profiili*-määritys on se elementti, joka mahdollistaa ohjelmien siirtämisen laitteesta toiseen. Kun ohjelma on luotu yhden *profiilin* päälle, niin se toimii kaikissa laitteissa, joissa kyseinen *profiil*i on toteutettu. *Profiilit* laajentavat *konfiguraatiota* ja niiden avulla voidaan hyödyntää laitteiden erityisominaisuuksia. *Profiili* voidaan ajatella sopimukseksi ohjelmoijan ja *Java*-ympäristön toimittajan välillä. *Konfiguraatioiden* tapaan profiilit määrittelee *JCP* (Java Commity Process) yhteistyössä muiden kyseisen alojen laitevalmistajien kanssa (White ja Hemphill, 2002).

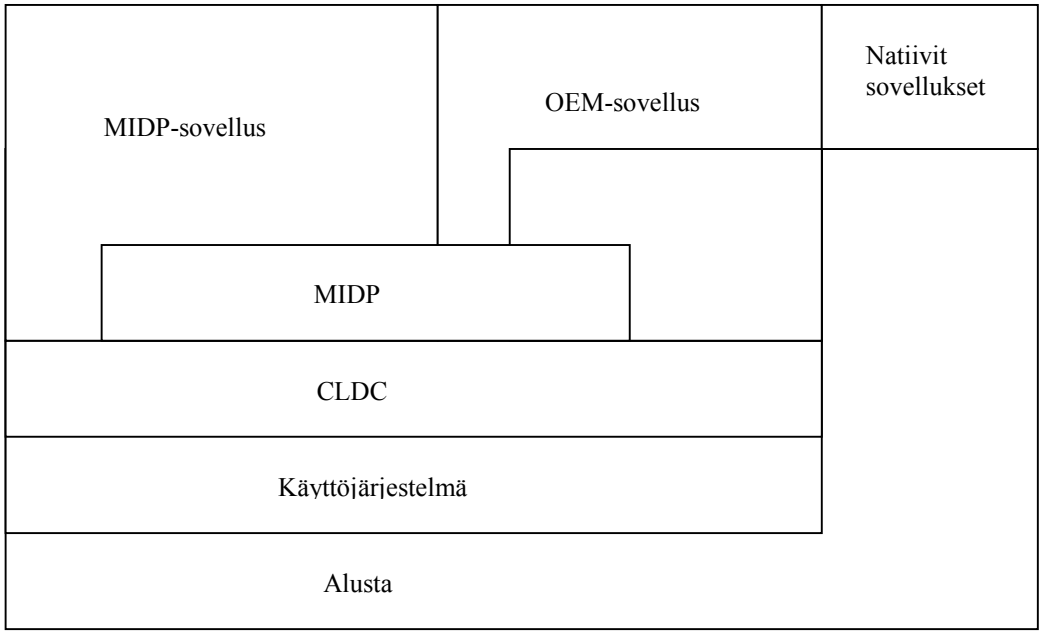
Selvästi tunnetuin *profiili* on *MIDP*-profiili (Mobile Information Device Profile), joka on tarkoitettu mobiililaitteisiin, kuten matkapuhelimiin ja hakulaitteisiin. *Profiilin* toteuttavassa ympäristössä ajettavaa *Java*-sovellusta kutsutaan *midletiksi*. *MIDP*-profiili määrittelee käyttöliittymän ohjelmoinnissa tarvittavat rajapinnat, liitynnät tiedot säilyttävään kirjastoon ja yksinkertaisen verkkorajapinnan, jota voi hyödyntää *MIDP*-profiilin mukaisissa *Java*-sovelluksissa (White ja Hemphill, 2002).

Profiili on tarkoitettu kolmansien osapuolten matkaviestimien sovelluskehittäjille, joiden kohdealueena ovat erilaiset mobiilipalvelut ja niiden tuottaminen. Tämä on tärkeää laitevalmistajien kannalta siksi, että laitteiden myynti tulee lähitulevaisuudessa riippumaan paljon siitä, minkälaisia ohjelmia, pelejä sekä muita lisäarvopalveluja näihin laitteisiin on saatavilla (White ja Hemphill, 2002).

MIDP-profiilin perusvaatimuksena on 128 kilotavua muistia *Java*-sovellukselle, 32 kilotavua ajonaikaista muistia ja 8 kilotavua muistia tiedon tallentamiselle. *Profiili* vaatii myöskin kaksisuuntaisen verkkoyhteyden sekä mahdollisuuden tiedon syöttöön, esimerkiksi näppäimistön tai näytön välityksellä. *Profiili* tukee vain *HTTP*-yhteyskäytönnön mukaista verkkoyhteyttä. Sun Microsystems (2000b) määrittelee myös laitteiden näytölle vähimmäisvaatimukset.

MIDP-profiili tarjoaa mahdollisuuden tallentaa tietoa laitteen muistiin ja käyttää sitä myöhemmin. Tallennusmekanismin nimi on *Record Management System* (RMS). Tallennuksilla on sama elinkaari kuin *MIDlet Suitella*, eli ne ovat olemassa kunnes *MIDlet Suite* tuhotaan. *MIDP*-sovellukset voivat käyttää muistiin tallennettuja tietoja ristiin yhden *MIDlet Suiten* sisällä. Toinen *MIDP*-sovellus voi esimerkiksi tallentaa tietoa, jota toinen sovellus käsittelee. Tallennusoperaatiot ovat atomisia, synkronoituja ja sarjallistettuja, joten tieto pysyy turmeltumattomana myös päällekkäisten tallennusyritysten jälkeen. Tieto tallennetaan muistiin tavutaulukkona. *RMS* tarjoaa valmiit rajapinnat tallennettavan tiedon muuttamiseen tavutaulukoiksi, joten tallennettava tieto voi olla minkä muotoista tahansa (White ja Hemphill, 2002).

Kuvassa 2.2 on esitetty Sun Microsystemsin (2000a) *MIDP*-määrityksen mukaan miten *MIDP*-profiili sijoittaa päätelaitteen *Java*-ympäristöön. Koska *MIDP* kattaa laajan kokoelman erilaisia laitteita, sen ei ole mahdollista tukea kaikkia eri laitevalmistajien (OEM, Original Equipment Manufacturer) laitteiden erilaisia ominaisuuksia. Siksi laitevalmistajat saattavat tarjota omia luokkiaan (OEM-Specific Classes), joilla näihin laitekohtaisiin ominaisuuksiin pääsee käsiksi.



Kuva 2.2: *J2ME*-arkkitehtuuri.

2.2.4 Turvallisuusmalli

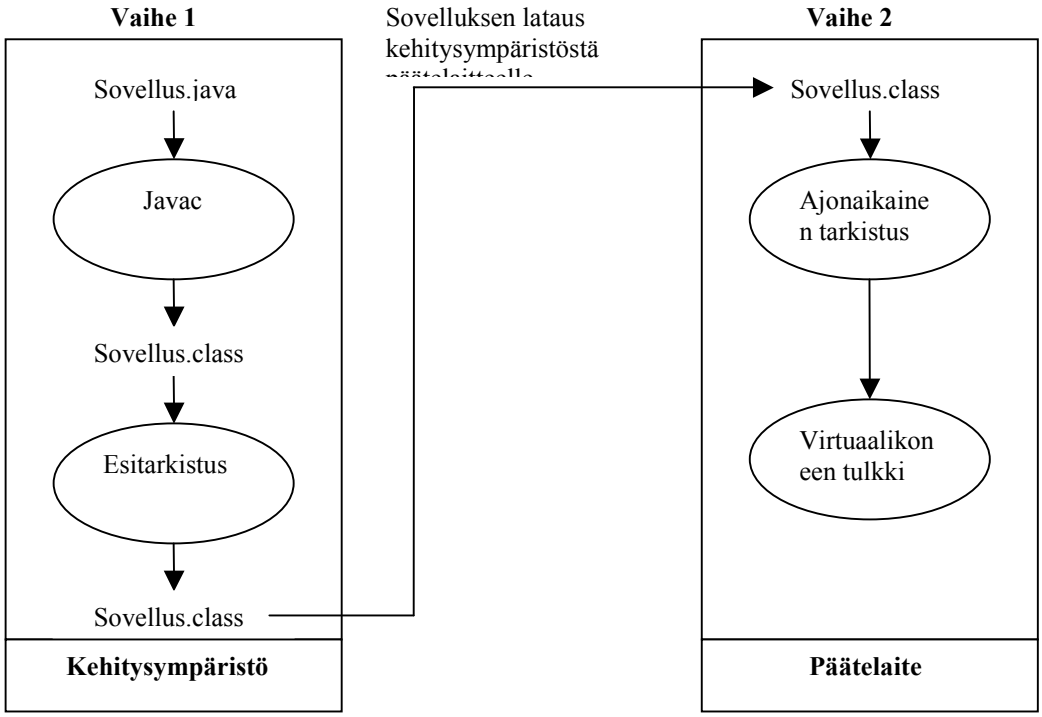
Mobiiliympäristössä sovelluksen kehitysympäristön tietoturvaominaisuuksille asetetaan vaatimuksia, koska sovelluksia voidaan ladata päätelaitteelle verkon yli. Tällöin on tärkeää varmistua, ettei sovellukset pääse vahingoittamaan laitetta eikä verkkoa. *J2SE* tarjoaa laajat turvallisuusominaisuudet, mutta niitä ei laajuutensa takia voitu sovittaa suoraan *J2ME*-ympäristöön. Tästä syystä *J2ME*-ympäristölle luotiin oma turvallisuusmalli *J2SE*:n pohjalta. White ja Hemphill (2002) sekä Mahmoud (2002) jakavat *CLDC*-konfiguraation turvallisuusmallin kahteen tasoon seuraavasti:

- *Matalan tason turvallisuusmalli*: Tunnetaan myös nimellä virtuaalikoneen turvallisuusmalli ja tarkoittaa lähinnä virtuaalikoneen tasolla tarvittavia turvallisuusominaisuuksia.
- *Sovellustason turvallisuus*: Sovellustason turvallisuuden avulla hallitaan sitä mihin rajapintoihin ja kirjastoihin sovelluksella on pääsy.

Matalan tason turvallisuusominaisuudet käsittävät ominaisuudet, joilla estetään haitallisen ohjelmiston aiheuttama virtuaalikoneen toiminnan häirintä tai laitteelle vahingollinen toiminta. Normaalisti *Javassa* suoritetaan tavukoodin tarkistus (bytecode verification) ennen sen suorittamista virtuaalikoneessa. Tällä varmistetaan, etteivät sovellukset voi viitata sallitun muistialueen ulkopuolelle ja suoritettavat luokkatiedostot ovat kelvollisia. *Java*-virtuaalikoneen on hylättävä epäkelvot tiedostot (Mahmoud, 2002).

J2SE-ympäristön tavukoodin tarkistus käyttää vähintään 50 kilotavua binaarikoodia ja 30-100 kilotavua ajonaikaista muistia, mikä saattaisi kuluttaa matkapuhelimen kaiken muistin. Lisäksi normaalia *Javaa* heikommasta suorituskyvystä johtuen *KVM*-pohjaiset *Java*-ympäristöt eivät kykene yleiseen luokkatiedoston tarkistukseen. Tästä syystä osa luokkatiedoston tarkistusrutiineista suoritetaan jo kehitysympäristössä. *CLDC*-määrityksessä luokkatiedostojen tarkistukset on toteutettu huomioiden laitteiden niukat resurssit. Tarkistuksen toteutus *CLDC*:n virtuaalikoneessa käyttää noin 10 kilotavua binaarikoodia ja vain 100 tavua ajonaikaista muistia (Sun Microsystems, 2000a).

KVM-virtuaalikoneessa ajettava koodi esitarkistetaan ennen sen suorittamista. Jokainen *CLDC*:n päällä ajettava sovellus on varustettu muutamilla tarkistuksilla ennen käyttöä, jottei se kuluta liikaa laitteen resursseja ja suoritusaikaa. Esitarkistus voidaan tehdä yleensä sovelluskehitysvaiheessa sovelluskehittimellä tai vaihtoehtoisesti palvelimella ennen sovelluksen lataamista päätelaitteelle. Mahmoudin (2002) mukaan esitarkistuksen seurauksena tavukooditiedostot ovat noin viisi prosenttia suurempia, koska niihin tallennetaan lisäattribuutteja. Kuvassa 2.3 on kuvataan Mahmoidin (2002) mukaan miten sovelluksen tarkistus on toteutettu *CLDC*/*MIDP*-ympäristössä.



Kuva 2.3: Sovelluksen tarkistus *CLDC/MIDP* -ympäristössä (Mahmoud, 2002).

Sovellustason turvallisuusominaisuuksien avulla hallitaan sovelluksen pääsyä rajapintoihin ja kirjastoihin. *J2ME* on lainannut sovellustason turvallisuusmallinsa perusteet *J2SE*-ympäristöltä. Käytännössä tämä on toteutettu ns. hiekkalaatikkomenetelmällä, jossa sovelluksella on mahdollisuus päästä käsiksi vain rajattuun ja tarkasti kontrolloituun rajapintojen joukkoon. Hiekkalaatikkomalli voi tuntua rajoittavalta, mutta rajoitteista seuraa luonnollisesti myös turvallisuutta, kun asennetaan sovelluksia verkosta tai testataan omia sovelluksia matkapuhelimessa (White ja Hemphill, 2002).

3 MIDP-SOVELLUSTEN TOTEUTUS

MIDP-sovelluksesta käytetään nimitystä *midlet*, joka suoritetaan *Java*-virtuaalikoneen alaisuudessa päätelaitteella. Kannettavilla päätelaitteilla ei ole vastaavaa komentokuorta kuin pöytätietokoneilla, jolla käyttäjä voi ohjata ohjelman suoritusta. *Midlet*-ohjelmien ohjaus tapahtuu päätelaitteelle sijoitetun *AMS*-ohjelmiston avulla, joka ohjaa koko *midletin* elinkaaren toiminnot. *AMS* (Application Management Software) toimii samalla tasolla kuin laitteen muut varusohjelmistot ja tekee kiinteää yhteistyötä *Java*-virtuaalikoneen kanssa. Kaikki *midletit* periytyvät kantaluokastaan *MIDlet*, joka sijaitsee paketissa *javax.microedition.midlet.MIDlet*. *Midlettien* kantaluokka määrittelee kolme abstraktia metodia: *startApp*, *pauseApp* ja *destroyApp*, jotka täytyy ylikirjoittaa jokaisessa *midletissä*. Näiden metodien avulla toteutetaan *midletin* elinkaaren aikaiset tilasiirtymiset (Giguire, 2002). Kohdassa 3.1 tarkastellaan *midletin* elinkaarimallin toteutusta. *MIDP*-määrityksen edellyttämää tallennusjärjestelmää, jolla käsiteltävät tiedot voidaan tallentaa pysyvään muotoon, käsitellään kohdassa 3.2.

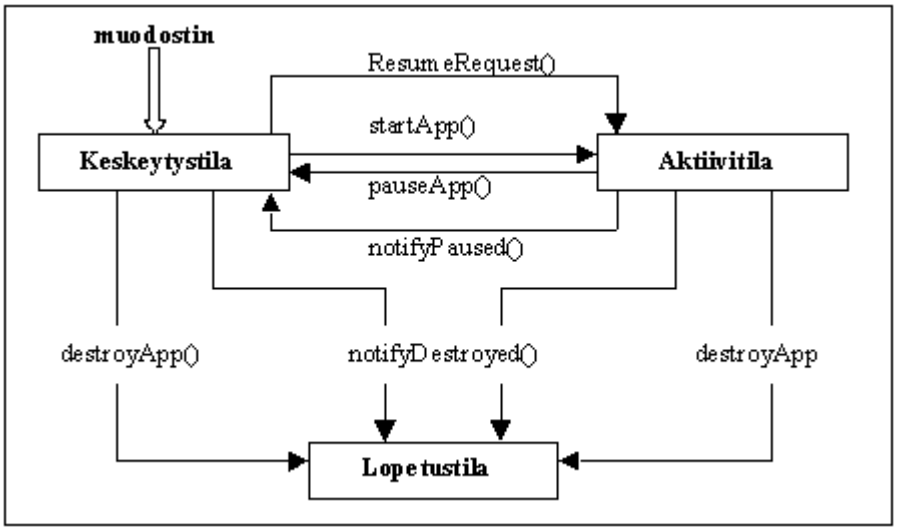
3.1 Elinkaarimalli

Midletin käynnistyspyynnön seurauksena päätelaitteen *AMS*-ohjelmisto kutsuu *midletin* muodostinta. *Midlettiin* voidaan toteuttaa parametriton muodostin, jossa suoritetaan muuttujien alustuksia. Jos muodostinta ei toteuteta, *Java*-kääntäjä suorittaa parametrittoman oletusmuodostimen. Onnistuneen muodostimen suorituksen jälkeen kutsutaan *midletin startApp*-metodia, johon toteutettu toiminnallisuus valmistaa *midletin* käyttökuntoon ja se on valmis vastaanottamaan käyttäjän komentoja sekä syötteitä. *Midletti* voi elinaikanaan olla yhdessä kolmesta mahdollisesta tilasta (Topley, 2002):

- *Keskeytystila* (paused): *Midletti* on keskeytystilassa, kun se on juuri käynnistetty eikä ole vielä suorittanut *startApp*-metodia, joka siirtää sen aktiivitilaan. Lisäksi *midletti* voi siirtyä aktiivitilasta keskeytystilaan, kun käyttäjä tai laite tarvitsee *midletin* käyttämiä resursseja toisen sovelluksen käyttöön. Keskeytystilassa ollessaan *midletin* käyttämien resurssien määrän tulee olla mahdollisimman pieni.

- *Aktiivitila* (active): Aktiivitilassa *midletti* on normaalissa toimintatilassa ja sillä on tarvittavat resurssit optimaalista suoritusta varten. Aktiivitilaan *midletti* siirtyy *startApp*-metodin suorituksen seurauksena.
- *Lopetustila* (destroyed): Lopetustilaan *midletti* voi siirtyä sekä keskeytystilasta että aktiivitilasta. Lopetustilasta *midletti* ei voi siirtyä enää muihin tiloihin. Ennen lopetustilaan siirtymistä *midletin* tulee vapauttaa kaikki varaamansa resurssit ja tallentaa tarvittavat tiedot *RMS*-tietuevarastoon.

Midletin tilasiirtyminen saa yleensä alkunsa päätelaitteen *AMS*-ohjelmiston kutsuessa jotakin seuraavista metodeista: *startApp*, *pauseApp* tai *destroyApp*. Lisäksi *midlett*i voi itse pyytää tilasiirtymistä seuraavien metodien avulla: *resumeRequest*, *notifyPaused* tai *notifyDestroyed*. Kuva 3.1 esittää *midletin* mahdolliset tilat ja sallitut tilasiirtymiset.



Kuva 3.1: *Midletin* tilat ja sallitut tilasiirtymiset (Topley, 2002).

Midletti voi siirtyä keskeytystilasta aktiivitilaan joko päätelaitteen *AMS*-ohjelmiston kutsuessa *startApp*-metodia tai epäsuorasti *midlet*-ohjelmiston kutsuessa *resumeRequest* -metodia. Tällä metodilla *midletti* voi ilmoittaa *AMS*-ohjelmistolle, että sillä on tarve siirtyä aktiivitilaan. *AMS*-ohjelmisto reakoi tähän ilmoitukseen kutsumalla *startApp*-metodia. *Midletin* normaalia suoritusta varten resurssit voidaan varata joko muodostimessa tai *startApp*-metodissa. Topleyn (2002) mukaan sellaiset resurssien varaukset, jotka suoritetaan *midletin* elinkaaren aikana ainoastaan kerran, tulisi sijoittaa muodostimeen. *Midletin startApp*-metodi voidaan suorittaa elinkaarensa aikana useita kertoja:

aina *midlettiä* käynnistettäessä ja aina kun *midletti* siirtyy keskeytystilasta aktiivitilaan. Täten yleisenä periaatteena voidaan pitää, että sijoitetaan vain sellaiset resurssien varaukset *startApp*-metodiin, jotka vapautetaan siirryttäessä aktiivitilasta pysäytystilaan.

Resurssien varaus voi aiheuttaa virhetilanteen, joka estää *startApp*-metodin onnistuneen loppuun suorituksen. Virhetilanteiden käsittelyyn täytyy kiinnittää erityistä huomiota, koska kiinnisaamattomien virhetapausten käyttäytyminen on *mildet*-tasolla määrittelemätön. Virhetilanne voi estää *startApp*-metodin loppuun suorituksen kahdella tavalla: hallitusti tai hallitsemattomasti (kuva 3.2). Hallitun virhetilanteen saattaa aiheuttaa esimerkiksi resurssien puute, jolloin ohjelma voi heittää *MidletStateChangeException* poikkeuksen ja yrittää resurssien varausta myöhemmin uudelleen. Hallitsemattoman virhetilanteen voi aiheuttaa odottamaton ajonaikainen virhe, jota ei saada *startApp*-metodissa kiinni. Tällöin *midletti* tulee siirtää lopetustilaan (Topley, 2002).

```
public void startApp() throws MIDletStateChangeException
  {
  ...
  try {
    // Muodostettaan yhteys palvelimelle.
    c = c.(HttpConnection)Connector.open(url);
    ...
    }
  catch (IOException ioe) {
    // Hallittu virhetilanne,
    // yhteyden muodostus ei onnistu.
    throw new MIDletStateChangeException("Yhteys ei saataville");
    }
  catch (Exception e) {
    // Hallitsematon virhetilanne,
    // vapautetaan resurssit.
    destroyApp(true);
    // Siirretään midletti lopetustilaan.
    notifyDestroyed();
    }
  }
```

Kuva 3.2: Poikkeusten käsittely *startApp*-metodissa.

Päätelaite saattaa normaalin toiminnan aikana joutua tilanteeseen, jossa sen täytyy keskeyttää tilapäisesti joitakin ohjelmia saadakseen resursseja korkeamman prioriteetin omaavan sovelluksen käyttöön. Tällainen tilanne voi syntyä esimerkiksi silloin, kun laitteen muistin määrä ei riitä kaikkien ohjelmien suorituksen ylläpitoon tai laitteelle saapuva puhelu tarvitsee näytön käyttöönsä. Vapauttaakseen resursseja muiden sovellusten käyttöön *AMS*-ohjelmisto voi siirtää *midletin* väliaikaisesti keskeytystilaan ja kutsua *midlet*-ohjelmistossa toteutettua *pauseApp*-metodia. Tämän metodin toteutuksesta

riippuu *midletin* reakointi tilasiirtymiseen. Mutta yleensä *midletin* tulee vapauttaa varaamansa resurssit siirtyessään keskeytystilaan. Keskeytystilasta aktiivitilaan siirtyminen tulisi toteuttaa *midlet*-ohjelman *startApp*-metodissa siten, että palautetaan käyttöliittymä ja sen sisäinen tila samaan tilaan kuin mitä se oli ennen keskeytystilaan siirtymistä. Tämä on parempi vaihtoehto kuin näyttää aloitusnäyttö. Keskeytystilassa päätelaitteen näyttö ei ole *midletin* käytössä (Topley, 2002).

Midlet-ohjelma voi myös pyytää tilamuutosta keskeytystilaan kutsumalla *notifyPaused*-metodia. Tällöin *midletti* ilmoittaa päätelaitteen *AMS*-ohjelmistolle, että se on vapauttanut resurssinsa ja on halukas siirtymään keskeytystilaan. Kun *midlet*-ohjelmassa kutsutaan *notifyPaused*-metodia, niin *AMS*-ohjelmisto ei kutsu *pauseApp*-metodia automaattisesti. Tämän vuoksi ohjelmassa on toteutettava resurssien vapautukset ja mahdolliset tietojen tallennukset *RMS*-tietuevarastoon tai vaihtoehtoisesti kutsuttava ohjelmasta käsin *pauseApp*-metodia, jossa on toteutettu tarvittavat resurssien vapautukset ja tietojen tallennukset (Topley, 2002).

Midletti voi siirtyä lopetustilaan sekä aktiivitilasta että keskeytystilasta. *Midletin* suorituksen lopetuspyynnön seurauksena päätelaitteen *AMS*-ohjelmisto kutsuu *midlet*-ohjelmaan toteutettua *destroyApp(boolean unconditional)*-metodia. Tähän metodiin toteutettu toiminnallisuus vastaa siitä, että sovellus vapauttaa kaikki varaamansa resurssit ja tallentaa tarvittavat tiedot *RMS*-tietuevarastoon ennen sovelluksen alasajoa. *Midlet*-sovellus voidaan siirtää lopetustilaan myös epäsuorasti kutsumalla ohjelmassa *notifyDestroyed*-metodia. Tämän metodin kutsun seurauksena päätelaitteen *AMS*-ohjelma saa tiedon, että suorituksessa oleva *midlet*-ohjelma on saanut tehtävänsä päätökseen, vapauttanut resurssinsa, tallentanut tarvittavat tietonsa ja on valmis siirtymään lopetustilaan. Siirrettäessä *midlet*-ohjelma lopetustilaan edellä kuvatulla tavalla täytyy ottaa huomioon se, että *notifyDestroyed*-metodin kutsun seurauksena *AMS*-ohjelma ei kutsu automaattisesti *destroyApp*-metodia. Tästä syystä resurssien vapautuksesta täytyy huolehtia ennen *notifyDestroyed*-metodin suoritusta. Yleensä tämä tehdään siten, että kutsutaan *midlet*-ohjelmaan toteutettua *destroyApp*-metodia ennen kuin kutsutaan *notifyDestroyed*-metodia (Topley, 2002).

Metodille *destroyApp(boolean unconditional)* välitetään *boolean*-tyyppiä oleva parametri. Tämän parametrin avulla voidaan määritellä *destroyApp*-metodin kutsun ehdollisuus. Jos metodia kutsutaan parametrin arvolla *true*, niin kutsu on ehdoton. Tällöin sovelluksella ei ole muuta mahdollisuutta kuin vapauttaa resurssinsa ja valmistautua siirtymään lopetustilaan. Kutsumalla metodia ehdollisena, parametrin arvolla *false*, annetaan sovellukselle mahdollisuus heittää *MidletStateChangeException-poikkeus*, jolloin sovellus voi estää tilamuutoksen lopetustilaan. Tällainen tilanne voi olla esimerkiksi, kun ohjelman suoritus on sellaisessa tilanteessa, ettei sitä ole järkevää heti siirtää lopetustilaan. Kuvassa 3.3 esitetään kuinka voidaan tällä tekniikalla toteuttaa ehdollinen *midlet*-sovelluksen siirto lopetustilaan (Topley, 2002).

```
try {
  // Resurssien vapautus.
  destroyApp(false);

  // Siirretään midletti lopetustilaan.
  notifyDestroyed();
  }
catch (MIDletStateChangeException ex) {
  // Poikkeuksen käsittely ja
  // midletti jatkaa suoritustaan.
  }
```

Kuva 3.3: *Midletin* ehdollinen siirto lopetustilaan.

Kuvan 3.3 koodissa kutsutaan aluksi *midlet*-ohjelmassa toteutettua *destroyApp*-metodia ehdollisena. Tähän metodiin on toteutettu resurssien vapautus ja tietojen tallennus *RMS*-tietuevarastoon. Lisäksi sen tulee sisältää otsikossaan *throws*-lause, joka siirtää mahdollisesti tapahtuvan poikkeuksen käsiteltäväksi kutsuvan ohjelman *catch*-lohkossa. Jos *destroyApp*-metodin suoritus ei aiheuta poikkeusta, niin ohjelman suoritus jatkuu kuvan 3.3 koodissa *try*-lohkossa ja kutsutaan *notifyDestroyed*-metodia, joka siirtää *midletin* lopetustilaan. Poikkeuksen tapahtuessa *destroyApp*-metodissa sen suoritus lopetetaan ja ohjelman suoritus siirtyy kutsuvan ohjelman *catch*-lohkoon. Tässä lohkossa poikkeus käsitellään ja *midlet*-ohjelman sallitaan jatkaa suoritustaan.

3.2 Tallennusjärjestelmä

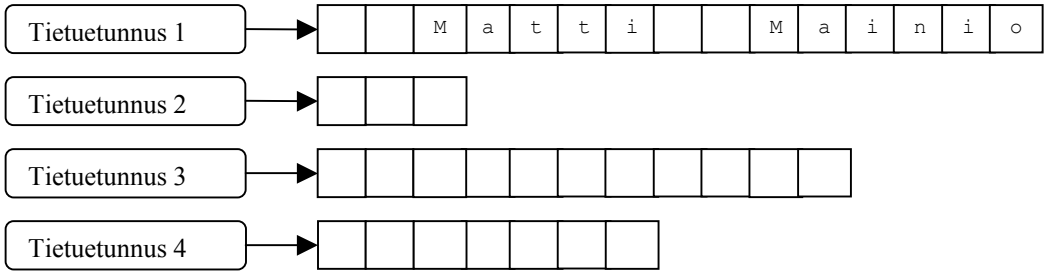
MIDP-määritys edellyttää laite- ja ohjelmistovalmistajilta, että toteutuksen on tuettava *RMS*-tietuevarastoa. *RMS*-tietuevarasto muodostuu luokkakirjastojen joukosta, joiden avulla voidaan tallentaa tietoja päätelaitteella olevaan paikalliseen tallennusjärjestelmään, josta ne voidaan hakea myöhemmin uudelleen esille. Alakohdassa 3.2.1 tarkastellaan *RMS*-tietuevaraston rakennetta ja alakohdassa 3.2.2 esitellään menetelmiä *RMS*-tietuevaraston hallintaan. Alakohdassa 3.2.3 esitellään *RMS API*:n eli *RMS*-sovellusrajapinnan *RecordEnumeration-rajapinta*, jonka avulla *RMS*-tietuevarasto voidaan käydä läpi halutussa järjestyksessä.

3.2.1 RMS-tietuevaraston rakenne

RMS:ssä tieto tallennetaan tietueina ja tietuevarasto on tietueiden kokoelma. Tietue on periaatteessa mielivaltaista tyyppiä oleva tavutaulukko. Tavutaulukon koko voi vaihdella tietueittain. *RMS* ei aseta rajoituksia sille millaista tietoa tallennetaan tietueisiin. Jokaiselle tietueelle määritellään yksilöllinen tunniste (recordID), joka on voimassa koko tietuevaraston elinajan. Vaikka tietue poistetaan, niin sen tunnistetta ei uudelleen käytetä uusille lisättäville tietueille. Tämän ansiosta yksittäisiä tietueita on helppo jäljittää. Tunniste toimii tietueiden perusavaimena (primary key), jonka avulla tietueita voidaan muuttaa ja poistaa. Tunnisteiden tietotyyppi on *integer*, ja ensimmäiselle luotavalle tietueelle annetaan arvo yksi. Aina kun tietueita lisätään tietuevarastoon, tunnisteen arvoa kasvatetaan yhdellä (White ja Hemphill, 2002).

MIDlet-sovellus ei voi suoraan olla yhteydessä laitteen paikalliseen *RMS*-tietuevarastoon, vaan *RMS API* muodostaa tarvittavan rajapinnan. Saman *jar*-pakkauksen (MIDlet suite) sisällä olevat *MIDlet*-sovellukset voivat käyttää pakkauksessa olevia *RMS*-tietuevarastoja, mutta toisen *jar*-pakkauksen tietuevarastoihin *MIDlet*-sovelluksen näkyvyysalue ei ulotu. Kun *MIDlet suite* poistetaan päätelaitteelta, niin kaikki siihen yhdistetyt *RMS*-tietuevarastot poistetaan myös laitteelta. Jokaisella *RMS*-tietuevarastolla täytyy olla yksilöllinen nimi, joka saa sisältää enintään 32 merkkiä. *RMS*-tietuevarastoilla, jotka eivät kuulu samaan *jar*-pakkaukseen, voi olla sama nimi (Mahmoud, 2000a). Kuvas-

sa 3.4 kuvataan paikallisen *RMS*-tietuevaraston rakenne, joka tässä esimerkkitapauksessa koostuu neljästä eri pituisesta tavutaulukosta.



Kuva 3.4: *RMS*-tietuevaraston rakenne

3.2.2 RMS-tietuevaraston hallinta

RMS-tietuevaraston toteuttavat luokat ja rajapinnat sijaitsevat *javax.microedition.rms*-paketissa. *RMS*-tietuevarasto koostuu yhdestä luokasta *RecordStore* ja rajapinnoista *RecordComparator*, *RecordEnumeration*, *RecordFilter* ja *RecordListener*. Lisäksi pakettiin kuuluu poikkeusten käsittelyä varten viisi rajapintaluokkaa. Luokan *RecordStore* metodeilla luodaan, suljetaan ja poistetaan *RMS*-tietuevarasto. Uuden *RMS*-tietuevaraston luonti ja olemassa olevan tietuevaraston avaus tehdään samalla *RecordStore*-luokan metodilla *openRecordStore* seuraavasti:

RecordStore rs = RecordStore.openRecordStore(nimi, true).

Metodille välitetään kaksi parametria. Ensimmäinen parametri kertoo luotavan tai avattavan paikallisen *RMS*-tietuevaraston nimen, joka saa sisältää enintään 32 merkkiä. Toisen parametrin arvolla *true* pyydetään metodia luomaan uusi *RMS*-tietuevarasto, jos ensimmäisen parametrin nimellä tietuevarastoa ei ole olemassa. Käytettäessä metodia olemassaolevan *RMS*-tietuevaraston avaamiseen parametrin arvoksi asetetaan *false*. Tällöin on käytettävissä poikkeusten käsittelyluokka *RecordStoreNotFoundException*, jossa poikkeus voidaan käsitellä tilanteen vaatimalla tavalla. Paikallinen *RMS*-tietuevarasto suljetaan metodilla *closeRecordStore*. Tämän metodin käytössä on huomioitava se,

että sitä on kutsuttava yhtä monta kertaa kuin metodia, jolla *RMS*-tietovarasto avattiin. Muutoin tietovarasto jää avoimeksi (Mahmoud, 2000a).

Järjestelmä ylläpitää tietoa *RMS*-tietuevarastoon suoritetuista toiminnoista. Yksi tällainen ylläpidettävä tieto on paikallisen *RMS*-tietuevaraston versionumero. Tätä *integer*-tyyppistä arvoa kasvatetaan aina kun *RMS*-tietuevaraston sisältöön tehdään muutoksia. Laitevalmistaja määrittää alkuversionumeron, mutta sen täytyy olla suurempi kuin nolla. *RMS*-tietuevaraston versionumeron lisäksi järjestelmä ylläpitää päiväystietoa, jota päivitetään aina kun *RMS*-tietuevaraston sisältöä muutetaan. Sovellus voi käyttää näitä tietoja hyväkseen, kun se tarvitsee tietoa *RMS*-tietuevarastoon tehdyistä muutoksista (White ja Hemphill, 2002).

```
RecordStore rs = RecordStore.openRecordStore(nimi, false);
byte[] tietue;  //tallennettava tieto tavukoodina
ind recordId;
try {
  // Tallennetaan tietue RMS-tietuevarastoon.
  recordId = rs.addRecord(tietue, 0, tietue.length);

  // Haetaan tietue esille RMS-tietuevarastosta.
  tietue = rs.getRecord(recordId);

  ... // muutetaan tietuetta

  // Kirjoitetaan tietue takaisin RMS-tietuevarastoon.
  rs.setRecord(recordId, tietue, 0, tietue.length);
  }
catch (Exception e) {
  // Käsitellään mahdolliset poikkeukset.
```

Kuva 3.5: Tietueiden käsittely *RecordStore*-luokan metodeilla.

Paikallisen *RMS*-tietuevaraston tietuejoukon käsittelyä varten *RecordStore*-luokassa määritellään neljä perusmetodia: *addRecord*, *getRecord*, *setRecord* ja *deleteRecord*. Tallentaminen suoritetaan koko tietue kerrallaan tavutaulukkona *addRecord*-metodilla. Metodi palauttaa uuden luodun tietueen tunnisteen. Tieto haetaan esille *RMS*-tietuevarastosta *getRecord*-metodilla, jolle välitetään parametrina halutun tietueen tunniste. Metodilla *setRecord* voidaan korvata olemassaoleva tietue uudella tietueella. *RMS*-tietuevarastossa ei ole mahdollista muuttaa tietystä tietueesta vain sen tiettyä osaa (Mahmoud, 2000a). Tietueen muuttaminen suoritetaan siten, että ensin luetaan koko tietue *RMS*-tietuevarastosta, tehdään muutokset siihen ja lopuksi tietue kirjoitetaan *RMS*-tietuevarastoon vanhan tietueen päälle *setRecord* -metodilla kuvan 3.5 mukaisesti.

Tietue poistetaan *RMS*-tietuevarastosta *deleteRecord*-metodilla, jolle välitetään parametrina tietueen tunniste. *RMS*-tietuevarasto ei sisällä menetelmää, millä saataisiin selville yksittäisen tietueen tunniste tietuevarastossa. Mahmoud (2000a) esittää menetelmän, jossa tietueen tunniste viedään aina vektoritauluun, kun tietue lisätään tietuevarastoon. Kun halutaan poistaa tietty tietue *RMS*-tietovarastosta, niin etsitään ensin *vektori*taulusta sopiva tunniste poistettavalle tietuelle. Kuvassa 3.6 kuvataan tämän menetelmän soveltamista mukaillen Mahmoudia (2000a).

```
// recordIDs sisältää tietueiden tunnisteet.
Enumeration IDs = recordIDs.elements();
While (IDs.hasMoreElements())
  {
  int id = ((Integer) IDs.nextElement()).intValue();
  // Suoritetaan vertailu.
  if (poistettava.compareTo(rs.getRecord(id) = 0)
    {
    // Poistetaan kyseinen tietue.
    rs.deleteRecord(id);
    }
  }
```

Kuva 3.6: Tietueen poisto *RMS*-tietuevarastosta.

MIDP-määrityksen mukaan paikalliselle *RMS*-tietuevarastolle täytyy olla vähintään 8 kilotavua muistia käytettävissään. Käytettävissä oleva muistin määrä vaihtelee laitteittain, mutta yleensä *RMS*-tietuevarastolla on enemmän muistia käytettävissään kuin vähimmäisvaatimus 8 kilotavua (Knutsen, 2002). *MIDlet*-sovelluksen toteutuksessa täytyy varautua *RMS*-tietuevaraston käytön yhteydessä muistin riittävyyteen. *RecordStore*-luokan metodi *getSizeAvailable* palauttaa käytettävissä olevan muistin määrän tietylle *RMS*-tietuevarastolle. Lisäksi *RMS*-paketti sisältää poikkeusten käsittelyä varten luokan *RecordStoreFullException*, joka aiheuttaa poikkeuksen, kun paikallinen *RMS*-tietuevarasto on täynnä (White ja Hemphill, 2002).

Usein vain yhden olion muuttujan tallentaminen paikallisen *RMS*-tietuevaraston tietueeseen ei riitä, vaan samaan tietueeseen täytyy pystyä tallentamaan koko olion sen hetkinen tila. Ennen olion tietojen kirjoitusta *RMS*-tietuevarastoon olio täytyy sarjallistaa. Mekanismin perusajatus on kirjoittaa olion tila *RMS*-tietuevarastoon sellaisessa muodossa, että olio voidaan myöhemmin lukea *RMS*-tietuevarastosta ja luoda tietuevarastoon kirjoitettua vastaava olio. *MIDP* ei tue *Javan* sarjallistamismekanismia, joten olioiden sarjallistaminen täytyy kirjoittaa alusta alkaen itse. Tämä voidaan toteuttaa esimer-

kiksi tekemällä oliolle metodit, joilla olio kirjoittaa itsensä esimerkiksi merkkivirtaan ja lukee itsensä merkkivirrasta (Sun Microsystems, 2003).

```
public class Toimittaja
  {
  ...
  public void serialize(DataOutputStream dataStream)
    throws IOException
    {
    // Kirjoitetaan olion jäsenmuuttjat tulostusvirtaan.
    dataStream.writeUTF(numero);
    dataStream.writeUTF(nimi);
    dataStream.writeUTF(osoite);
    }

  public Toimittaja deserialize(DataInputStream dataStream)
    throws IOException
    {
    Toimittaja toimittaja = new Toimittaja();
    // Luetaan jäsenmuuttujat syötevirrasta.
    toimittaja.numero = dataStream.readUTF();
    toimittaja.nimi = dataStream.readUTF();
    toimittaja.osoite = dataStream.readUTF();
    return toimittaja;
    }
  }
```

Kuva 3.7: Olion tilan sarjallistaminen ja sarjallistamisen purku.

Kuvassa 3.7 kuvataan *Toimittaja*-luokka, jonka jäsenmuuttujat tallennetaan *RMS*-tietuevarastossa samaan tietueeseen. Tästä syystä olion tiedot täytyy pystyä sarjallistamaan, jotta ne voidaan myöhemmin palauttaa samassa muodossa ja muodostaa vastaava olio. Sarjallistamista varten luokkaan on toteutettu metodi *serialize*, jonka avulla luokan jäsenmuuttujat kirjoitetaan *DataOutputStream*-tyyppisen olion avulla tulostusvirtaan. Olion uudelleenluomiseksi luokkaan on toteutettu metodi *deserialize*, joka lukee tiedot olion jäsenmuuttujiin *DataInputStream*-tyyppisestä syötevirrasta.

Sarjallistetun olion tiedot tallennetaan *RMS*-tietuevarastoon *RecordStore*-luokan metodilla *addRecord*. Metodille välitetään parametrina tavutaulukko, joka sisältää tallennettavan olion kaikki tiedot. Kuvassa 3.8 kuvataan metodi, joka tallentaa sarjallistetun olion tiedot *RMS*-tietuevarastoon. Metodille välitetään parametrina tallennettava olio. Aluksi metodissa luodaan *ByteArrayOutputStream*-tyyppinen olio, johon muodostetaan *DataOutputStream*-tyyppinen tulostusvirta. Seuraavaksi tallennettava olio kirjoittaa itsensä tämän tulostusvirran avulla luotuun ByteArrayOutputStream-tyyppiseen olioon.

```
public int lisaaTietue(Toimittaja toimittaja)
    throws IOException, RecordStoreNotOpenException,
           RecordStoreException, RecordStoreFullException
    {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    // Muodostetaan tulostusvirta ByteArrayOutputStream-olioon.
    DataOutputStream dataStream  = new DataOutputStream(stream);
    // Sarjallistetaan Toimittaja-tyyppinen olio.
    toimittaja.serialize(dataStream);
    dataStream.close();
    // Palutetaan tietuetunnus.
    return recordStore.addRecord(stream.toByteArray(), 0, stream.size());
    }
```

Kuva 3.8: Sarjallistetun olion tietojen tallennus *RMS*-tietuevarastoon.

Tietue luetaan *RMS*-tietuevarastosta *RecordStore*-luokan metodilla *getRecord*, jolle välitetään parametrina haettavan tietueen tietuetunnus. Jos haettavaan tietueeseen on tallennettu koko olion tiedot, niin tietueesta täytyy palautettaessa muodostaa tallennettua vastaava olio.

```
public Toimittaja haeTietue(int tunnus)
    throws IOException, RecordStoreNotOpenException,
           InvalidRecordIDException, RecordStoreException
    {
    byte [] data = recordStore.getRecord(tunnus);
    // Muodostetaan ByteArrayInputStream-olio.
    ByteArrayInputStream stream = new ByteArrayInputStream(data);
    // Muodostetaan lukuvirta ByteArrayOutputStream-oliosta.
    DataInputStream dataStream = new DataInputStream(stream);
    // Puretaan sarjallistus ja palautetaan luotu olio.
    Toimittaja toimittaja = Toimittaja.deserialize(dataStream);
    return toimittaja;
    }
```

Kuva 3.9: Tietojen haku *RMS*-tietuevarastosta.

Kuvassa 3.9 kuvataan metodi, joka noutaa *RMS*-tietuevarastosta tietueen ja luo tietuetta vastaavan uuden olion. Metodille välitetään parametrina haettavan tietueen tietuetunnus. Aluksi luodaan *ByteArrayInputStream*-tyyppinen olio, jolle välitetään parametrina *RecordStore*-luokan metodin *getRecord* palauttama tietueen muodostava tavutaulukko. Seuraavaksi muodostetaan *DataInputStream*-syötevirta *ByteArrayInputStream*-tyyppisestä oliosta. Tästä syötevirrasta *Toimittaja*-olio lukee itsensä siihen toteutetun metodin *deserialize* avulla.

3.2.3 RMS-tietuevaraston läpikäynti

RMS API:ssa määritellyt rajapintaluokat *RecordEnumeration*, *RecordComparator* ja *RecordFilter* mahdollistavat paikallisen *RMS*-tietuevaraston läpikäynnin molempiin suuntiin. Rajapintojen avulla voidaan hakea *RMS*-tietuevarastosta haluttu joukko tietueita halutussa järjestyksessä. Rajapinnan *RecordEnumeration* kautta hallittava olio saadaan käyttöön kutsumalla *RecordStore*-luokan metodia *enumerateRecords* seuraavasti (White ja Hemphill, 2002):

```
RecordEnumeration re = rs.enumerateRecords(null, null, false).
```

Metodille välitetään kolme parametria. Ensimmäinen parametri on *RecordFilter*-rajapinnan toteuttava olio. Tämän olion avulla voidaan määritellä haluttuja kriteerejä, jotka täyttävät tietueet *RMS*-tietuevarastosta otetaan mukaan läpikäytävään joukkoon. Jos parametrille annetaan arvoksi *null*, niin kaikki tietueet *RMS*-tietuevarastosta kelpuutetaan joukkoon. *RecordFilter*-rajapinta määrittelee vain metodin *matches*, jonka rajapinnan toteuttavan luokan täytyy sisältää (White ja Hemphill, 2002).

Toinen parametri on viittaus *RecordComparator*-rajapinnan toteuttavaan olioon. Tämän olion avulla voidaan määrittää lajittelujärjestys läpikäytävälle tietuejoukolle. Jos parametrille annetaan arvoksi *null*, niin tietueet läpikäytävässä joukossa ovat määrittelemättömässä järjestyksessä. Tietuetunnus ei määrittele mitenkään joukon järjestystä. *RecordComparator*-rajapinta määrittelee vain yhden metodin *compare*, joka täytyy toteuttaa rajapinnan toteuttavaan luokkaan

Kolmas parametri on *boolean*-tyyppiä. Parametrin avulla tietuekokoelma voidaan asettaa reagoimaan automaattisesti *RMS*-tietuevarastossa tapahtuviin muutoksiin. Jos parametrin arvon ollessa *true* *RMS*-tietuevarastossa tapahtuu muutoksia, niin tietuekokoelman sisältö päivitetään automaattisesti vastaamaan *RMS*-tietuevaraston sen hetkistä tilannetta. White'n ja Hemphillin (2002) mukaan tämä automaattinen tietuekokoelman päivitys on raskas operaatio pieniresurssisille päätelaitteille. Tästä syystä sen käytölle tulee olla hyvät perustelut.

RecordFilter-rajapinnan toteuttava olio toimii suodattimena. Kun *RecordStore*-luokan metodille *enumerateRecords* välitetään parametrina viittaus *RecordFilter*-rajapinnan toteuttavaan olioon, niin *enumerateRecords*-metodi välittää jokaisen tietueen argumenttina *RecordFilter*-rajapinnan metodille *matches*. Jos tietue täyttää *matches*-metodissa asetetut ehdot, niin se kelpuutetaan mukaan läpi käytävään joukkoon. Kuvassa 3.10 kuvataan suodattimena toimiva luokka, joka toteuttaa rajapinnan *RecordFilter*. Rajapinnan *RecordFilter* toteuttavan luokan muodostimessa annetaan suodatustekijä, joka tässä tapauksessa on osoite. Luokan metodissa *matches* jokaista *RMS*-tietuevaraston tietuetta verrataan suodatustekijään. Jos tietueella on sama osoite kuin suodatustekijälle on annettu arvoksi muodostimessa, niin *matches*-metodi palauttaa arvon *true* ja tietue kelpuutetaan mukaan läpi käytävään joukkoon (White ja Hemphill, 2002).

```
public class Filter implements RecordFilter
  {
  private String filter;

  public Filter(String filter) {
    this.filter = filter;
    }

  public boolean matches(byte[] candidate) {
    try {
      ByteArrayInputStream bais = new ByteArrayInputStream(candidate);
      DataInputStream inputStream = new DataInputStream(bais);
      String nimi  = inputStream.readUTF();
      String osoite  = inputStream.readUTF();
      return filter.equals(osoite);
      }
    catch (Exception e) {
      return true;
      }
    }
  } // end
```

RecordStore-luokan *enumerateRecords*-metodin palauttama tietuejoukko saadaan haluttuun järjestykseen *RecordComparator*-rajapinnan toteuttavan luokan avulla. Rajapinta määrittelee metodin *compare*, jolle välitetään parametreina kaksi tietuetta tavutaulukkona. Metodi palauttaa yhden kolmesta rajapinnan määrittelemistä arvoista: *EQUIVALENT*, *FOLLOWS* tai *PRECEDES*. Rajapintaa voi käyttää yksittäisten tietueiden vertailuun, mutta tärkeämpi käyttökohde on *enumerateRecords*-metodin yhdeydessä. Tällöin *RecordComparator*-rajapinnan toteuttavaa luokkaa käytetään *RMS*-tietuevarastosta haettavan tietuejoukon järjestämiseen (White ja Hemphill, 2002). Kuvassa 3.11 kuvataan *RecordComparator*-rajapinnan toteuttavan luokan *compare*-metodin toteutus, jossa tietuejoukko järjestetään henkilön nimen mukaan.

```
public int compare(byte[] tietue1, byte[] tietue2)
    {
    try {
      ByteArrayInputStream bais1 = new ByteArrayInputStream(tietue1);
      ByteArrayInputStream bais2 = new ByteArrayInputStream(tietue2);
      DataInputStream inputStream1 = new DataInputStream(bais1);
      DataInputStream inputStream2 = new DataInputStream(bais2);
      int numero1  = inputStream1.readInt();
      String nimi1  = inputStream1.readUTF();
      String numero2  = inputStream2.readInt();
      String nimi2  = inputStream2.readUTF();
      if (nimi1.compareTo(nimi2) > 0)
        return (RecordComparator.FOLLOWS);
      else if (nimi1.compareTo(nimi2) < 0)
        return (RecordComparator.PRECEDES);
      else
        return (RecordComparator.EQUIVALENT);
      }
    catch (Exception e) {
      // Käsitellään poikkeus.
      }
    }
```

Kuva 3.11: Tietueiden lajittelu.

RMS API eli *RMS*-sovellusrajapinta tarjoaa vaihtoehtoisen tavan seurata *RMS*-tietuevarastossa tapahtuvia muutoksia. *RMS*-sovellusrajapinta määrittelee *RecordListener*-rajapinnan, jonka avulla voidaan reagoida *RMS*-tietuevarastossa tapahtuviin muutoksiin (White ja Hemphill, 2002). Saman *MIDlet*-pakkauksen sisällä on mahdollista käyttää samaa *RMS*-tietuevarastoa useasta eri *MIDP*-sovelluksesta. Sovelluksen tekemistä muutoksista *RMS*-tietuevarastoon voidaan muille sovelluksille ilmoittaa kuuntelijan avulla. Mikä tahansa olio voidaan asettaa *RMS*-tietuevarastossa tapahtuvien muutosten *kuuntelijaksi*. Tämän olion täytyy toteuttaa *RecordListener*-rajapinta, jossa on määritelty kolme takaisinkutsumetodia.

```
public class RMSKuuntelija implements RecordListener
  {
  public void recordAdded(RecordStore nimi, int tunnus) {
    // Käsitellään tapahtuma
    }

  public void recordChanged(RecordStore nimi, int tunnus) {
    // Käsitellään tapahtuma
    }

  public void recordDeleted(RecordStore nimi, int tunnus) {
    // Käsitellään tapahtuma
    }
  }
```

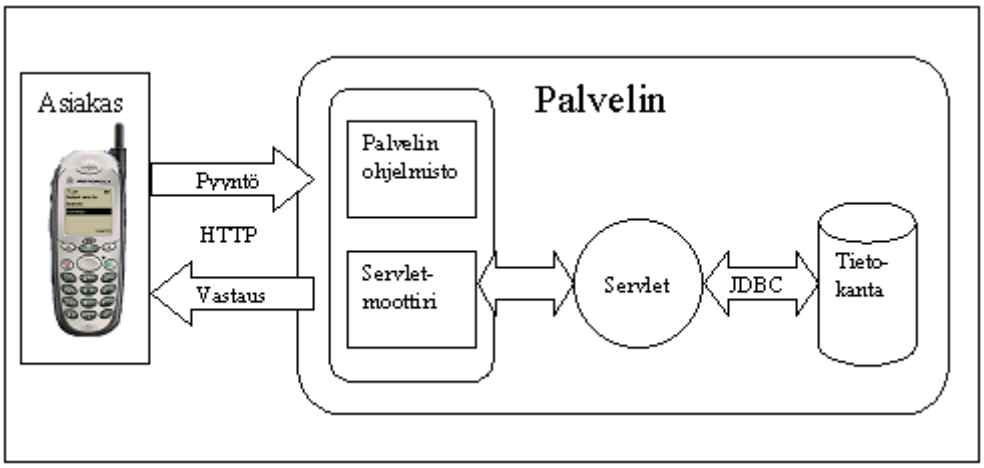
Kuva 3.12: *RMS*-tietuevaraston muutosten kuuntelija.

Kuvassa 3.12 kuvataan kuuntelijaluokka, johon on toteutettu *RecordListener*-rajapinnassa määritellyt metodit. Uuden tietueen lisäys *RMS*-tietuevarastoon aiheuttaa metodin *recordAdded* kutsun. Olemassaolevan tietueen muutos *RMS*-tietuevarastossa aiheuttaa *recordChanged*-metodin kutsun. Metodia *recordDeleted* kutsutaan, kun tietue poistetaan *RMS*-tietuevarastosta. Takaisinkutsumetodeille välitetään parametreina *RMS*-tietuevaraston nimi ja tapahtuman kohteena olevan tietueen tietuetunnus. Tapahtumien kuuntelijaluokka otetaan käyttöön *RecordStore*-luokan *addRecordListener*-metodilla, jolle välitetään parametrina viittaus kuuntelijaolioon (White ja Hemphill, 2002).

4 VERKKOYHTEYS

Kehitettäessä teknologiaa matkapuhelinten kautta käytettäviä lisäarvopalveluja varten on täytynyt ottaa huomioon verkkojen ja päätelaitteiden asettamat rajoitukset. Matkapuhelimissa ja muissa mobiilipäätelaitteissa ongelmana on ollut puhelimien suorituskyky ja käyttöliittymä, esimerkiksi pieni prosessoriteho, muistin vähyys, rajoitettu virran määrä, pienet näyttöresoluutiot ja pienen näppäimistön aiheuttamat tiedon syöttövaikeudet. Langattomien tietoverkkojen ongelmina ovat olleet tiedonsiirtonopeus ja yhteyden säilymiseen liittyvät kysymykset, kuten katvealueitten esiintyminen langattomassa verkossa (White ja Hemphill, 2002).

Asiakas-palvelin-sovelluksella tarkoitetaan toteutusta, jossa päätelaitteelle sijoitettu asiakasohjelmisto ottaa yhteyttä palvelinohjelmistoon, lähettää palvelinohjelmistolle palvelupyynnön, vastaanottaa palvelimen palauttaman tiedon ja tulostaa sen päätelaitteen näytölle. Palvelinkoneessa käynnistettävän palvelinohjelmiston tarkoituksena on vastata asiakasohjelmiston palvelupyyntöön, noutaa tietokannasta tarvittavia tietoja ja palauttaa asiakasohjelmiston haluama tieto päätelaitteelle (kuva 4.1). *CGI*-ohjelmointi oli pitkään lähes ainoa tapa tehdä sovelluksia www-palvelimille. Viime vuosina *Java-servletit* ovat tulleet varteen otettaviksi vaihtoehdoiksi perinteisille *CGI*-ohjelmille.

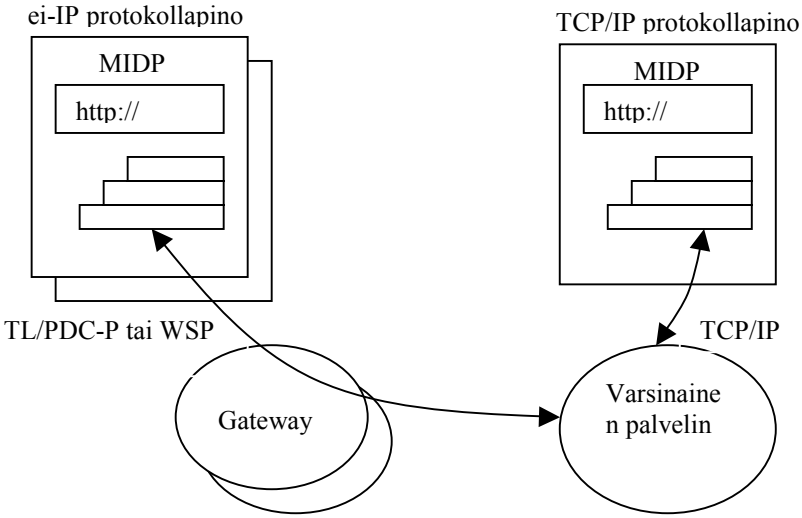


Kuva 4.1: Tietoliikennearkkitehtuuri (Mahmoud, 2001b).

Kohdassa 4.1 käsitellään verkkoyhteyden muodostusta *GCF*-kehyksen avulla ja kohdassa 4.2 käsitellään tutkielman tavoitteiden mukaiset yhteyskäytännöt toteuttajan näkökulmasta.

4.1 Verkkoyhteyden toteutus

MIDP-määritys edellyttää, että kaikki *MIDP*-toteutukset tukevat *HTTP*-yhteyskäytäntöä. Tämän vuoksi kaikkien *MIDP*-sovellusten on pystyttävä kommunikoimaan verkossa olevien laitteiden kanssa vähintään *HTTP*-pyyntöjä käyttäen. Koska *J2ME*:n käyttöympäristö on tavallisesti muistiltaan ja tallennuskapasiteetiltaan rajallinen, ohjelmat voivat siirtää raskaat tehtävät palvelimen hoidettavaksi. Luonnollisesti ohjelman pitää kommunikoida palvelimen kanssa ja tähän *HTTP* soveltuu hyvin. *HTTP*-yhteyksien avulla voidaan noutaa verkkosivuja ja kutsua palvelinpuolen ohjelmia, kuten *Java-servlettejä* tai *JSP*-sovelmia. *MIDP*-sovellukset voivat käyttää *HTTP*-yhteyksiä *HttpConnection*-rajapinnan kautta.

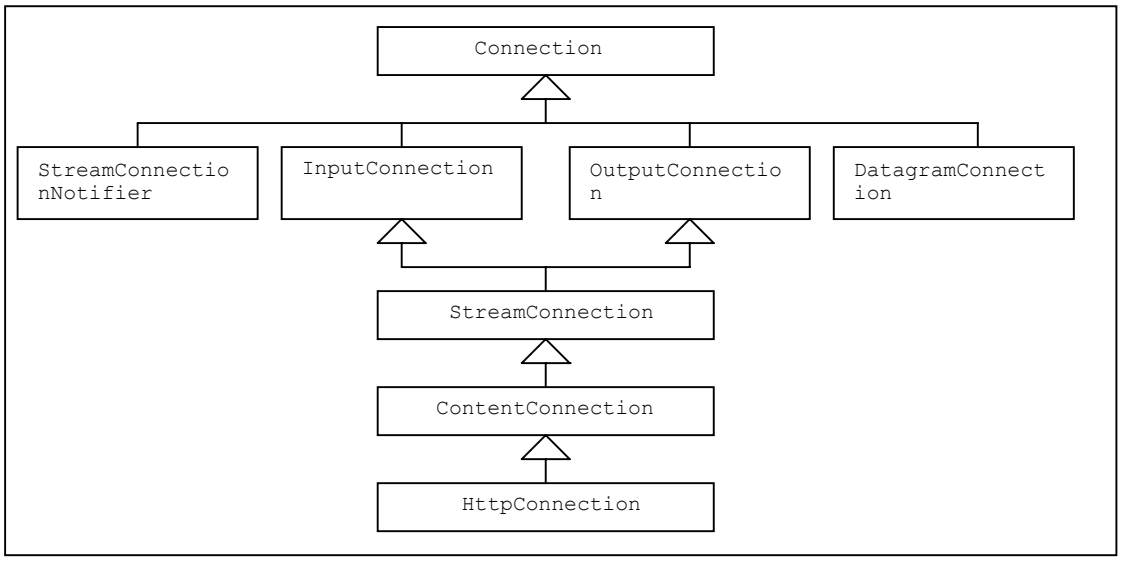


Kuva 4.2: *HTTP*:n toteutus *MIDP*-ympäristössä (Sun MicroSystems, 2000b).

Kuvassa 4.2 kuvataan *HTTP*-tuen toteutus *MIDP*-toteutuksissa. *HTTP*-tuki voidaan toteuttaa joko käyttäen *TCP/IP*-pohjaista ratkaisua tai sitten jotakin *IP*-yhteyskäytäntöön pohjautumatonta teknologiaa, kuten esimerkiksi *WAP* tai *I-Mode*. *WAP*-arkkitehtuuri vaatii päätelaitteen ja varsinaisen palvelimen väliin välityspalvelimen (gateway), jonka

yhdyskäytävä huolehtii kommunikoinnista *WWW*-ympäristön ja päätelaitteen välillä. Laitevalmistaja voi tarjota tuen myös esimerkiksi *UDP*:lle, mutta se ei takaa viestien perillemenoa (Sun MicroSystems, 2000b).

GCF (Generic Connection FrameWork) muodostaa perustan tietoliikennetoteutuksille *MIDP*-ohjelmoinnissa. *GCF* sijaitsee paketissa *javax.microedition.io* ja koostuu yhdestä luokasta (Connector) ja useista rajapinnoista, jotka yhdessä määrittelevät *GCF*:n. Kuvassa 4.3 esitetään rajapintojen suhteet. Rajapintahierarkiassa ylimpänä on *Connection*, josta muut periytyvät. Kaikissa tapauksissa yhteydet luodaan käyttämällä *Connector*-luokan *open*-metodia (Mahmoud, 2000b). Tässä tutkielmassa tarkastellaan *HTTP*-rajapintaa, joka on osa *javax.microedition.io*-pakkausta ja sisältää kaikki tarvittavat metodit *HTTP*-yhteyden hallitsemiseen.



Kuva 4.3: *GCF*:n muodostavat rajapinnat (Mahmoud, 2000b).

Seuraavaksi tarkastellaan *MIDP*-sovelluksen avulla yhteyden muodostamista palvelimelle ja pyynnön lähettämistä sinne sekä vastauksen vastaanottamista. Yhteyden muodostamiseen käytetään *Connector*-luokassa määriteltyä *open*-metodia, jolle välitetään parametrina *URL*-osoite (kuva 4.4, rivi 2). Metodilla *setRequestMethot* määritellään käytettävä pyyntötyyppi, joko *GET* tai *POST* (kuva 4.4, rivi 4). Seuraavaksi asetetaan tarvittavat *HTTP*-otsakkeet. *MIDP*-määritys edellyttää ainakin otsakkeiden *User-Agent* ja *Content-Language* määrittämistä (kuva 4.4, rivi 6 ja 7). Giguere (2001) suosittelee tietyissä tapauksissa *Connection*-otsakkeen asettamista. Asettamalla *Connection*-otsake

voidaan varmistaa, että palvelin lopettaa yhteyden heti vastauksen lähettämisen jälkeen. Toisaalta tehtäessä useita pyyntöjä samalle palvelimelle lyhyen ajan sisällä, saadaan parempi tehokkuus, kun ei aseteta tätä otsaketta. Mahmoudin (2001b) mukaan käytettäessä palvelimella *servletin* metodia *ServletRequest.getParameter(String nimi)* vastaanottamaan asiakkaalta tulevaa pyyntöä täytyy asettaa myös *CONTENT-TYPE*-otsake (kuva 4.4, rivi 8). Seuraavaksi määritellään syöttö- ja tulostusvirrat sekä lähetetään pyyntö ja otetaan vastaan palvelimen vastaus.

```
 1 // Muodostetaan yhteys palvelimelle
 2 c = (HttpConnection) Connector.open(url);
 3 // Asetetaan pyyntötyyppi.
 4 c.setRequestMethod(HttpConnection.POST);
 5 // Asetetaan HTTP-otsakekentät.
 6 c.setRequestProperty("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");
 7 c.setRequestProperty("Content-Language", "en-CA");
 8 c.setRequestProperty("CONTENT-TYPE", "application/x-www-form-encoded");
 9 // Muodostetaan tulostusvirta-
10 os = c.openOutputStream();
11 // Muodostetaan syötevirta.
12 is = c.openDataInputStream();
13 // Lähetetään pyyntö ja luetaan vastaus.
```

Kuva 4.4: Yhteyden muodostaminen palvelimelle.

Tiedon lukemista palvelimelta voidaan tehostaa käyttämällä puskurointia, tällöin syötevirrasta siirretään kerralla enemmän tietoa muistiin. Syötevirran lukeminen yksi tavu kerrallaan on tehoton toimenpide. *MIDP*-määritys ei sisällä valmista puskuroitua virtojen käsittelyä. Kuvassa 4.5 kuvataan Yuan'n (2003) esittämä tapa lukea syötevirta puskuroidusti *MIDP*-sovelluksissa. Menetelmässä syötevirrasta luetaan tietty määrä tavuja kerralla aina *ByteArrayOutputStream*-tyyppiseen olioon.

```
// Muodosteaan yhteys palvelimelle
HttpConnectin conn = (HttpConnection) Connector.open(url);
Conn.setRequestMethod(HttpConnection.GET);
DataInputStream din = conn.openDataInputStream();
ByteArrayOutputStream bos = new ByteArrayOutputStream();
Byte [] buf = new Byte[256];
// Luetaan palvelimelta syötevirta.
While (true) {
  int rd = din.read(buf, 0, 256);
  if (rd == -1)
    break;
  bos.write(buf, o, rd);
  }
bos.flush();
buf = bos.toByteArray();
```

Kuva 4.5: Syötevirran lukeminen puskuroidusti palvelimelta (Yuan, 2003).

4.2 Tiedonsiirron yhteyskäytännöt

Järjestelmien keskinäinen kommunikointi edellyttää tietojärjestelmien välistä yhteistoimintaa hajautetussa ympäristössä. Eri koneilla sijaitsevien sovellusten yhteistoimintaa varten tarvitaan tietoliikenneyhteyksien lisäksi menetelmiä sovellusten välisen vuorovaikutuksen toteuttamiseksi. *MIDP*-pohjaisissa mobiilisovelluksissa verkkoyhteyksien käyttö perustuu useimmiten *HTTP*-yhteyskäytäntöön. Tässä kohdassa esitellään tutkielman tavoitteiden mukaiset yhteyskäytännöt toteuttajan näkökulmasta. Luvussa 5 tarkastelunäkökulma muutetaan enemmän käytännön läheiseksi ja tarkastellaan yhteyskäytäntöjä esimerkkisovelluksen avulla ohjelmoijan näkökulmasta.

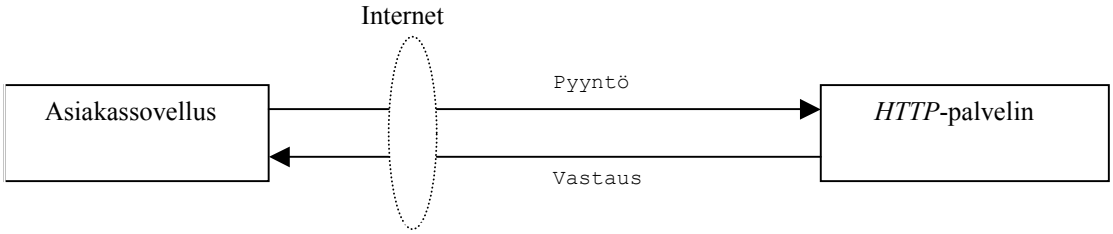
Alakohdassa 4.2.1 tarkastellaan *HTTP*-yhteyskäytännön ominaisuuksia määrityksien pohjalta. Alakohdassa 4.2.2 tarkastellaan *XML*-kielen perusteita sovellusten välisen tiedonsiirron kannalta. *XML-RPC* on *XML*-pohjainen tiedonsiirron yhteyskäytäntö ja sen perusteita käsitellään alakohdassa 4.2.3. Toinen *XML*-pohjainen tiedonsiirron yhteyskäytäntö on *SOAP* ja sen perusteita esitellään alakohdassa 4.2.4.

4.2.1 HTTP-pohjainen yhteyskäytäntö

HTTP (Hypertext Transfer Protocol) käsittelee *URL* (Uniform Resource Location) -osoitteen avulla nimettyjä resursseja. Yleisempi termi resurssiosoitteelle on *URI* (Uniform Resource Identifier). *URI* on yleinen nimeämismäärittely, joka käsittää kaikki merkkimuotoiset nimet ja osoitteet, joiden avulla viitataan erilaisiin resursseihin. *URI*-määrittely sisältää myös *URL*-osoitteet. Esimerkiksi *URL*-osoite http://www.palvelin.fi/hakemistopolku/tiedosto koostuu yhteyskäytäntöosasta (http://), palvelimen nimestä (www.palvelin.fi) ja resurssiosasta (/hakemistopolku/tiedosto) (W3C, 1999a).

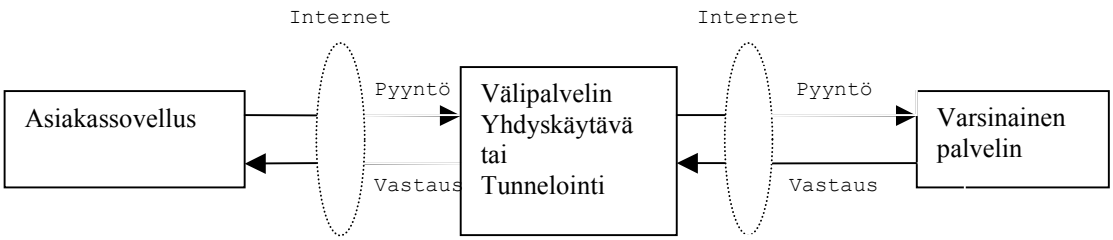
HTTP-yhteyskäytäntö on *Internet*-ohjelmoinnin keskeisimpiä verkkoyhteyskäytäntöjä, joka on samalla hyvin yksinkertainen mutta myös tehokas. *HTTP*-yhteyskäytäntö on asiakas-palvelin -mallin mukainen. Yksinkertaisimmassa tapauksessa *HTTP*-pohjainen asiakassovellus avaa yhteyden ja lähettää palvelupyynnön (request) *HTTP*-palvelimelle kuvan 4.6 mukaisesti. Tämän jälkeen palvelin palauttaa vastauksen (response), joka on yleensä se resurssi, jota asiakassovellus on palvelupyynnössään pyytänyt. Kun palvelin

on toimittanut vastauksen, se myös sulkee yhteyden. Näin ollen *HTTP* on tilaton yhteyskäytäntö (W3C, 1999a).



Kuva 4.6: *HTTP*-yhteyskäytännön mukainen pyyntö-vastaus -malli.

Kuvassa 4.7 kuvataan monimutkaisempi tapaus missä pyyntö tai vastaus kulkevat yhden tai useamman välittäjän kautta matkalla asiakassovellukselta varsinaiselle palvelimelle. Välittäjäjärjestelmästä on olemassa kolme erilaista muotoa: *välipalvelin* (proxy), *yhdyskäytävä* (gateway) ja *tunnelointi* (tunneling). *Välipalvelin* on välitysagentti, joka vastaanottaa pyyntöjä, kirjoittaa uudelleen viestin ja toimittaa uudelleenkirjoitetun viestin takaisin *URI*:sta tunnistetulle palvelimelle. *Yhdyskäytävä* on vastaanottava agentti, joka muuntaa pyynnöt *yhdyskäytävän* toisella puolella olevalle yhteyskäytännölle sopiviksi. *Tunnelointi* toimii linkkinä kahden yhteyden välillä muuttamatta viestiä mitenkään. *Tunnelointia* käytetään, kun yhteyden pitää kulkea jonkin välittäjän kautta, vaikka välittäjä ei ymmärtäisi mitään viestin sisällöstä. *Tunnelointia* voidaan käyttää esimerkiksi palomuurin ohitukseen. Palvelin, jolle alkuperäinen pyydetty *URI* kuuluu, on nimeltään *varsinainen palvelin* (original server) (W3C, 1999a).



Kuva 4.7: *HTTP*-yhteyskäytännön mukainen välitystä käyttävä pyyntö-vastaus -malli.

HTTP-kommunikointia käytetään yleensä *TCP/IP*-yhteyden yli. Oletusportti on *TCP 80*, mutta muitakin portteja voidaan käyttää. *HTTP*-yhteyskäytännön versioon 1.1 lisättiin

ominaisuus, joka mahdollistaa samaa yhteyttä käyttäen tehdä useita pyyntöjä. Tämä tehostaa kommunikointia palvelimen kanssa, koska yhteyden muodostus on usein suhteellisen hidas toimenpide. Jokainen *HTTP*-asiakkaan pyyntö ja palvelimen vastaus koostuu kolmesta osasta (W3C, 1999a): aloitusrivistä, otsakeriveistä ja viestirungosta. Otsakerivien jälkeen ennen viestirunkoa on tyhjä rivi.

Asiakassovellus ottaa yhteyden *HTTP*-palvelimelle ennalta määriteltyyn porttiin ja lähettää palvelupyynnön määritellen *HTTP*-komennon kutsumaan metodia *GET* tai *POST*. Seuraavaksi alustusriville tulee paikallinen polku pyydettyyn dokumenttiin ja viimeisenä on käytettävän *HTTP*-version numero. Alustusrivi palvelupyynnölle voisi olla seuraavan esimerkin mukainen, jossa käytetään *POST*-metodia kutsumaan resurssia *RPCServlet HTTP*-yhteyskäytännön versiolla 1.1:

```
POST /gradu/RPCServlet HTTP/1.1
```

Palautteella on erilainen alustusrivi kuin palvelupyynnöllä. Palvelinsovellus vastaa asiakassovelluksen pyyntöön alustusrivillä, joka koostuu kolmesta kentästä: *HTTP*-versiosta, tilakoodista ja tilakoodin kuvauksesta. Kentät on erotettu toisistaan välilyönneillä. Seuraava esimerkki palvelinsovelluksen lähettämästä alustusrivistä kertoo, että palvelin käyttää vastauksessaan *HTTP*-yhteyskäytännön versiota 1.1. Tilakoodi 200 tarkoittaa, että asiakassovelluksen pyyntöön onnistuttiin vastaamaan (W3C, 1999a):

```
HTTP/1.1 200 OK
```

Alustusriviä seuraavat otsakerivit tarjoavat informaatiota palvelupyynnöstä tai vastauksesta sekä viestirungossa siirrettävästä datasta. Otsakerivit lähetetään tekstirivimuotoisina nimi-arvo-pareina. Rivit on erotettu toisistaan rivinvaihtomerkein. *HTTP*:ssä on määritelty käyttötarkoituksen mukaan neljää eri tyyppiä olevia otsakkeita seuraavasti: yleiset otsakkeet, pyynnön otsakkeet, vastauksen otsakkeet ja entiteetin otsakkeet. Yleisiä otsakkeita käytetään sekä asiakkaan pyynnöissä että palvelimen vastauksissa. Pyynnön otsakkeet mahdollistavat pyyntöihin sekä asiakkaaseen itseensä liittyvän tiedon kuljettamisen asiakkaalta palvelimelle. Vastauksen otsakkeet antavat mahdollisuuden kuljettaa palvelimelta sellaista vastaukseen liittyvää lisätietoa, jota ei voi kuljettaa tilarivillä. Entiteetin otsakkeet sisältävät tietoa siirrettävästä datasta (W3C, 1999a).

4.2.2 XML-pohjainen yhteyskäytäntö

XML on lyhenne nimestä *eXtensible Markup Language* - laajennettava merkintäkieli. Siitä odotetaan ratkaisua moniin elektronisen julkaisemisen ja tiedonsiirron ongelmiin. *XML*:stä näyttäisi olevan tulossa eräänlainen tietojenkäsittelyn yleiskieli, joka saattaa mahdollistaa tiedon siirron kaikkien sovellusten ja järjestelmien välillä.

XML-määrityksen (W3C, 2004) mukaan *XML*-dokumentilla on sekä looginen että fyysinen rakenne. Loogisesti dokumentti koostuu esittelyistä, elementeistä, komennoista, merkkiviittauksista ja käsittelyohjeista, jotka kaikki näkyvät dokumentissa eksplisiittisenä merkkauksena. *XML*-dokumentin fyysinen rakenne koostuu entiteeteistä (entity). Entiteetit ovat ennalta määriteltyjä vakioita, joilla on nimi ja sisältö. Entiteetit jaetaan sekä sisäisiin että ulkoisiin. Sisäinen entiteetti ei ole fyysisesti tallennettu esimerkiksi levylle, vaan sen arvo on määritelty rakennemäärityksessä, entiteetin esittelyssä. Ulkoiset entiteetit ovat erillisiä tiedostoja, jotka voivat sijaita fyysisesti erillään toisistaan. Ulkopuolisia entiteettejä voivat olla muun muassa kuvatiedostot ja toiset *XML*-dokumentit. Niihin viitataan *XML*-dokumentissa entiteetin nimellä.

Eräs rakenteisten dokumenttien tärkeimpiä ominaisuuksia on mahdollisuus pakottaa looginen rakenne tietyn mallin mukaiseksi. Jotta dokumentti olisi *XML*-dokumentti, sen tulee olla *XML*-määrityksen mukaan hyvin muodostettu:

- sisältää täsmälleen yhden juurielementin ja kaikki muut sen mahdollisesti sisältämät elementit alkavat ja päättyvät saman elementin sisällä eli elementit ovat tasapainossa,
- vastaa XML-määrittelyn asettamia hyvinmuodostetun dokumentin rajoituksia,
- jokainen jäsennetty entiteetti, johon dokumentissa viitataan, on hyvinmuodostunut.

Hyvinmuodostuneisuuden lisäksi *XML*-dokumentille voidaan asettaa myös muita vaatimuksia, joista tärkein on dokumentin loogisen rakenteen vakiomuotoisuus eli valiuden vaatimus. Validilla *XML*-dokumentilla tarkoitetaan *XML*-dokumenttia, joka on hyvin muodostettu ja sen looginen rakenne on kuvattu rakennemäärittelyssä. *XML*-dokumen-

tin rakenteen määrittelyssä voidaan käyttää joko *DTD*:tä (Document Type Definition) tai *XML*-skeemoja (XML Schema).

DTD voi olla sisäinen tai ulkoinen. Sisäinen *DTD* sijoitetaan suoraan XML-dokumentin alkuun *!DOCTYPE*-elementin sisään. Tällöin dokumentin rakennemäärittely on vain kyseisen dokumentin käytettävissä. Ulkoinen *DTD* sijaitsee omassa tiedostossaan *XML*-dokumentin ulkopuolella. Käytettäessä ulkoista rakennemäärittelyä helpotetaan monien ihmisten kesken jaettavien *XML*-sovellusten luomista. *DTD* määrittelee, kuinka dokumentti on muotoiltu. Sen täytyy määritellä kaikki dokumentissa sallitut elementit, sallitut attribuutit, elementtien sijoittaminen toisiinsa nähden, elementtien esiintymät ja kaikki ulkoiset entiteetit (W3C, 2004). Kuvassa 4.8 kuvataan yksinkertainen *XML*-dokumentti ja siihen liittyvä sisäinen *DTD*-rakennemäärittely.

```
<?xml version="1.0" standalone="yes" encoding="ISO-8859-1"?>
<!DOCTYPE Yritys [
  <!ELEMENT Yritys (Asiakas)
  <!ELEMENT Asiakas(Nimi, Osoite)
  <!ELEMENT Nimi (#PCDATA)
  <!ELEMENT Osoite (#PCDATA)
  ]
<Yritys>
  <Asiakas>
    <Nimi>Nieminen Oy</Nimi>
    <Osoite>Joensuu</Osoite>
  </Asiakas>
</Yritys>
```

Kuva 4.8: *XML*-dokumentin rakenne ja sen rakennemäärittely.

Kuvassa 4.8 kuvatussa yksinkertaisessa *XML*-dokumentissa juurielementti on *Yritys*, jonka sisällä kaikki muut elementit alkavat ja päättyvät, joten dokumentti täyttää hyvinmuodostuneisuuden vaatimuksen. Dokumentti sisältää sisäisen rakennemäärittelyn, jonka mukaan dokumentin juurielementin *Yritys* tulee sisältää nolla tai useampia *Asiakas*-elementtejä. *Asiakas*-elementillä täytyy olla tasan yksi *Nimi*-elementti ja *Osoite*-elementti, jotka sisältävät merkkimuotoista tietoa. Dokumentin varsinainen sisältö tallennetaan merkkitietona ja sen sisältömallina käytetään avainsanaa *#PCDATA* (Parserable Character DATA). *DTD*-rakennemäärittelyssä varsinainen sisältö määritellään aina merkkitietona, eikä merkkitietoa voida määrätä sisältämään mitään erityistä tietotyyppiä. Tämä täsmällisyyden puute on yksi syy siihen, miksi *DTD*-rakennemäärittely

halutaan korvata *XML*-skeemoilla. XML-skeema mahdollistaa tietotyypit, enimmäis- ja vähimmäismäärät elementtien sisällön rajoituksina sekä uniikit arvot.

Jotta sovellus pystyisi vastaanottamaan ja käsittelemään *XML*-tietoa, pitää sillä olla *XML*:ää ymmärtävä rajapinta. *XML*-dokumentti on yhteydessä vain rajapintaan, jolloin se näkee vain rajapinnan eikä sovellusta. Samoin sovellus näkee vain rajapinnan ja saa kaikki tiedot sen kautta. Rajapinta on fyysisesti ohjelmaluokka, joka määrittää joukon metodeja, joiden avulla sovellusohjelma saa käyttöönsä *XML*-dokumentin sisältämää tietoa. Tärkeimmät *XML*-rajapinnat ovat *SAX* (Simple Api for XML) ja *DOM* (Document Object Model) (McLaughlin, 2001).

SAX on tapahtumapohjainen *XML*-rajapinta. Jotta sovellus voisi tehdä jotain hyödyllistä jäsennettävällä *XML*-tiedolla, sen on rekisteröitävä käsittelijöitä (handlers) *SAX*-jäsentimeen. Käsittelijät ovat takaisinkutsuluokkia, jotka mahdollistavat sovelluskohtaisen ohjelmakoodin liittämisen *SAX*:n jäsentämistapahtumiin. Nämä tapahtumat käsitellään jäsentämisen aikana eikä vasta jäsentämisen tapahduttua. Tästä syystä dokumentti voidaan käsitellä lataamatta sitä ensin kokonaan muistiin. *DOM* on puupohjainen *XML*-rajapinta, jossa dokumentti luetaan tietokoneen muistiin hierarkkiseksi rakennepuuksi. Puupohjainen jäsennin mahdollistaa monipuolisemman *XML*-dokumentin käsittelyn. Toisaalta se vaatii enemmän työmuistia, koska kuvaus koko dokumentista ladataan tietokoneen muistiin.

Yksi *XML*:n keskeisimmistä ajatuksista on aina ollut se, että sitä voitaisiin käyttää tiedon vaihdossa hajautettujen järjestelmien välillä. Kun *W3C* julkisti ensimmäisen *XML*-suosituksensa helmikuussa 1998, suositus ei ottanut millään tavalla kantaa siihen, miten eri *XML*-sanastoja voitaisiin yhdistellä ja käsitellä. Suosituksesta puuttui kokonaan tuki nimiavaruuksille. Tämän seurauksena kaikkien *XML*-dokumenttien elementtien katsottiin kuuluvan yhteen ja samaan nimiavaruuteen. Tästä syystä elementtien yksiselitteinen nimeäminen oli vaikeaa. *W3C* kehitti nimiavaruuksiin viittaamistekniikan vastauksena ongelmiin, jotka aiheutuivat samalla tavalla nimetyistä *XML*-dokumentin elementeistä ja attribuuteista. W3C (1999b) määrittelee *XML*-nimiavaruuden seuraavasti: *XML*-nimiavaruudet tarjoavat menetelmän, jonka avulla kokoelma elementtien ja attribuuttien nimiä voidaan sovelluksissa yksikäsitteisesti nimetä.

Elementtien ja attribuuttien nimiä kutsutaan sovelluksen sanastoksi. *XML*-nimiavaruus on yksinkertaisesti sanasto, jolle on annettu nimi. *XML*-nimiavaruuksien nimeämiseen käytetään *URI*-viitettä. Viite on hieman harhaanjohtava nimitys, sillä todellisuudessa *XML*-nimiavaruuksia ei esiinny missään fyysisessä eikä käsitteellisessä muodossa. Nimiavaruuden nimi on siis vain tunniste, jolla nimiavaruus voidaan erottaa toisista nimiavaruuksista. W3C (1999b) suositus nimiavaruuksille ei määritä mitään muuta kuin kaksiosaisen nimeämisjärjestelmän elementtityypeille ja attribuuteille. Ensimmäinen osa nimestä on *URI*-tunniste, jota käytetään nimiavaruuden tunnistamiseen. Toisen osan tässä nimeämisjärjestelmässä muodostavat itse elementtien ja attribuuttien nimet. Yhdessä nämä kaksi osaa muodostavat elementtien tai attribuuttien yleiskäyttöisen nimen.

4.2.3 XML-RPC -pohjainen yhteyskäytäntö

XML-RPC on *XML*-pohjainen proseduurien etäkutsuyhteyskäytäntö (Remote Procedure Call), jota käytetään *Internetin* yli. *XML-RPC* sallii kirjaimellisesti kaikenlaisten sovellusten yhteistoiminnan, koska siirtoyhteyskäytännöksi käy *HTTP*. Koska käytännössä kaikissa nykyään käytössä olevissa ohjelmointikielissä on jonkinlainen tuki *HTTP*-yhteyskäytännölle, *XML-RPC* on hyvä vaihtoehto yhteydenpitoon eri ohjelmointikielillä kirjoitettujen ohjelmien kanssa. *RPC*:n ongelma on perinteisesti ollut tiedon sarjallistaminen kuljetuksen ajaksi. *XML*:n käyttö proseduurien etäkutsuissa mahdollistaa järjestelmäriippumattoman tavan tiedon sarjallistamiseen ja sarjallistamisen purkuun. Sarjallistamisella tarkoitetaan tiedon koodaamista *XML*-muotoon viestin siirtämisen ajaksi ja sarjallistamisen purulla taas *XML*-muodossa olevan tiedon palauttamista takaisin alkuperäiseen muotoonsa. *XML-RPC* mahdollistaa yksinkertaisen tavan muodostaa proseduurien etäkutsuja sisältäviä viestejä ja näiden vastauksia (Harold, 2002).

Periaatteeltaan *XML-RPC* on asiakas-palvelin -sovellus. Asiakassovelluksessa muodostetaan *XML*-pohjainen viesti, joka sisältää kutsuttavan palvelun nimen ja sen mukana lähetettävät parametrit. Viesti lähetetään palvelinsovellukselle käyttäen *HTTP-POST* -

metodia. Palvelinsovellus kutsuu palvelua ja välittää sille parametrit, jotka se otti vastaan asiakassovellukselta. Lopuksi palvelinsovellus muodostaa *XML*-pohjaisen vastaussanoman ja lähettää sen kutsun tehneelle asiakassovellukselle (Winer, 1999).

Palvelusta puhutaan usein aliohjelmana, vaikka palvelun toteutusta ei *XML-RPC*-määritys mitenkään sidokaan. Palvelu voi olla esimerkiksi *Java*-luokan metodi tai suoritettava skripti. Palvelun toteutustapa on täysin palvelinsovelluksen valittavissa eikä toteutustapa näy millään tavalla asiakassovellukselle. Tarvittaessa alkuperäinen toteutus voidaankin korvata toisella eikä palvelua käyttäviin asiakassovelluksiin tarvita mitään muutoksia. Kuvassa 4.9 kuvataan palvelimella *URL*-osoitteessa http://stock.cafeconleche.org/quotes.cgi löytyvän metodin *getQuote(String symbol)* kutsun sarjallistaminen *XML-RPC* kutsuksi (Harold, 2002).

```
POST /quotes.cgi HTTP/1.0
Host: stock.cafeconleche.org
Content-Type: text/xml
Content-length: 167

<?xml version="1.0"?>
<methodCall>
  <methodName>getQuote</methodName>
  <params>
    <param>
      <value><string>RHAT</string></value>
    </param>
  </params>
</methodCall>
```

Kuva 4.9: *XML-RPC* -kutsun sarjallistaminen kuljetusta varten (Harold, 2002).

Esimerkissä kutsutaan palvelimelta *stock.cafeconleche.org* metodia *getQuote*, jolle välitetään parametrina merkkijono. *XML-RPC* -yhteyskäytännön mukainen viesti kulkee verkossa aina *HTTP*-yhteyskäytännön mukana. *XML-RPC* -viesti sidotaan *HTTP*-viestiin upottamalla se HTTP-viestin runkoon ja tämän takia täytyy viestissä olla *HTTP*-yhteyskäytäntöön pakollisena kuuluvat otsaketiedot. Pakollisia otsaketietoja aloitusrivin lisäksi ovat seuraavat (Winer, 1999):

- *Host-kenttä* ilmaisee *HTTP*-palvelimen nimen ja mahdollisen portin numeron.
- *User-agent* on selain tai muu vastaava ohjelma, jolla otetaan yhteyttä palvelinsovellukseen.

- *Content-type* kertoo viestin MIME-tyypin, jonka avulla viestin vastaanottava järjestelmä kykenee tunnistamaan viestin sisällön tyypin.
- *Content-length* ilmaisee viestin pituuden tavuina.

HTTP-viestin runko-osaan *methodCall*-elementin sisälle kääritään *XML-RPC* -etäkutsu. Elementti *methodCall* sisältää kaksi lapsielementtiä. Kutsuttavan metodin nimi sijoitetaan *methodName*-elementin sisälle. Jos kutsussa välitetään parametreja, pitää *methodCall*-elementin sisältää *params*-elementti, joka sisältää tarvittavan määrän *param*-elementtejä. Jokainen *param*-elementti sisältää yhden *value*-elementin, joka sisältää arvon tyypin ilmaisevan elementin ja tyyppiä vastaavan arvon. Jos parametrin tyyppiä ei ole ilmoitettu, niin tyyppinä käytetään oletusarvoisesti merkkijonotyyppiä. *XML-RPC* –kutsuun ei merkitä parametrien tunnuksia, vaan ainoastaan niiden tyypit ja arvot. Vastaanottava sovellus tunnistaa parametrien arvot toisistaan sen perusteella, missä järjestyksessä ne on esitetty *XML-RPC* -viestissä (Winer, 1999).

```
HTTP/1.1 200 OK
Date: Mon, 16 Jul 2001 20:12:37 GMT
Server: Apache/1.3.12 (Unix) mod_perl/1.24
Content-Length: 140
Content-Type: text/xml
Connection: close

<?xml version="1.0"?>
<methodResponse>
  <params>
    <param>
      <value><double>4.12</double></value>
    </param>
  </params>
</methodResponse>
```

Kuva 4.10: *XML-RPC* -vastaus (Harold, 2002).

Palvelinsovellus vastaanottaa palvelupyynnön ja muodostaa vastausviestin, joka on myös *XML*-dokumentti. Kuvassa 4.10 kuvataan vastausviesti kuvan 4.9 pyyntöön. Vastausviesti kääritään *methodResponse*-elementin sisälle ja vastausviesti voi sisältää ainoastaan yhden palautettavan arvon. *HTTP*-yhteyskäytännön vaatimille otsakekentille on asetettu seuraavat vaatimukset: viestin pitäisi aina palauttaa 200 *OK*, ellei pyynnön toteutuksessa ole tapahtunut virhettä, *Content-type* on oltava *text/xml* ja *Content-length* on määriteltävä ja sen oltava oikein. Otsake *Content-length* määrittelee *HTTP*-viestin runko-osassa kuljetettavan datan koon tavuina (Harold, 2002).

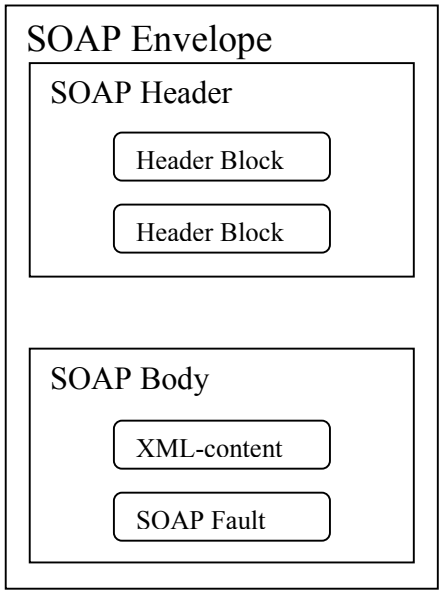
XML-RPC:n avulla voidaan kutsua mitä tahansa metodia, kunhan vain metodin parametrien ja paluuarvojen tyypit ovat *XML-RPC*:n tukemia. Parametrin tyypin oletusarvo on merkkijono (String). Muita mahdollisia tyyppejä *XML-RPC*:ssä ovat kokonaisluku (int), liukuluku (double), totuusarvo (boolean), päivämäärä (date), taulukko (array) ja tietue (struct). *XML-RPC* ei mahdollista omien tietotyyppien käyttöä. Vaikka tuettavien tietotyyppien määrä on rajoitettu, ne riittävät suurimpaan osaan etäproseduurikutsuja. Taulukko- ja tietue-tietotyyppien sisältämät yksittäiset alkiot voivat olla mitä tahansa *XML-RPC*:n tukemia tietotyyppejä, joten ne voivat siten olla myös taulukko- tai tietuetyyppiä. Tästä syystä *XML-RPC* mahdollistaa varsin monipuolisen tietotyyppien valikoiman (Harold, 2002).

4.2.4 SOAP-pohjainen yhteyskäytäntö

SOAP (Simple Object Access Protocol) on yksinkertainen ja kevyt tiedon siirtoon tarkoitettu yhteyskäytäntö. Se on *XML*-kieleen pohjautuva yhteyskäytäntö, joka on suunniteltu rakenteisen ja tyypitetyn tiedon esitykseen ja siirtoon keskittämättömissä ja hajautetuissa ympäristöissä. *SOAP*:in merkittävä etu muihin yhteyskäytäntöihin verrattuna on sen käyttöjärjestelmä- ja ohjelmointikieliriippumattomuus. *SOAP*-määritys kuvaa tavan kuljettaa *XML*-pohjaisia viestejä verkossa. Se tekee tämän tarjoamalla *XML*-pohjaisen viestikehyksen, jolla on mm. seuraavia ominaisuuksia (W3C, 2003a):

- *Laajennettavuus*: laajennettavuus on yksi *SOAP*:in avainominaisuuksista. Yhteyskäytännön yksinkertaisuutta pidettiin tärkeänä päämääränä *SOAP*:ia suunniteltaessa. Tästä osoituksena on hajautetuille järjestelmille ominaisten tiettyjen ominaisuuksien puuttuminen määrityksestä. Tällaisia ominaisuuksia ovat mm. turvallisuus, luotettavuus ja viestin reititys. *SOAP* mahdollistaa näiden ja mahdollisten uusien ominaisuuksien lisäyksen laajennusmekanismin avulla.
- *Siirtoyhteyskäytäntöriippumattomuus*: *SOAP*-yhteyskäytännön mukainen viesti ei voi itsenäisesti kulkea verkossa, vaan se kuljetetaan aina jonkun tiedonsiirtoyhteyskäytännön päällä. *SOAP* ei myöskään määrittele viestien siirtoon käytet-

tyä yhteyskäytäntöä, mutta *SOAP*-määritys sisältää kuvauksen *HTTP*:n käytöstä tiedonsiirtoyhteyskäytäntönä.



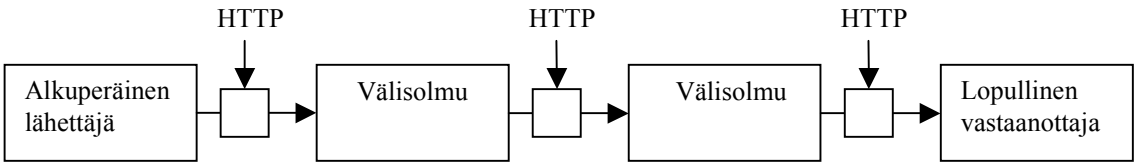
Kuva 4.11: *SOAP*-viestin rakenne.

SOAP-viesti on pohjimmiltaan *XML*-dokumentti, joka koostuu aina kolmesta perusosasta kuvassa 4.11 kuvatulla tavalla. *SOAP*-viestin kehyksen muodostaa kuorielementti (envelope) ja on pakollinen kaikissa viesteissä. Otsake-elementti sisältää viestin käsittelemiseen tarvittavaa tietoa ja toteuttaa tavan liittää *SOAP*-viestiin laajennuksia. Otsake-elementti ei ole pakollinen osa viestissä. Runkoelementti on pakollinen kaikissa *SOAP*-viesteissä ja siinä kuljetetaan varsinainen siirrettävä tieto. Elementtien sisältö jää suurelta osin sovelluksen kehittäjän vastuulle. *SOAP*-määritys määrittelee vain niiden rakenteen ja järjestyksen (W3C, 2003a).

SOAP-viestin uloimman kerroksen muodostaa kuorielementti, joka kapseloi viestin muut osat sisäänsä. Kuorielementti sisältää tietoa viestin lähettäjästä ja viestin käsittelyohjeita viestin vastaanottajalle sekä viestin koodauksesta (encoding). Kuva 4.13 esittää tavan, jolla kuorielementissä otetaan käyttöön nimiavaruus, jonka avulla *SOAP*-viestin oikeellisuus voidaan todentaa (W3C, 2003b).

Otsake-elementi (header) on valinnainen ja sen tulee olla kuorielementin välitön lapsielementti. Otsake-elementti muodostaa *SOAP*-viestin laajennusmekanismin, joka voi sisältää mielivaltaisen määrän lapsielementtejä. Näistä lapsielementeistä käytetään nimitystä otsakelohkoelementti (header block) ja kukin näistä muodostaa laajennuksen *SOAP*-määritykseen. Tällainen laajennus on mm. viestin reititys, joka toteutetaan lisämäärityksen avulla (W3C, 2003a).

SOAP-määritys kuvaa viestin käsittelymallin, jonka mukaan viesti on käsiteltävä sen kulkiessa lähettäjältä vastaanottajalle. Viestin käsittelijöistä käytetään nimitystä *SOAP*-solmu (SOAP-node). Viestin muodostavasta ja lähettävästä solmusta käytetään nimitystä *alkuperäinen lähettäjä* (initial sender) ja vastaanottajasta, jolle viestin runko-osa on tarkoitettu, käytetään nimitystä *lopullinen vastaanottaja* (ultimate receiver). Näiden lisäksi viestipolulla voi sijaita muita solmuja, joiden kautta viestin on kuljettava ennen saapumistaan lopulliselle vastaanottajalleen. Näistä solmuista käytetään nimitystä *välisolmu* (intermediate node), jotka voivat käsitellä vain niille tarkoitetut otsakelohkoelementit (W3C, 2003a).



Kuva 4.12: *SOAP*-viestipolun rakenne.

Kuvassa 4.12 kuvataan viestipolku alkuperäiseltä lähettäjältä välittäjäsolmujen kautta lopulliselle vastaanottajalle. *SOAP*-määritys ei ota kantaa siihen, miten tällainen viestipolku määritellään ja toteutetaan (W3C, 2003b). *SOAP*:issa viestipolku toteutetaan lisämäärityksen avulla, jonka *SOAP*:in laajennettavuus mahdollistaa. Reititys-määritys lisätään *SOAP*:in otsake-elementtiin (W3C, 2003a).

SOAP-määritys kuitenkin kuvaa, kuinka solmun on käsiteltävä vastaanottamansa viesti. Solmuille on määritelty roolit, joiden mukaan solmun täytyy käsitellä vastaanottamansa viesti. Jokaisessa *SOAP*-solmussa voidaan käsitellä kyseiselle solmulle kohdistetut ot-

sakelohkoelementit. *SOAP*-viestin runko-osa voidaan käsitellä vain viestipolun viimeisessä solmussa. Jos solmussa ei käsitellä sille tarkoitettua viestin osaa tarkoituksen mukaisella tavalla, tulee sen kuitenkin pystyä muodostamaan tästä virheilmoitus ja lähettämään se eteenpäin sekä lopettaa viestin käsittely. *SOAP*-määrityksen mukaan solmun täytyy omaksua tietty *rooli*, jossa sen täytyy käsitellä vastaanottamaansa viestiä. Roolit mahdollistavat viestin otsakelohkoelementtien kohdistamisen halutuille *SOAP*-solmuille. Rooli ilmaistaan rooliattribuutin avulla, jonka arvona käytetään *URI*-osoitetta seuraavasti: http://w3.org/2003/05/soap-envelope/role/roolinnimi. Taulukossa 4.1 kuvataan *SOAP*-määrityksessä määritellyt rooliattribuuttien arvot. Lisäksi *SOAP*-määritys mahdollista myös itsemääriteltyjen sovelluskohtaisten roolien käytön (W3C, 2003b).

Taulukko 4.1: *SOAP*-viestin rooliattribuutit.

Rooliattribuutin nimi	Kuvaus
next	Jokaisen välittäjäsolmun ja viestin lopullisen vastaanottajan täytyy toimia tässä roolissa.
ultimateReceiver	Solmun ollessa viestin lopullinen vastaanottaja, sen täytyy toimia tässä roolissa. Välittäjäsolmut eivät saa toimia tässä roolissa
none	Solmu ei saa toimia tässä roolissa.

Rooliattribuuttiin läheisesti liittyy *mustUnderstand*-attribuutti, joka on *boolean*-tyyppiä ja täten sen arvo voi olla joko *true* tai *false*. Attribuutilla *mustUnderstand* osoitetaan solmulle pakolliset otsakelohkoelementit vastaanottamastaan *SOAP*-viestistä, jotka sen täytyy pystyä käsittelemään oikein. Kuvassa 4.13 kuvataan attribuutin *mustUnderstand* ja rooliattribuutin käyttöä. Koska otsakelohkoelementin *warp:path* rooliattribuutin arvoksi on asetettu *next* ja attribuutilla *mustUnderstand* arvona on *true*, täytyy viestipolulla seuraavaksi viestin vastaanottavan solmun kyetä käsittelemään kyseinen otsakelohkoelementti oikein. Jos *SOAP*-solmu ei käsittele sille suunnattuna pakollista otsakelohkoelementtiä asianmukaisella tavalla, määrityksen mukaan sen täytyy muodostaa virheilmoitus ja lopettaa viestin käsittely (W3C, 2003b).

```
<soap:Envelope
  xml:soap=http://www.w3.org/2003/05/soap-envelope>
    <soap:Header>
      <wsrp:path xmlns:wsrp=http://schemas.xmlsoap.org/rp
        soap:role=http://www.w3.org/2003/05/soap-envelope/role/next
        soap:mustUnderstand = "true">
   ...
      </wsrp:path
   </soap:Header>
</soap:Envelope>
```

Kuva 4.13: Attribuutin *mustUnderstand* käyttö otsakelohkoelementissä.

SOAP-viestejä on mahdollista kuljettaa monen eri tiedonsiirtoyhteyskäytännön mukana, mutta *SOAP*-määrityksessä on kuvattu vain *HTTP*-yhteyskäytännön päällä kulkeva *SOAP*-viesti. *SOAP*-määrityksessä kuvattu sidos määrittelee tavan käyttää *SOAP*-yhteyskäytäntöä yhdessä *HTTP*-yhteyskäytännön kanssa. Tätä kahden yhteyskäytännön käyttämistä rinnakkain kutsutaan *sitomiseksi* (binding). Kun *SOAP*-viesti välittää etäkutsun, palautetaan vastaus *HTTP*-vastauksena. Sidoksen ansiosta *SOAP*-viestissä voidaan hyödyntää *HTTP*-yhteyskäytännössä määriteltyä vastauksen tilatietomallia. Esimerkiksi tilakoodi 200 kertoo, että asiakkaan pyyntö *SOAP*-osa mukaan lukien vastaanotettiin ja käsittely onnistui. Tilakoodit 400 ja 500 kertovat puolestaan pyynnön epäonnistumisesta. *SOAP*-viestin sisältävän *HTTP*-viestin *Content-Type* -otsakkeen on käytettävä mediatyyppiä *application/soap+xml*. Lisäksi *HTTP*-viestin otsakkeessa on käytettävä lisäkenttää *SOAPAction*, joka sisältää *SOAP*-viestin tarkoituksen kertovan *URI*:n (W3C, 2003a).

Proseduurien etäkutsujen suorittamista pidettiin yhtenä päätavoitteena, kun *SOAP*:ia määriteltiin. Käytettäessä *HTTP*:tä *SOAP*-viestien välitykseen saadaan etäkutsut välitettyä *HTTP-POST* -pyynnöissä ja etäkutsujen tulokset niiden *HTTP*-vastausviesteissä. Etäkutsujen käyttö ei kuitenkaan vaadi *HTTP*-yhteyskäytännön käyttöä. Etäkutsun tekemiseen tarvitaan seuraavat tiedot (W3C, 2003b):

- kohdeolion URI,
- kutsuttavan metodin nimi,
- valinnainen metodikuvaus (signature),
- metodin parametrit,
- valinnainen lisätieto otsikkoelementissä.

RPC-etäkutsut ja niiden vastaukset kuljetetaan *SOAP*-viestissä tietuerakenteena. Tietuerakenteen nimi on kutsuttavan metodin nimi ja sen tulee olla runkoelementin ainoa lapsielementti. Parametrit esitetään tietuerakenteen lapsielementteinä, jolloin parametrien tunnukset ovat myös kyseisten lapsielementtien tunnuksia ja lapsielementit sisältävät parametrien arvot. Kutsussa välitettävien parametrien on oltava samassa järjestyksessä kuin metodikuvauksessa. Kohdesolmun *URI* välitetään joko *HTTP*-viestin otsakkeen ensimmäisellä rivillä tai *SOAP*-viestin otsakelohkoelementissä.

5 TIEDONSIIRTOFORMAATTIEN ANALYSOINTI

Tutkielman tekemiseen kuuluu yhtenä osana kokeellinen osuus. Kokeellisessa osuudessa selvitetään kohdassa 4.2 esiteltyjen tiedonsiirtoyhteyskäytäntöjen suorituskykyjä ja ohjelmointipanosta empiirisesti esimerkkisovelluksen avulla tutkielman tavoitteiden mukaisesti. Kohdassa 5.1 esitellään empiirisessä tutkimuksessa käytettävät järjestelmät. Kohdassa 5.2 esitellään tutkimuksesta saadut suorituskykytulokset. Lopuksi kohdassa 5.3 vertaillaan empiirisessä tutkimuksessa käytettävien yhteyskäytäntöjen tarvitsemaa ohjelmointipanosta esimerkkisovelluksen pohjalta.

5.1 Esimerkkisovellus

Esimerkkisovellus on asiakas-palvelin-ohjelmisto, joka mahdollistaa palvelimella olevien toimittajatietojen haun matkapuhelimen avulla. Matkapuhelin käyttää verkkoyhteyttä palvelimeen ainoastaan silloin, kun haetaan halutut toimittajatiedot palvelimelta päätelaitteelle. Palvelimelta haetut toimittajatiedot tallennetaan päätelaitteen paikalliseen *RMS*-tietuevarastoon. Tutkimuksessa selvitetään asiakas- ja palvelinsovelluksen väliseen tiedonsiirtoon käytetty aika eri tiedonsiirtoyhteyskäytännöillä. Lisäksi mitataan aika, joka kuluu toimittajatietojen tallentamiseen päätelaitteen paikalliseen *RMS*-tietuevarastoon.

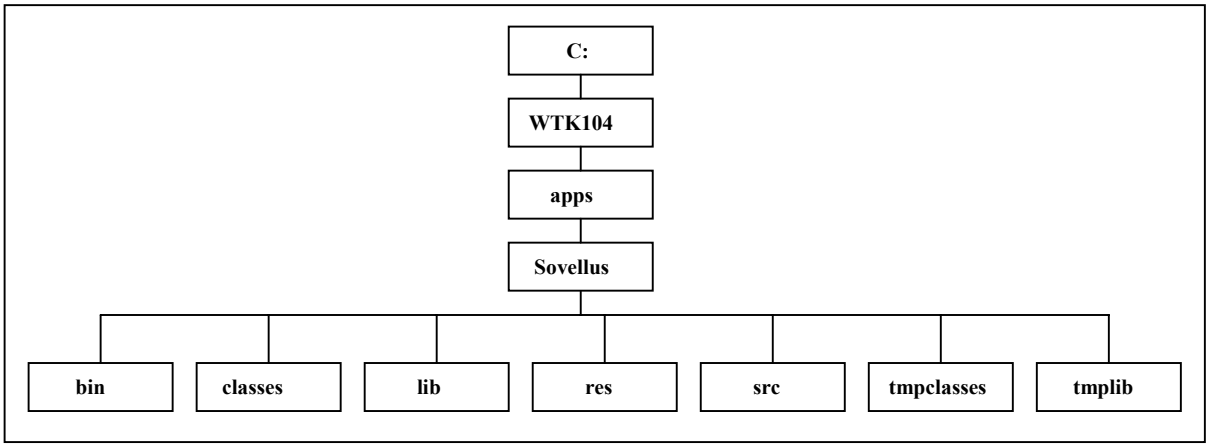
5.1.1 Asiakassovellus

Esimerkkisovelluksen toteutuksessa käytetään *Microsoft Windows NT* -ympäristössä erityisesti *MIDP*-sovellusten kehitykseen suunniteltua *Sunin J2ME Wireless Toolkit* (WTK) -toteutusta (Sun Microsystems, 2005), joka sisältää kääntäjän ja *MIDletin* paketointi- ja tarkistustoiminnot. Lisäksi *WTK* sisältää emulaattorin, jolla sovellusta voidaan testata kehitysaikana. *MIDP* versio 1.0 ei sisällä sisäänrakennettua *XML*-jäsennintä, *XML-RPC*-toteutusta eikä *SOAP*-toteutusta, joten tässä tutkimuksessa käytetään seuraavia *Enhydran MIDP*-ympäristöön tarkoitettuja *XML*-työkaluja:

- *XML*-jäsennin: *kXML* versio 1.2 (Enhydra, 2003a),
- *XML-RPC* -toteutus: *kXML-RPC* versio 0.6 (Enhydra, 2003c),
- *SOAP*-toteutus: *kSOAP* versio 1.2 (Enhydra, 2003b).

.

WTK muodostaa projektista kuvassa 5.1 kuvatun hakemistorakenteen. *MIDP*-sovellusten projektit luodaan aina *apps*-hakemiston alle. Projektin nimi on sovelluksen juurihakemisto ja siitä tulee myös valmiin *jar*-tiedostoksi pakatun *MIDP*-sovelluksen nimi.

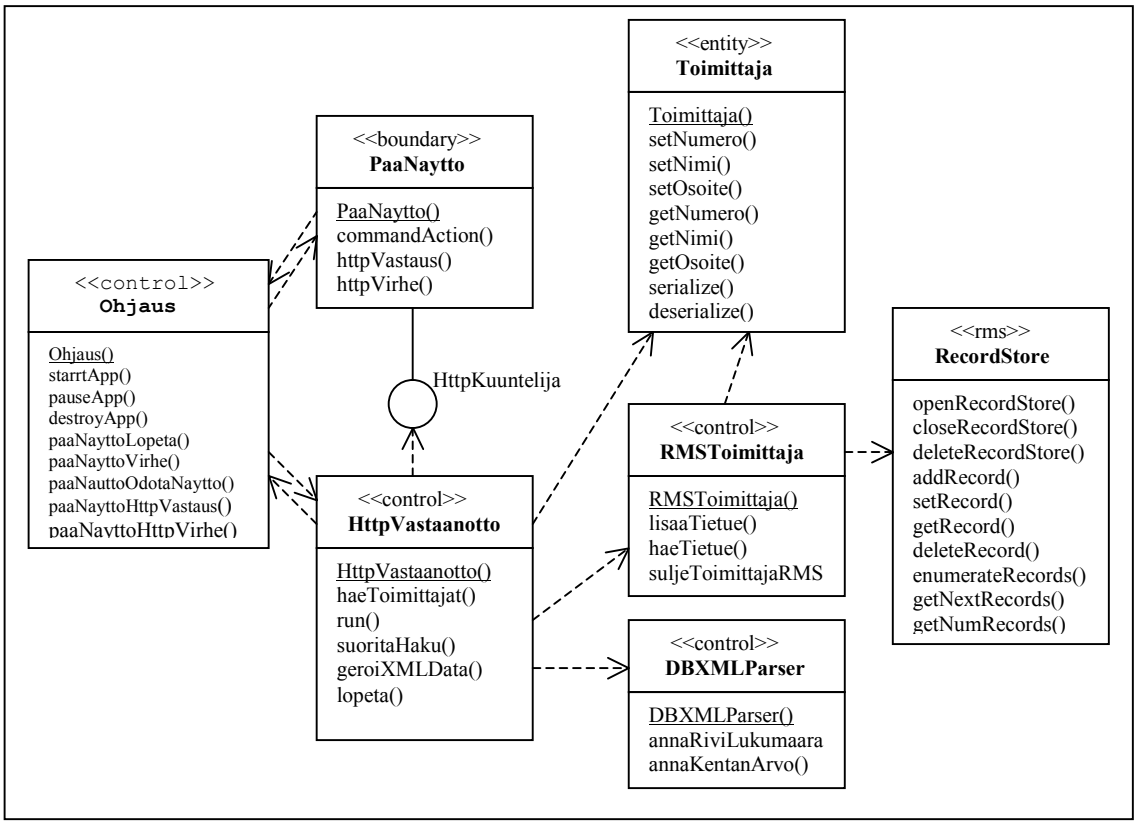


Kuva 5.1: *WTK*-kehitysympäristön hakemistorakenne.

Lähdekoodi tallennetaan hakemistoon *src*. *XML*-dokumentin jäsennin, *kXML-RPC*-toteutus ja *kSOAP*-toteutus tallennetaan hakemistoon *lib*. *WTK*:n *Build*-toiminnolla sovelluksen lähdekoodi käännetään ja esitarkistetaan ja käännetyt tiedostot sijoitetaan hakemistoon *classes*. Tämän jälkeen ohjelmaa voidaan testata *WTK*:hon sisältyvässä emulaattorissa. Todelliseen päätelaitteeseen siirtoa varten *MIDP*-sovellus pakataan *jar*-tiedostoksi *bin*-hakemistoon.

Tutkimuksen jokaiselle yhteyskäytännölle on oma asiakassovellus. Asiakassovellukset koostuvat kuudesta luokasta, *XML*-pohjaisen tiedonsiirron toteuttava asiakassovellus sisältää lisäksi luokan *DBXMLParser*, joka hoitaa *XML*-dokumentin jäsennyksen. Kuvassa 5.2 esitetään *XML*-pohjaisen tiedonsiirron toteuttava asiakassovelluksen luokka-

kaavio. Eri yhteyskäytäntöjä varten toteutetut asiakassovellukset poikkeavat toisistaan ainoastaan *HttpVastaanotto*-luokan *suoritaHaku*-metodin osalta.



Kuva 5.2: Asiakassovelluksen luokkakaavio.

XML-pohjaisen tiedonsiirron toteuttavan asiakassovelluksen pääluokka on *Ohjaus*, jonka avulla hoidetaan asiakassovelluksen näytön hallinta. Sovelluksen käyttöliittymän muodostaa *PaaNayttö*-luokka, joka vastaanottaa käyttäjän syötteet ja näyttää tulosteet. Luokan *HttpVastaanotto* sekä rajapinnan *HttpKuuntelija* avulla hoidetaan asiakassovelluksen verkkoyhteys palvelimelle. *RMSToimittaja*-luokka huolehtii toimittajatietojen tallentamisesta paikalliseen *RMS*-tietuevarastoon. *XML*-pohjaisen tiedonsiirron toteuttava asiakassovellus sisältää *DBXMLParser*-luokan, jolla hoidetaan vastaanotetun *XML*-dokumentin jäsennys. Liite 1 sisältää *XML*-pohjaisen tiedonsiirron toteuttavan asiakassovelluksen lähdekooditiedostot. Muiden asiakassovellusten lähde-koodien erot tulevat esille kohdissa 5.1.3 - 5.1.6, joissa annetaan asiakassovellusten tarkempi kuvaus.

Asiakas- ja palvelinsovelluksen väliseen tiedonsiirtoon ja toimittajatietojen tallentamiseen päätelaitteen *RMS*-tietuevarastoon käytetyn ajan mittaus toteutetaan kuvan 5.3 esittämällä tavalla. Tiedonsiirtoon käytetty aika sisältää palvelupyynnön muodostuksen ja lähetyksen asiakassovelluksessa, palvelupyynnön käsittelyn ja vastausviestin muodostuksen palvelimella sekä vastausviestin käsittelyn asiakassovelluksella. Asiakassovellus muodostaa vastausviestistä *Toimittaja*-tyyppisen olion ja tämä olion muodostus kuuluu vielä tiedonsiirtoon käytettyyn aikaan. Tiedon tallentamiseen *RMS*-tietuevarastoon käytetty aika sisältää toimittajatiedot sisältävän *Toimittaja*-tyyppisen olion tallennuksen *RMS*-tietuevarastoon.

```
Public void suoritaHaku(HttpKuuntelija kuuntelija)
  {
  ...
  alkuAika = System.currentTimeMillis();
    Asiakassovellus muodostaa palvelupyynnön.
    Asiakassovellus lähettää palvelupyynnön palvelimelle.
    Palvelin vastaanottaa palvelupyynnön.
    Palvelin käsittelee palvelupyynnön.
    Palvelin muodostaa vastausviestin.
    Palvelin lähettää vastausviestin asiakassovellukselle.
    Asiakassovellus vastaanottaa vastausviesti.
    Asiakassovullus käsittelee vastausviestin ja muodostaa
    Toimittaja-tyyppisen olion toimittajatiedoista.
  verkkoAika = System.currentTimeMillis() - alkuAika;

  alkuAika = System.currentTimeMillis();
    Asikassovellus tallentaa toimittajatiedot sisältävän
    Toimittaja-tyyppisen olion RMS-tietuevarastoon.
  rmsAika = System.currentTimeMillis() - alkuAika;

  Näytetään päätelaitteen näytöllä verkkoaika ja tallennusaika
  }
```

Kuva 5.3: Suoritusajan mittauksen toteutus.

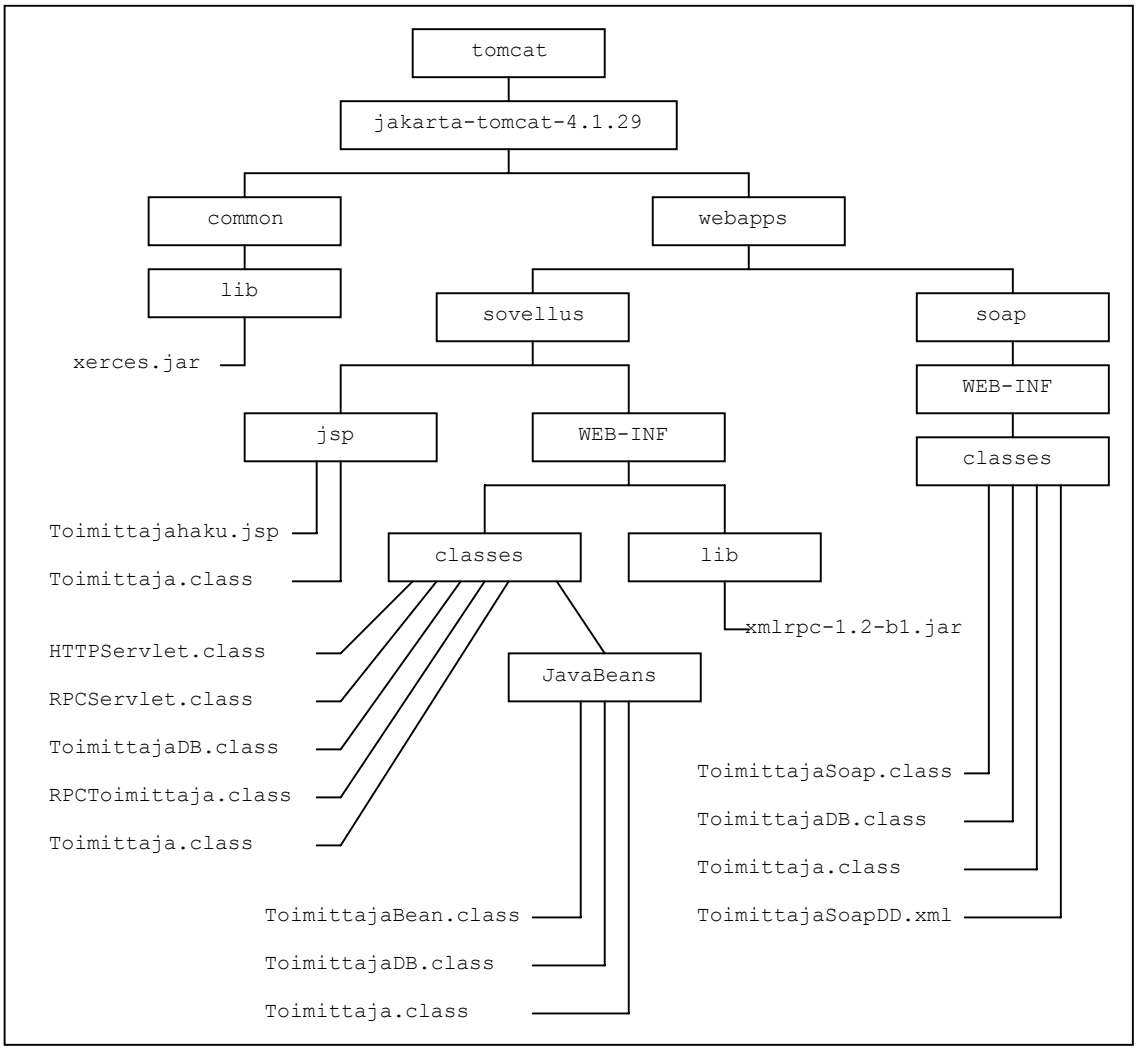
5.1.2 Palvelinarkkitehtuuri

Sovelluspalvelimena järjestelmässä käytetään *Unix*-koneelle sijoitettua *Tomcat*-palvelinta (Apache, 2004b), jolle sijoitetut *JSP*-sovelmat ja *servletit* käsittelevät asiakassovelluksen pyynnöt ja muodostavat vastaukset. Palvelinpuolen toteutuksessa käytetään seuraavia *XML*-työkaluja:

- *XML*-jäsennin: *Apache Xerces* versio 1.44 (Apache, 2004a),
- *XML-RPC* -toteutus: *Apache XML-RPC* versio 1.2 (Apache, 2003b),

- *SOAP*-toteutus: *Apache SOAP* versio 2.2 (Apache, 2003a).

Kuvassa 5.4 näytetään *Tomcat*-palvelimen hakemistarakenne ja käytettyjen ohjelmien sekä työkalujen sijoittelu sinne.

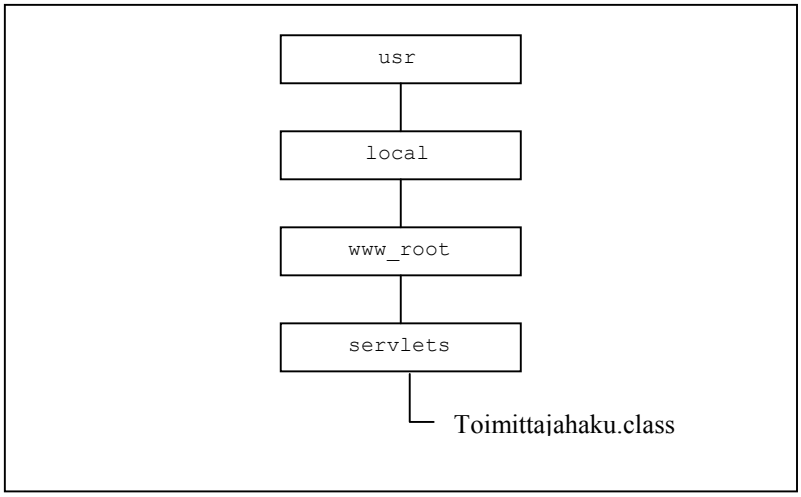


Kuva 5.4. *Tomcat*-palvelimen hakemistorakenne.

Tomcat-palvelimen hakemistoarkkitehtuurissa *webapps*-hakemisto sisältää kaikki web-sovellukset eli *kontekstit*. Konteksti *soap* sisältää SOAP-pohjaisen testisovelluksen luokkatiedostot ja *Apache SOAP* -toteutuksen. Konteksti *sovellus* sisältää *XML-RPC*-, *XML*- ja *HTTP*-pohjaisten testisovellusten luokkatiedostot. *JSP*-sovelmat sijoitetaan aina *web*-sovelluksen juurihakemistoon tai sen alihakemistoihin. *WEB-INF*-hakemisto sisältää kaksi alihakemistoa, jotka ovat *classes* ja *lib*. *Servletit* ja *JavaBeans*-

komponentit sijoitetaan hakemistoon *classes* tai sen alihakemistoihin. Kolmansien osapuolien komponentit sijoitetaan *lib*-hakemistoon. Tässä tutkimuksessa *lib*-hakemistoon sijoitettaan *Apachen XML-RPC*-toteutus.

Varsinaisena palvelimena toimiva *Tomcat* on palomuurilla suojattu. Tästä syystä asiakassovelluksen pyynnöt lähetetään varsinaiselle palvelimelle välittäjäpalvelimena toimivan *Apache*-palvelimen *tunnelointina*. Tunnelointi toimii linkkinä kahden yhteyden välillä muuttamatta viestiä mitenkään. Kuva 5.5 näyttää tunneloinnin toteuttavan *servletin* sijoituksen *Apache*-palvelimelle. Tarkempi dokumentointi löytyy lähteestä Holopainen (2003).



Kuva 5.5: *Apache*-palvelimen hakemistorakenne.

Palvelinpuolella suorituskykymittauksessa käytettävien toimittajatietojen varastoimiseen ei käytetä varsinaista tietokantaa, vaan toimittajatietojen varastointi toteutetaan *Java*-luokkatiedoston *ToimittajaDB* avulla. Tästä luokasta käytetään jatkossa nimeä *tietovarasto*, jonka toteuttamista tietokantana on tarkasteltu lähteissä Holopainen (2002; 2003). Kuvassa 5.6 kuvataan tietovaraston toteutus. Toimittajatietojen varastoinnissa käytetään *Hashtable*-tyyppistä oliota, jonne yksittäisen toimittajan tiedot tallennetaan aina *Toimittaja*-tyyppisessä oliossa. Palvelinsovellus käyttää tietovarastoa sen tarjoaman *getToimittaja*-metodin kautta, jolle välitetään parametrina haettavan toimittajan numero. Metodi palauttaa *Toimittaja*-tyyppisen olion, joka sisältää jäsenmuuttujissaan toimittajan numeron, nimen ja osoitteen.

```
public class ToimittajaDB
  {
  private Hashtable db;
  private String nimi = "Matti J Mainio";
  private osoite = "Joensuu";

  public ToimittajaDB() {
    db = new Hashtable();
    lisaa(new Toimittaja("10", "Matti J Mainio", "Joensuu"));
    lisaa(new Toimittaja("11",  kasvata(10, nimi), kasvata(10, osoite)));
    lisaa(new Toimittaja("12",  kasvata(50, nimi), kasvata(50, osoite)));
    lisaa(new Toimittaja("13",  kasvata(100, nimi), kasvata(100, osoite)));
    lisaa(new Toimittaja("14",  kasvata(200, nimi), kasvata(200, osoite)));
    lisaa(new Toimittaja("15",  kasvata(400, nimi), kasvata(400, osoite)));
    }

  private void lisaa(Toimittaja toimittaja) {
    db.put(toimittaja.getNumero(), toimittaja);
    }

  public Toimittaja getToimittaja(String numero) {
    return (Toimittaja) db.get(numero);
    }

  private String kasvata(int lkm, String param) {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < lkm; i++)
      sb.append(param);
    return sb.toString();
    }
  } // end
```

Kuva 5.6: Tietovaraston toteutus palvelimella.

5.1.3 HTTP-pohjaisen yhteyskäytännön toteutus

Tässä tutkimuksessa *HTTP*-pohjaisella yhteyskäytännöllä tarkoitetaan asiakas- ja palvelinsovelluksen välistä tiedonsiirtoa *HTTP*-viestin runko-osassa. *HTTP* on asiakas/palvelin-mallin mukainen verkkoyhteyskäytäntö. Siinä asiakassovellus avaa yhteyden ja lähettää palvelupyynnön *HTTP*-palvelimelle. Tämän jälkeen palvelin muodostaa ja palauttaa vastauksen, joka on yleensä se resurssi, jota asiakassovellus on palvelupyynnössään pyytänyt.

Esimerkkisovelluksessa *J2ME*-asiakassovellus kommunikoi verkon yli *HTTP*-yhteyttä käyttäen *J2EE*-palvelisovelluksen kanssa. Palvelinsovellus sijaitsee *Tomcat*-palvelimella, jossa palvelinsovelluksena käytetään *servlettiä*. Asiakassovellus lähettää palvelupyyntönä halutun toimittajan toimittajanumeron servletille, joka käsittelee pyynnön ja palauttaa vastauksena toimittajanumeroa vastaavan toimittajan tiedot asiakassovelluk-

selle. Kuvassa 5.7 esitetään miten, asiakassovelluksessa muodostetaan palvelupyyntö ja lähetetään palvelinsovellukselle.

```
con = (HttpConnection) Connector.open(url);
// Asetetaan tarvittavat HTTP-otsakekentät.
con.setRequestMethod(HttpConnection.POST);
con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
con.setRequestProperty("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");
con.setRequestProperty("Content-Language", "en-US");
con.setRequestProperty("Accept", "application/octet-stream");
con.setRequestProperty("Connection", "close");
// Muodostetaan palvelupyyntö.
ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
DataOutputStream out = new DataOutputStream(byteOut);
out.writeUTF(toimittajanro);
out.flush();
byte data[] = byteOut.toByteArray();
con.setRequestProperty("Content-Lengh", Integer.toString(data.length));
// Lähetetään palvelupyyntö palvelinsovellukselle.
os = con.openOutputStream();
os.write(data);
os.close();
```

Kuva 5.7: Palvelupyynnön muodostus asiakassovelluksessa.

Palvelupyynnön lähettämistä varten muodostetaan *HttpConnection*-tyyppinen yhteys palvelimelle sijoitetulle *servletille Connector*-luokan *open*-metodilla, jonka jälkeen *HTTP*-yhteys on *alustustilassa*, jolloin asetetaan tarvittavat *HTTP*-otsakekentät. *HTTP*-palvelupyyntö on toteutettu siten, että *ByteArrayOutputStream* ja *DataOutputStream* luokkien avulla pakataan parametrina välitettävä toimittajanumero tavutaulukoksi. *DataOutputStream*-luokan *writeUTF*-metodi kirjoittaa toimittajanumeron *Byte-ArrayOutputStream*-tyyppiseen olioon. Seuraavksi muutetaan *ByteArrayOutputStream*-olion sisältö tavutaulukoksi. Lopuksi avataan kirjoitusvirta kutsumalla metodia *open-OutputStream*, joka siirtää yhteyden tilaan yhdistetty. Pyyntö kirjoitetaan palvelimelle *OutputStream*-luokan metodilla *write*, jolle välitetään parametrina tavutaulukko.

Tomcat-palvelimella oleva servletti *HTTPServlet* käsittelee asiakassovelluksen lähettämän palvelupyynnön. *HTTPServlet*-luokan rakenne esitetään kuvassa 5.8. *Servlet*-sovelluksessa luodaan ilmentymä tietovarastoluokasta *ToimittajaDB*. Seuraavaksi tietovarastosta haetaan toimittajanumeroa vastaavan toimittajan tiedot *Toimittaja*-tyyppiseen olioon *ToimittajaDB*-luokan metodilla *getToimittaja*. *Toimittaja*-tyyppinen olio sarjallistaa itsensä tiedonsiirtoa varten *ByteArrayOutputStream*-tyyppiseen olioon. Tämän olion sisältö muutetaan tavutaulukoksi metodilla *toByteArray*. Lopuksi muodostetaan tulostusvirta *response*-olion avulla, jolla vastausviesti lähetetään asiakassovellukselle.

```
public class HTTPServlet extends HttpServlet
  {
  public void doPost(HttpServletRequest req, HttpServletResponse res)
    throws ServletException, IOException
    {
    DataInputStream in = new DataInputStream(req.getInputStream());
    // Luodaan tietovarasto
    ToimittajaDB db = new ToimittajaDB();
    // Haetaan toimittajatiedot tietovarastosta.
    Toimittaja toimittaja = db.getToimittaja(in.readUTF());

    // Muodostetaan vastausviesti.
    ByteArrayOutputStream baos = new ByteArrayOutputStream();
    DataOutputStream dos = new DataOutputStream(baos);
    toimittaja.serialize(dos);
    byte [] data = baos.toByteArray();
    res.setStatus(res.SC_OK);
    res.setContentLength(data.length);
    res.setContentType("application/octet-stream");

    // Lähetetään vastausviesti.
    OutputStream os = res.getOutputStream();
    os.write(data);
    os.close();
    }
  }
```

Kuva 5.8: *HTTPServlet*-luokan toteutus palvelimella.

Asiakassovellus vastaanottaa palvelinsovelluksen lähettämän vastausviestin muodostamalla *DataInputStream*-tyyppisen lukuvirran. Lukuvirta välitetään parametrina kuvan 5.2 *Toimittaja*-luokan metodille *deserialize*, missä lukuvirrasta muodostetaan *Toimittaja*-tyyppinen olio liitteen 1 mukaisesti. Kuvassa 5.9 kuvataan palvelinsovelluksen lähettämän vastausviestin käsittely asiakassovelluksessa. Lopuksi *Toimittaja*-tyyppisen olion sisältö tallennetaan päätelaitteen paikalliseen *RMS*-tietuevarastoon.

```
int rc = con.getResponseCode();
if (rc == HttpConnection.HTTP_OK) {
  DataInputStream dis = new DataInputStream(con.openInputStream());
  Toimittaja toimittaja = Toimittaja.deserialize(dis);
  ...
  // Tallennetaan toimittajatiedot RMS-tietuevarastoon.
```

Kuva 5.9: Vastausviestin käsittely asiakassovellusessa.

5.1.4 XML -pohjaisen yhteyskäytännön toteutus

XML-datan siirtämiseksi *J2ME*-asiakassovellus kommunikoi verkon yli *HTTP*-yhteyttä käyttäen *J2EE*-palvelinsovelluksen kanssa, jolloin palvelinsovelluksen *JSP*-sovelma käsittelee *MIDP*-asiakassovelluksen lähettämät *HTTP*-palvelupyynnöt. *JavaBeans*-komponentti muodostaa järjestelmän sovelluslogiikan, joka hoitaa tietovarastoyhteydet. *JSP*-sovelma muodostaa vastauksen *MIDP*-asiakassovellukselle käyttämällä sovelluslogiikan tarjoamia palveluja. Kuvassa 5.10 esitetään miten asiakassovelluksessa muodostetaan *XML*-dokumentti ja lähetetään se palvelupyyntönä palvelimelle.

```
con = (HttpConnection) Connector.open(url);
// Asetetaan tarvittavat HTTP-otsakekentät.
con.setRequestMethod(HttpConnection.POST);
con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
con.setRequestProperty("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");
con.setRequestProperty("Content-Language", "en-US");
con.setRequestProperty("Accept", "application/actet-stream");
con.setRequestProperty("Connection", "close");
// Muodostetaan palvelupyynnön XML-dokumentti
byte data [] = generoiXMLData();
con.setRequestProperty("Content-Length", Integer.toString(data.length));
//Lähetetään palvelupyyntö palvelinsovellukselle.
os = con.openOutputStream();
os.write(data);
os.close();
...
public byte [] generoiXMLData() {
  StringBuffer sb = new StringBuffer();
  sb.append("<root>");
  sb.append("<toimittaja>");
  sb.append("<toimittajanro>");
  sb.append(toimittajanro);
  sb.append("</toimittajanro>");
  sb.append("</toimittaja>");
  sb.append("</root>");
  return sb.toString().getBytes();
  }
```

Kuva 5.10: *XML*-dokumentin muodostus asiakassovelluksessa.

Palvelupyynnön muodostava *XML*-dokumentti rakennetaan metodissa *generoiXMLData*. *XML*-dokumentti kootaan *StringBuffer*-tyyppiseen olioon käyttämällä *StringBuffer*-luokan metodia *append*. Metodi *generoiXMLData* palauttaa *XML*-dokumentin tavutaulukkona. Kuvassa 5.11 esitetään palvelupyynnössä välitettävän *XML*-dokumentin rakenne. Palvelupyynnön lähettämistä varten muodostetaan *HttpConnection*-tyyppinen yhteys palvelimelle sijoitettuun *JSP*-sovelmaan *Connector*-luokan *open*-metodilla. Tämän seurauksena *HTTP*-yhteys siirtyy alustustilaan, eikä sitä ole vielä avattu. Alustilassa asetetaan tarvittavat *HTTP*-otsakekentät. Seuraavaksi avataan kirjoitusvirta kutsumal-

la metodia *openOutputStream*, joka siirtää yhteyden tilaan yhdistetty. *XML*-dokumentti kirjoitetaan palvelimelle *OutputStream*-luokan metodilla *write*, jolle välitetään parametrina *XML*-dokumentti tavutaulukkona.

```
<root>
  <toimittaja>
    <toimittajanro>11</toimittajanro>
  </toimittaja>
</root>
```

Kuva 5.11: Palvelupyynnössä välitettävä *XML*-dokumentti.

Kuvan 5.4 *Tomcat*-palvelimella oleva *JSP*-sovelma *Toimittajahaku* vastaanottaa asiakassovelluksen palvelupyynnön muodostavan *XML*-dokumentin. *JSP*-sovelma käyttää sovelluslogiikan muodostavan *JavaBeans*-komponentin *ToimittajaBean* palveluja hakiessaan halutun toimittajan tiedot tietovarastosta. *JSP*-sovelmassa *Toimittajahaku* *JavaBeans*-komponentti *ToimittajaBean* otetaan käyttöön seuraavalla lauseella

<jsp:useBean id="parser" class="JavaBeans.ToimittajaBean" />.

Lause tarvitsee kaksi attribuuttia *id* ja *class*. Attribuutti *id* määrittelee tunnuksen, jolla *JavaBeans*-komponenttia käytetään *JSP*-sovelmassa ja attribuutti *class* määrittelee käytettävän *JavaBeans*-komponentin luokkatiedoston. *JavaBeans*-komponentin käyttöönotto aiheuttaa sen muodostimen suorituksen. *JSP*-sovelmasta asiakassovelluksen pyynnön muodostava *XML*-dokumentti välitetään *ToimittajaBean*-komponentille kutsumalla sen *getToimittaja*-metodia. Kuvassa 5.12 esitetään, miten *XML*-dokumentti välitetään *JSP*-sovelmalta *JavaBeans*-komponentille.

```
<%@ page import="JavaBeans.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<jsp:useBean id="parser" class="JavaBeans.ToimittajaBean" />

<%
  // Haetaan toimittajaolio.
  Toimittaja toimittaja = parser.getToimittaja(request.getReader());
  // Muodostetaan XML-dokumentti vastaukselle.
  ...
```

Kuva 5.12: *JSP*-sovelma Toimittajahaku.

ToimittajaBean-komponentissa *XML*-dokumentin ohjelmallinen käsittely aloitetaan jäsentämällä dokumentti *Apache Xerces XML*-jäsentimellä tapahtumapohjaisen *SAX*-rajapinnan avulla:

1. Jäsennyksen suorittavan luokan täytyy toteuttaa sisällönkäsittelijärajapinta (ContentHandler). Käsittelijät ovat takaisinkutsuluokkia, jotka mahdollistavat sovelluskohtaisen ohjelmankoodin liittämisen *SAX*:in jäsennystapahtumiin.
2. Luodaan lukija-olio (XMLReader).
3. Rekisteröidään sisällönkäsittelijät lukija-olioon.
4. Luodaan *InputSource*-luokan ilmentymä.
5. Aloitetaan *XML*-dokumentin jäsennys kutsumalla *XMLReader*-luokan metodia parse, jolle välitetään parametrina askeleessa 4 luotu *InputSource*-luokan ilmentymä.

```
public class ToimittajaBean extends DefaultHandler
  {
  ToimittajaDB db;
  Toimittaja toimittaja = null;
  boolean lue = false;

  public ToimittajaBean() {
    db = new ToimittajaDB();
    }

  public Toimittaja getToimittaja(Reader reader) throws SAXException, IOException
    {
    XMLReader xmlReader = new SAXParser();
    xmlReader.setContentHandler(this);
    xmlReader.parse(new InputSource(reader));
    return toimittaja;
    }

  // Takaisinkutsumetodit: katso liite 2
```

Kuva 5.13: *JavaBeans*-komponentti *ToimittajaBean.*

Kuvassa 5.13 kuvattavan jäsennyksen suorittavan luokan *ToimittajaBean* täytyy toteuttaa *ContentHandler*-rajapinta. Tämä rajapintaluokka sisältää takaisinkutsumetodit, jotka koskevat dokumentin sisältöä. Jos rajapintaa käytetään suoraan, täytyy sovelluksen toteuttaa kaikki tässä rajapinnassa määritellyt metodit. Tästä syystä *ToimittajaBean*-luokka periytyy luokasta *DefaultHandler*, joka toteuttaa mm. *ContentHandler*-rajapinnan. *DefaultHandler*-luokassa on tyhjät toteutukset kaikille *ContentHandler*-rajapinnassa määritellyille metodeille. Tästä syystä sovelluksen tarvitsee toteuttaa vain ne takaisin-

kutsumetodit, joita ohjelmassa todella tarvitaan, jotta saadaan tarvittavat tiedot poimittua *XML*-dokumentista.

Kuvan 5.12 *JSP*-sovelman luodessa ilmentymän kuvan 5.13 *ToimittajaBean*-luokasta, sen muodostimessa luodaan ilmentymä kuvan 5.6 *ToimittajaDB*-luokasta. Seuraavaksi *JSP*-sovelma kutsuu *ToimittajaBean*-luokan metodia *getToimittaja*. Tämä metodi luo lukija-olion luokasta *XMLReader* ja rekisteröi sisällönkäsittelijän *ContentHandler* lukijaan metodilla *setContentHandler*. Seuraavaksi kutsutaan luokan *XMLReader* metodia *parse*, joka aloittaa dokumentin jäsennyksen. Lopuksi toteutetaan *ToimittajaBean*-luokkaan tarvittavat *ContentHandler*-rajapinnan määrittelemistä takaisinkutsumetodeista. Takaisinkutsumetodit ovat metodeja, joita sovellus ei itse kutsu suoraan, vaan jäsennin kutsuu niitä tiettyjen tapahtumien kohdalla jäsentämisen edetessä.

```
<root>
  <toimittaja>
    <toimittajanro>11</toimittajanro>
    <toimittajanimi>Matti J Mainio</toimittajanimi>
    <lahiosoite>Joensuu</lahiosoite>
  </toimittaja>
</root>
```

Kuva 5.14: Vastausviestissä välitettävä *XML*-dokumentti.

ToimittajaBean-luokan metodi *getToimittaja* palauttaa haetun toimittajan tiedot *Toimittaja*-tyyppisessä oliossa. *JSP*-sovelmassa *Toimittajahaku* vastausviestin sisältävästä *Toimittaja*-oliosta muodostetaan *XML*-dokumentti kirjoittamalla sen jäsenmuuttujien arvot *ByteArrayOutputStream*-tyyppiseen olioon kuten liitteessä 2 on esitetty. *ByteArrayOutputStream*-olio muunnetaan tavutaulukoksi metodilla *toByteArray*. Lopuksi muodostetaan tulostusvirta *response*-olion avulla, jolla *XML*-dokumentti lähetetään vastauksena asiakassovellukselle. Kuvassa 5.14 kuvataan vastausviestissä välitettävä *XML*-dokumentti.

Kuvassa 5.15 kuvataan miten asiakassovellus käsittelee palvelinsovelluksen palauttaman vastausviestin. Asiakassovelluksessa muodostetaan lukuvirta kutsumalla metodia *openInputStream*, joka siirtää yhteyden tilaan yhdistetty. Palvelimen palauttama *XML*-dokumentti välitetään *DBXMLParser*-luokan muodostimelle, jossa *XML*-dokumentista

muodostetaan jäsennyspuu. *DBXMLparser*-luokan metodin *annaKentanArvo* avulla toimittajatiedot haetaan *XML*-dokumentista ja tiedoista muodostetaan *Toimittaja*-tyyppinen olio, joka tallennetaan *RMS*-tietuevarastoon. *XML*-dokumentin jäsennyksestä huolehtiva *DBXMLParser*-luokka esitetään liitteessä 1. *MIDP*-laitteiden muistirajoitukset asettavat jäsentimille omia vaatimuksia ja sen vuoksi tavalliset jäsentimet eivät useinkaan kelpaa. Erikoisesti *MIDP*-laitteille soveltuvia jäsentimiä on kuitenkin saatavilla ja tässä tutkimuksessa käytetään *MIDP*-laitteille suunnattua Enhydran puupohjaista *kXML*-jäsennintä (Enhydra, 2003a).

```
// Luetaan vastaus palvelimelta.
int rc = con.getResponseCode();
if (rc == HttpConnection.HTTP_OK) {
  in = con.openInputStream();
  DBXMLParser p = new DBXMLParser(in);
  String numero     = p.annaKentanArvo("toimittaja", "toimittajanro", 0);
  String nimi       = p.annaKentanArvo("toimittaja", "toimittajanimi", 0);
  String lahiosoite = p.annaKentanArvo("toimittaja", "lahiosoite", 0);
  Toimittaja toimittaja = new Toimittaja(numero, nimi, lahiosoite);
  // Tallennetaan toimittajatiedot tietuevarastoon.
```

Kuva 5.15: Vastausviestin käsittely asiakassovelluksessa.

5.1.5 XML-RPC -pohjaisen yhteyskäytännön toteutus

XML-RPC on *XML*-pohjainen proseduurien etäkutsuyhteyskäytäntö, joka mahdollistaa yksinkertaisen tavan muodostaa proseduurien etäkutsuja sisältäviä viestejä ja niiden vastauksia. *XML-RPC* -viesti välitetään *HTTP-POST* -muotoisena viestinä, jonka runko on *XML*:ää. Asiakassovellus lähettää viestin palvelinsovellukselle, joka suorittaa pyydetyt toiminnot ja lähettää vastauksen *HTTP*-viestinä, jonka runko on myös *XML*:ää. *XML-RPC* -toteutuksia löytyy useimmille suosituimmille kielille ja teknologioille. *MIDP*-ympäristö ei sisällä sisäänrakennettua *XML-RPC* -toteutusta. Tässä tutkimuksessa käytetään *Enhydran* resursseiltaan rajoittuneeseen *MIDP*-ympäristöön tarkoitettua *kXML-RPC* -toteutusta, joka on toteutettu *kXML*-jäsentimen päälle (Enhydra, 2003c).

Kuvassa 5.16 esitetään miten asiakassovelluksessa muodostetaan *XML-RPC* -palvelupyyntö ja lähetetään se palvelimelle. Asiakassovelluksessa käytetään *kXML-RPC* – kirjaston *XmlRpcClient*-luokkaa määrittelemällä aluksi palvelin, johon halutaan muodostaa yhteys. Palvelin määritellään luomalla ilmentymä luokasta *XmlRpcClient*, jolle

välitetään parametrina palvelimen *ULR*-osoite. Palvelupyyntö sarjallistetaan kuljetusta varten *XmlRpcClient*-luokan *execute*-metodilla. Tämä metodi vaatii kaksi parametria: kutsuttavan käsittelijäluokan tunnisteen ja metodin yhtenä merkkijonona sekä etämetodille välitettävät parametrit *Vector*-tyyppisessä oliossa. Luokan tunnisteena käytetään sitä nimeä, jolla käsittelijäluokka on rekisteröity kuvan 5.19 ilmaisemalla tavalla palvelimella.

```
// Määritellään palvelin
XmlRpcClient xmlrpc = new XmlRpcClient(url);

// Luodaan palvelupyyntö
Vector params = new Vector();
params.addElement(toimittajanro);

//Lähetetään palvelupyyntö ja tulostetaan vastaus.
Object o = xmlrpc.execute("rpcToimittaja.haeToimittaja", params);
Hashtable result = (Hashtable) o;
Toimittaja toimittaja = new Toimittaja((String) result.get("numero),
                                       (String) result.get("nimi"),
                                       (String) result.get("osoite"));

// Tallennetaan toimittajatiedot RMS-tietuevarastoon
```

Kuva 5.16: *XML-RPC* -palvelupyynnön muodostus ja vastauksen käsittely asiakassovelluksessa.

Kuvassa 5.17 kuvataan kuljetusta varten sarjallistettu *XML-RPC* -etäproseduurikutsu, jonka asiakassovellus lähettää palvelimelle. Palvelin etsii pyydetyn luokan ja yrittää etsiä siitä vaadittua metodia. Jos sopivan niminen metodi löytyy, sen parametrityyppejä verrataan palvelupyynnön parametreihin.

XML-RPC -toteutus ja käsittelijä sijaitsevat *Tomcat*-palvelimella kuvassa 5.4 kuvatulla tavalla. *XML-RPC* -maailmassa käsittelijäksi kutsutaan luokkaa, jonka metodeja etäkutsutaan *RPC*:n yli (Laurent & al., 2001). Kuvassa 5.18 kuvataan esimerkkisovelluksessa käytetty käsittelijäluokka *RPCToimittaja*, jonka *haeToimittaja*-metodia asiakassovellus etäkutsuu kuvan 5.16 mukaisesti. Metodille välitetään parametrina toimittajanumero. Aluksi metodissa luodaan ilmentymä tietovarastoluokasta *ToimittajaDB*. Seuraavaksi tietovarastosta haetaan parametrina saatua toimittajanumeroa vastaavan toimittajan tiedot. Metodi *haeToimittaja* palauttaa toimittajatiedot *Hashtable*-tyyppisessä oliossa.

```
POST /gradu/RPCServlet HTTP/1.1
Host: cs.joensuu.fi:18080
Content-type: text/xml
Content-Length: 180

<medhodCall>
  <medhodName>rpcToimittaja.haeToimittaja</medhodName>
    <params>
      <param>
        <value>
          <string>11</string>
        </value>
      </param>
    </params>
  </medhodName>
</medhodCall>
```

Kuva 5.17: *XML-PRC* -etäproseduurikutsu sarjallistettuna.

XML-RPC -palvelimen tehtävänä on kuunnella asiakassovelluksen tekemiä palvelupyyntöjä ja välittää niitä käsittelijöille. *XML-RPC* -palvelin voidaan toteuttaa joka käyttämällä *XML-RPC* -toteutukseen sisäänrakennettua *WebServer*-luokkaa pyyntöjen välittämiseen käsittelijöille tai käyttämällä *sovelluspalvelimelle* sijoitettua servlettiä välittämään *XML-RPC* -pyyntöjä käsittelijälle (Apache, 2003b). Tässä tutkimuksessa käytetään *servlettiä RPCServlet*, joka vastaanottaa asiakassovelluksen lähettämät palvelupyynnöt ja välittää palvelupyynnön edelleen käsittelijälle *RPCToimittaja*. *RPCServlet*-luokka on sijoitettu *Tomcat*-palvelimelle kuvan 5.4 mukaisesti.

```
public class RPCToimittaja {
  public Hashtable haeToimittaja(String numero) {
    // Luodaan tietovarasto.
    ToimittajaDB db = new ToimittajaDB();
    Hashtable t = new Hashtable();
    // Haetaan toimittajan tiedot tietovarastosta.
    Toimittaja toimittaja = db.getToimittaja(numero);
    t.put("numero", toimittaja.getNumero());
    t.put("numero", toimittaja.getNumero());
    t.put("numero", toimittaja.getNumero());
    // Palautetaan haetun toimittajan tiedot.
    return t;
    }
  }
```

Kuva 5.18: *XML-RPC* -käsittelijäluokan toteutus palvelimella.

RPCServlet-luokan rakenne esitetään kuvassa 5.19. *Servletin init*-metodissa luodaan *XML-RPC* -toteutuksen sisältämästä *XmlRpcServer*-luokasta ilmentymä, jonka avulla palvelin voi vastaanottaa *XML-RPC* -etäkutsuja. Seuraavaksi *XmlRpcServer*-luokan *addHandler*-metodilla rekisteröidään *RPCToimittaja*-luokka käsittelijäksi, jota asiakas-

sovellus etäkutsuu. Metodille *addHandler* välitetään parametrina halutun luokan tunniste ja ilmentymä käsittelijäluokasta. Rekisteröinnissä etämetodille asetetaan tunnisteeksi *rpcToimittaja*, jota asiakassovelluksen on käytettävä kuvan 5.16 mukaisesti suorittaessaan etäkutsuja tälle palvelulle.

```
public class RPCServlet extends HttpServlet
  {
  XmlRpcServer xmlrpc;

  public void init(ServletConfi confic)
    throws ServletException
    {
    // Luodaan XmlRpcServer-olio
    xmlrpc = new XmlRpcServer();
    // Rekisteröidään käsittelijä
    xmlrpc.addHandler("rpcToimittaja", new RPCToimittaja());
    super.init(config);
    }

  public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
    {
    // Pyyntö asiakkaalta ja suoritetaan execute-metodi.
    byte [] result = xmlrpc.execute(request.getInputStream());
    // Muodostetaan vastaus ja lähetetään se asiakassovellukselle.
    response.setContentType("text/xml");
    response.setContentLength(result.length);
    OutputStream out = response.getOutputStream();
    Out.write(result);
    Out.close();
    }
  }
```

Kuva 5.19: *RPCServlet*-luokan toteutus palvelimella.

Asiakassovelluksen palvelupyynnön seurauksena suoritetaan *RPCServlet*-luokan metodi *doPost*, jonka *HttpServletRequest*-tyyppinen olio sisältää asiakassovelluksen palvelupyynnön sisältämän *XML*-datan. Metodissa *doPost* suoritetaan *XmlRpcServer*-luokan *execute*-metodi, jolle välitetään parametrina *HttpServletRequest*-rajapinnan *InputStream*-olio. Tämän olion avulla luetaan *POST*-metodin mukana tullut *XML*-muotoinen palvelupyyntö. Metodissa *execute XML*-muodossa vastaanotettu palvelupyyntö jäsennetään ja suoritetaan kuvan 5.16 mukaisesti etäkutsuttava pyynnön käsittelijä *haeToimittaja*, joka palauttaa toimittajatiedot *Hashtable*-tyyppisessä oliossa. Metodissa *execute* käsittelijän palauttama *Java*-tietotyyppi sarjallistetaan *XML*-muotoon tiedonsiirron ajaksi. *Hashtable*-tyyppinen olio sarjallistetaan *XML-RPC* -tyypiksi tietue (Laurent & al., 2001). Seuraavaksi *RPCServlet*-luokassa asetetaan vastausviestille tarpeelliset *HTTP*-otsakekentät ja muodostetaan kirjoitusvirta, jolla vastausviesti kirjoitetaan asiakassovel-

lukselle. Kuva 5.20 sisältää *XML-RPC* -vastausviestin rakenteen, joka koostuu *HTTP*-otsakekentistä ja varsinaisesta vastausviestistä *XML*-muodossa.

```
HTTP/1.1 200 OK
Server: Apache Tomcat/4.1.29 (HTTP/1.1 Connector
Status: 200
Date: Tue, 18 Nov 2003 08:50:52 GMT
Content-Type: text/xml
Content-Length: 314

<?xml version="1.0" encoding="ISO-8859-1"?>
<medhodResponse>
 <params>
  <param>
   <value>
    <struct>
     <member>
      <name>numero</name><value>11</value>
     </member>
     <member>
      <name>nimi</name><value>Seppo Nieminen</value>
     </member>
     <member>
      <name>osoite</name><value>Joensuu</value>
     </member>
    </struct>
   </value>
  </param>
 </params>
</medhodResponse>
```

Kuva 5.20: *XML-RPC* -toteutuksen generoiman vastausviestin rakenne.[1]

Palvelinsovelluksen lähettämä vastausviesti käsitellään asiakassovelluksessa kuvassa 5.16 kuvatulla tavalla. *XmlRpcClient*-luokassa tiedonsiirtoa varten kuvan 5.20 mukaisesti sarjallistettu vastausviesti puretaan *Object*-tietotyypiksi, joka edelleen muunnetaan *Hashtable*-tietotyypiksi. *Hashtable*-tyyppisen olion sisältämistä toimittajatiedoista muodostetaan *Toimittaja*-tyyppinen olio. Lopuksi *Toimittaja*-tyyppisen olion sisältö tallennetaan päätelaitteen *RMS*-tietuevarastoon.

[1] Vastausviesti tulostettiin *WTK*-kehitysympäristön emulaattoriin.

5.1.6 SOAP-pohjaisen yhteyskäytännön toteutus

SOAP on yksinkertainen ja kevyt *XML*-pohjainen yhteyskäytäntö, joka on suunniteltu rakenteisen ja tyypitetyn tiedon siirtoa ja proseduurien etäkutsujen suorittamista varten keskittämättömissä ja hajautetuissa ympäristöissä. *SOAP*-viesti välitetään *HTTP-POST* -muotoisena viestinä, jonka runko on *XML*:ää. Asiakassovellus lähettää viestin palvelinsovellukselle, joka suorittaa pyydetyt toiminnot ja lähettää vastauksen *HTTP*-viestinä, jonka runko on myös *XML*:ää. *SOAP*-toteutuksia löytyy useimmille suosituimmille kielille ja teknologioille. *MIDP*-ympäristö ei sisällä sisäänrakennettua *SOAP*-toteutusta. Tässä tutkimuksessa käytetään *Enhydran* resursseiltaan rajoittuneeseen *MIDP*-ympäristöön tarkoitettua *kSOAP*-toteutusta (Enhydra, 2003b). Yuan (2002) esittää, että *SOAP*-yhteyskäytännön avulla toteutettu proseduurien etäkutsu koostuu seuraavista vaiheista:

1. Asiakassovellus muodostaa *SOAP*-pyyntöviestin, joka sisältää etäkutsuttavalle metodille välitettävät parametrit.
2. Asiakassovellus lähettää *SOAP*-pyyntöviestin *SOAP*-palvelimelle käyttäen jotain rinnakkaisyhteyskäytäntöä tiedonsiirtoon. Yleisimmin käytetty tiedonsiirtoyhteyskäytäntö *SOAP*:in yhteydessä on *HTTP*.
3. *SOAP*-palvelin kutsuu viestissä määriteltyä palvelua ja välittää palvelulle *SOAP*-pyyntöviestistä poimimansa parametrit.
4. *SOAP*-palvelin muodostaa etämetodin palauttamista arvoista *SOAP*-vastausviestin ja lähettää sen asiakassovellukselle. Käytettäessä tiedonsiirtoyhteyskäytäntönä *HTTP*:tä, samaa *HTTP*-yhteyttä voidaan käyttää sekä pyyntöön että vastaukseen.
5. Asiakassovellus vastaanottaa *SOAP*-vastausviestin ja jäsentää paluuarvot *Java*-olioiksi.

Kuvassa 5.21 esitetään, miten asiakassovelluksessa muodostetaan *SOAP*-palvelupyyntö ja lähetetään se palvelimelle. *HttpTransport*-luokka on *kSOAP*-kirjaston osa, jonka avulla muodostetaan *HTTP*-yhteys palvelimelle. *ClassMap*-luokan metodeja käytetään omien tietotyyppien sarjallistamiseen kuljetusta varten ja sarjallistamisen purkuun. *SoapObject*-luokan avulla muodostetaan *SOAP*-pyyntö. Aluksi perustetaan *HTTP*-yhteys palvelimelle luomalla ilmentymä *HttpTransport*-luokasta. Luokan muodostimelle väli-

tetään parametreina *SOAP*-palvelimen *URL*-osoite ja *SOAP*-pyynnön *HTTP*-otsakekentälle *SOAPAction* välitettävä arvo. Seuraavaksi rakennetaan SOAP-palvelupyyntö luomalla ilmentymä *SoapObject*-luokasta. Luokan muodostimelle välitetään parametreina nimiavaruus ja *SOAP*-palvelun nimi. Nimiavaruus määritellään *SOAP*-palvelun kuvaustiedostossa kuvan 5.24 mukaisesti ja *SOAP*-palvelun nimi määrittelee *SOAP*-viestin runkoelementin lapsielementin nimen. Seuraavaksi *SoapObject*-olioon asetetaan kutsuttavalle etämetodille välitettävänä parametrina haettavan toimittajan toimittajanumero. Parametrit asetetaan *SoapObject*-luokan *addProperty*-metodilla, jolle välitetään parametreina elementin nimi ja arvo.

```
String url = "http://cs.joensuu.fi:18080/soap/servlet/rpcrouter";

// Muodostetaan HTTP-yhteys palvelimelle
HttpTransport ht = new HttpTransport(url, "urn:toimittaja-db#getToimittaja");
ClassMap classMap = new ClassMap();
ht.setClassMap(classMap);

// Rakennetaan SOAP-palvelupyyntö.
SoapObject rpc = new SoapObject("urn:toimittaja-db", "getToimittaja");
rpc.addProperty("toimittajanro", toimittajanro);

// Lähetetään palvelupyyntö ja käsitellään vastaus.
try {
  SoapObject tout = (SoapObject) ht.call(rpc);
  // Muodostetaan Toimittaja-olio vastauksesta.
  Toimittaja toimittaja = new Toimittaja((String) tout.getProperty("numero"),
                                         (String) tout.getProperty("numero"),
                                         (String) tout.getProperty("numero"));

  // Tallennetaan toimittajatiedot RMS-tietuevarastoon.
  ...
  }
catch (Exception e) {
  // Käsitellään poikkeus.
  }
```

Kuva 5.21: *SOAP*-palvelupyynnön muodostus asiakassovelluksessa.

SOAP-palvelupyyntö lähetetään *SOAP*-palvelinsovellukselle *HttpTransport*-luokan metodilla *call*. Metodille välitetään parametrina edellä luotu *SoapObject*-olio, joka sisältää tiedot suoritettavasta etäproseduurikutsusta. *HttpTransport*-luokassa *SoapObject*-olion sisältämä etäproseduurikutsu sarjallistetaan ennen palvelupyynnön lähettämistä *SOAP*-viestiksi. Kuvassa 5.22 kuvataan *SOAP*-palvelinsovellukselle lähetettävän palvelupyynnön rakenne. *HttpTransport*-luokan *call*-metodi palauttaa *SoapObject*-tyyppisen olion, joka sisältää *SOAP*-palvelinsovelluksen lähettämän paluuviestin.

```
POST /soap/servlet/rpcrouter HTTP/1.1
User-Agent: kSOAP/1.0
SOAPAction: urn:toimittaja-db#getToimittaja
Host: cs.joensuu.fi:18080
Content-type: text/xml
Content-Length: 526

<SOAP-ENV:Envelope
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema
  xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/>
      <getToimittaja xmlns="urn:toimittaja-db" id="o0" SOAP-ENC:root="1">
        <toimittajanro xmlns="" xsi:type="xsd:string">11</toimittajanro>
      </getToimittaja>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Kuva 5.22: *SOAP*-palvelupyynnön rakenne.

Palvelinpuolen *SOAP*-toteutukseen käytetään *Apachen SOAP* versiota 2.2. Palvelinpuolen *SOAP* vaatii toimiakseen *servlet*-palvelimen. Tässä tutkielmassa käytetään *Tomcat*-palvelinta kuvassa 5.4 kuvatulla tavalla. Kuvassa 5.23 esitetään esimerkkisovelluksessa käytetty *SOAP*-palvelu, jonka *getToimittaja*-metodia asiakassovellus kutsuu *RPC*:n yli kuvan 5.21 mukaisesti. Metodille välitetään parametrina toimittajanumero ja se palauttaa haetut toimittajatiedot *Toimittaja*-tyyppisessä oliossa. Näin käytettynä *SOAP* toimii kehittyneenä *XML-RPC*-järjestelmänä, jossa on tuki omien tietotyyppien sarjallistamiselle.

```
public class ToimittajaSoap
  {
  private ToimittajaDB toimittajaDB = null;

  public ToimittajaSoap() {
    toimittajaDB = new ToimittajaDB();
    }

  public Toimittaja getToimittaja(String numero) {
    return toimittajaDB.getToimittaja(numero);
    }
  }
```

Kuva 5.23: *SOAP*-palvelun toteutus.

Kuvassa 5.24 esitetään *SOAP*-palvelun *XML*-pohjainen kuvaustiedosto (Deployment Descriptor), jossa kerrotaan palvelusta seuraavat asiat *SOAP*-toteutukselle: *SOAP*-palvelun *URN*, asiakasohjelmille tarjottavat metodit ja ohjeet omien tietotyyppien sarjallistamiseen sekä sarjallistamisen purkuun (Yuan, 2002). Kuvaustiedoston alussa viita-

taan *Apachen SOAP*-nimiavaruuteen. Sen jälkeen palvelulle annetaan *URN*-tunniste, jota asiakasohjelmien on käytettävä ottaessaan yhteyttä *SOAP*-palvelinsovellukseen. Seuraavana esitellään asiakasohjelmille tarjottavien metodien luettelo, joka kertoo asiakasohjelmille ja palvelimelle mitkä metodikutsut ovat sallittuja. Seuraavaksi *java*-elementin *class*-attribuutin avulla määritellään palvelun toteuttavan luokan nimi ja *static*-attribuutin avulla kerrotaan, että luokan metodit eivät ole staattisia. Lopuksi *mappings*-elementti määrittelee, miten *SOAP*-toteutuksen täytyy sarjallistaa omat tietotyypit. Attribuutilla *javaType* palvelimelle kerrotaan uuden tietotyypin *Java*-luokan nimi, tässä tapauksessa *Toimittaja*. Koska sarjallistettava *Toimittaja*-luokka on *JavaBeans*-ehtojen mukainen, voidaan se sarjallistaa *Apachen SOAP*:in mukana tulevan *BeanSerializer*-luokan avulla. *Java*-luokka on *JavaBeans*-ehtojen mukainen, jos sillä on parametriton muodostin ja kaikkiin sen jäsenmuuttujiin päästään käsiksi *setXXX*- ja *getXXX*-tyyppisillä metodeilla (Apache, 2003a).

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
     id="urn:toimittaja-db">

  <isd:provider type="java" scope="Application" methods="getToimittaja">
    <isd:java class="ToimittajaSoap" static="false" />
  </isd:provider>

  <isd:faultListener>org.apache.soap.server.DOMFaultListener</isd:faultListener>

  <isd:mapping>
    <isd:map encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
             xmlns:x="urn:toimittaja-db-viite" qname="x:toimittaja"
             javaType="Toimittaja"
             java2XMLClassName="org.apache.soap.encoding.soapenc.BeanSerializer"
             xml2JavaClassName="org.apache.soap.encoding.soapenc.BeanSerializer"/>
  </isd:mapping>
</isd:service>
```

Kuva 5.24: *SOAP*-palvelun kuvaustiedoston toteutus.

SOAP-toteutus vastaanottaa asiakassovelluksen lähettämän palvelupyynnön ja jäsentää *SOAP*-pyyntöviestin sisältämän etäproseduurikutsun. Seuraavaksi *SOAP*-toteutus kutsuu viestissä määriteltyä etämetodia *getToimittaja* ja välittää metodille *SOAP*-pyyntöviestistä poimimansa parametrin *toimittajanumero*. Etämetodi palauttaa haetut toimittajatiedot *Toimittaja*-tyyppisen olion jäsenmuuttujissa. SOAP-toteutus sarjallistaa *Toimittaja*-olion *SOAP*-vastausviestiksi. Kuva 5.25 sisältää *SOAP*-vastausviestin rakenteen, joka koostuu *HTTP*-otsakekentistä ja varsinaisesta vastausviestistä *XML*-muodossa.

```
HTTP/1.1 200 OK
Server: Apache Tomcat/4.1.29 (HTTP/1.1 Connector
Status: 200
Date: Mon, 17 Nov 2003 11:12:21 GMT
Set-Cookie: JSESSIONID=6E6AEED3B053F760B5349A99C7CDBF6C;Path=/soap
Content-Type: text/xml; charset=utf-8
Content-Length: 632

<SOAP-ENV:Envelope
  xmlns:SOAP-ENV=http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Body>
      <ns1:getToimittajaResponse xmlns:ns1="urn:toimittaja-db"
        SOAP-ENV:encodingStyle=http://schemas.xmlsoap.org/soap/encoding/>
        <return xmlns:ns2="urn:toimittaja-db-viite" xsi:type=ns2:toimittaja">
          <nimi xsi:type="xsd:string">Seppo Nieminen</nimi>
          <numero xsi:type="xsd:string">11</numero>
          <osoite xsi:type="xsd:string">Joensuu</osoite>
        </return>
      </ns1:getToimittajaResponse>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

Kuva 5.25: *SOAP*-vastausviestin rakenne.[2]

Palvelinsovelluksen lähettämä vastausviesti käsitellään asiakassovelluksessa kuvassa 5.21 kuvatulla tavalla. Asiakassovelluksessa *HttpTransport*-luokan *call*-metodi vastaanottaa kuvassa 5.25 kuvatun vastausviestin ja muodostaa siitä *SoapObject*-tyyppisen olion. *SoapObject*-olion sisältämistä toimittajatiedoista muodostetaan *Toimittaja*-tyyppinen olio käyttämällä *SoapObject*-luokan *getProperty*-metodia. Lopuksi *Toimittaja*tyyppisen olion sisältö tallennetaan päätelaitteen paikalliseen *RMS*-tietuevarastoon.

5.2 Tiedonsiirtoformaattien vertailu

Tutkielman kokeellisessa osuudessa tarkastellaan kohdassa 4.2 esiteltyjen yhteyskäytäntöjen soveltamista matkapuhelimen ja palvelimen väliseen kommunikointiin. Tarkastelun tavoitteena on selvittää *XML*-pohjaisten yhteyskäytäntöjen vaikutus siirrettävän tiedon määrään ja suorituskykyyn. Empiirisessä tutkimuksessa selvitetään *MIDP*-asiakassovelluksen ja palvelinsovelluksen väliseen tiedonsiirtoon käytetty aika. Lisäksi mitataan aika, joka kuluu palvelimelta haettujen toimittajatietojen tallentamiseen päätelaitteen paikalliseen *RMS*-tietuevarastoon. Palvelupyynnön suoritusaikaan sisältyy kuvan 5.3 mukaisesti seuraavat vaiheet:

- asiakassovelluksella palvelupyynnön muodostus,
- yhteyden muodostus palvelimelle,
- palveluviestin lähetys palvelimelle,
- palvelimen käyttämä aika viestin käsittelyyn,
- palvelimella vastausviestin muodostus sekä lähetys asiakassovellukselle,
- asiakassovelluksella vastausviestin vastaanotto ja käsittely.

Palvelupyyntö katsotaan saaduksi päätökseen, kun palvelimelta haetuista toimittajatiedoista muodostetaan *Toimittaja*-tyyppinen olio. Empiirisessä tutkimuksessa lasketaan näin saatu yhteisaika eikä eritellä mitenkään palvelupyynnön eri vaiheita. Päätelaitteen paikalliseen tietuevarastoon tallennusaikaan käytettyyn aikaan sisältyy *Toimittaja*-tyyppisen olion sarjallistaminen ja näin saatujen toimittajatietojen tallennus tietuevarastoon. Alakohdassa 5.2.1 esitellään empiirisessä tutkimuksessa käytettävät laitteistot. Tutkimusaineisto esitellään kohdassa 5.2.2 ja lopuksi kohdassa 5.2.3 esitetään empiirisessä tutkimuksessa saadut tulokset.

5.2.1 Tutkimuslaitteistot

Palvelupyynnön suoritusajat mitataan yhteyskäytännöittäin sekä kehitysympäristön emulaattorissa että kahdella todellisella matkapuhelimella. Kehitysympäristömittaus

[2] Vastausviesti tulostettiin *WTK*-kehitysympäristön emulaattoriin.

suoritetaan työasemaympäristössä, jolloin on käytössä kiinteä laajakaistainen verkkoyhteys. Lisäksi työasema tarjoaa mittausympäristön, josta puuttuvat mobiililaitteen ominaispiirteet, kuten hyvin rajoitettu määrä muistia ja suhteellisen heikkotehoinen prosessori. Kehitysympäristötestauksessa käytetään seuraavaa laitteisto- ja ohjelmistokokoonpanoa:

- käyttöjärjestelmä: Microsoft Windows NT,
- prosessori: Intel Pentium III 1.0 GHz,
- keskusmuisti: 256 Mt,
- verkkokortti: 3 com etherlink 10/100 PCI NIC,
- emulaattori: Wireless Toolkit (WTK), versio 1.0.4.

Suoritusajat selvitetään myös todellisilla matkapuhelimilla, jolloin asiakassovelluksen ja palvelinsovelluksen väliseen kommunikointiin on käytettävissä kapeakaistainen verkkoyhteys. Tässä tutkimuksessa mittauslaitteina käytetään *Motorola Accompli 008* ja *Nokia 6610i* -matkapuhelimia. Tutkimuksen tavoitteena ei ole vertailla laitteita keskenään, vaan tavoitteena on vertailla eri tiedonsiirtoyhteyskäytäntöjä eri alustoilla. Tutkimuksessa käytetyt matkapuhelimet ovat tulleet markkinoille eri aikoina, joten laitteiden suorituskyvyssä näkyy matkapuhelinalan voimakas kehitys. Empiirisessä tutkimuksessa käytettävien matkapuhelimien suoritusarvot ja olennaisimmat ominaisuudet esitellään taulukossa 5.1.

Taulukkoa 5.1: Tutkimuksessa käytettyjen mobiilien alustojen ominaisuudet.

Ominaisuus	**Nokia 6610i**	**Motorola Accompli 008**
Saatavuusvuosi	2004	2002
Prosessori	104 Mhz	33 Mhz
Muisti	Kasamuisti 200 kb	Kasamuisti 640 kb
Java-tuki	CLDC/MIDP 1.0	CLDC/MIDP 1.0
Java-sovelluksen koko	Max. 64 kb	-
GPRS	Class 6 (3+1/2+2 slots)	Class 2 (2+1 slots)

5.2.2 Tutkimusaineisto

Tutkielman kokeellisen osuuden tarkoitus on selvittää eri yhteyskäytäntöjen suorituskyky *MIDP*-asiakassovelluksen ja palvelinsovelluksen välisessä tiedonsiirrossa. Kokeellisen osuuden yhtenä tavoitteena on selvittää *XML*-pohjaisten yhteyskäytäntöjen aiheuttaman lisäkuorman vaikutus tiedonsiirron suorituskykyyn. Tästä syystä *perusaineiston* muodostaa mahdollisimman pieni siirrettävä hyötykuorma. Tällöin yhteyskäytännön aiheuttaman lisäkuorman suhteellinen osuus muodostuu suureksi. Toisena tavoitteena on selvittää tiedonsiirron suorituskyvyn riippuvuutta hyötykuorman koosta yhteyskäytännöittäin. Tästä syystä *perusaineiston* hyötykuormaa kasvatetaan laatimalla viisi uutta *laajennettua* aineistoa, jolloin hyötykuorman koko vaihtelee 25 - 8404 tavun välillä kuvan 5.6 ilmaisemalla tavalla. Näitä samoja aineistoja käytetään myös tutkittaessa tiedon tallennuksen tehokkuutta *MIDP*-päätelaitteen paikalliseen *RMS*-tietuevarastoon. Tutkimuksessa käytettävät termit *hyötykuorma*, *lisäkuorma* ja *kokonaiskuorma* määritellään seuraavasti:

- *Hyötykuorma*: Hyötykuormalla tarkoitetaan siirrettävän viestin osaa, jonka siirtämistä varten sovellus yleensä rakennetaan. Tässä tutkielmassa käytetyissä sovelluksissa palvelupyynnön hyötykuormana on toimittajanumero. Vastausviestin hyötykuorma koostuu toimittajanumerosta ja haetuista toimittajatiedoista.
- *Lisäkuorma:* Lisäkuorma aiheutuu yleensä viestin esitysmuodosta ja viestin käsittelyohjeista.
- *Kokonaiskuorma:* Kokonaiskuormalla tarkoitetaan hyötykuorman ja tarkasteltavan yhteyskäytännön aiheuttaman lisäkuorman yhteismäärä. Tässä tutkielmassa tiedonsiirron kokonaiskuormalla tarkoitetaan *HTTP*-viestin runko-osassa kuljetettavaa tietoa.

5.2.2.1 Lisäkuorman vaikutuksen selvittäminen

Asiakassovelluksen palvelupyyntöön kuuluu pyyntöviestin siirto palvelinsovellukselle ja vastausviestin siirto takaisin asiakassovellukselle. Taulukossa 5.2 esitetään tarkastelun kohteena olleiden yhteyskäytäntöjen pyyntö- ja vastausviestien koot perusaineis-

ton osalta eriteltynä hyötykuormaksi, lisäkuormaksi ja kokonaiskuormaksi. *XML*-pohjaiset viestit vaikuttavat palvelupyyntöjen suorituskykyyn sekä tiedonsiirron että niiden vaatiman käsittelyn kautta. Tiedonsiirron suorituskykyyn *XML*-pohjaisuus vaikuttaa sen vaatiman lisäkuorman kautta. Lisäksi *XML*-muodossa siirrettävät viestit täytyy jäsentää sekä palvelinpuolella että asiakaspuolella. Erikoisesti niukkaresurssisilla *MIDP*-laitteilla *XML*-pohjaisen viestin jäsennys voi aiheuttaa merkittävän suorituskyvyn alennuksen.

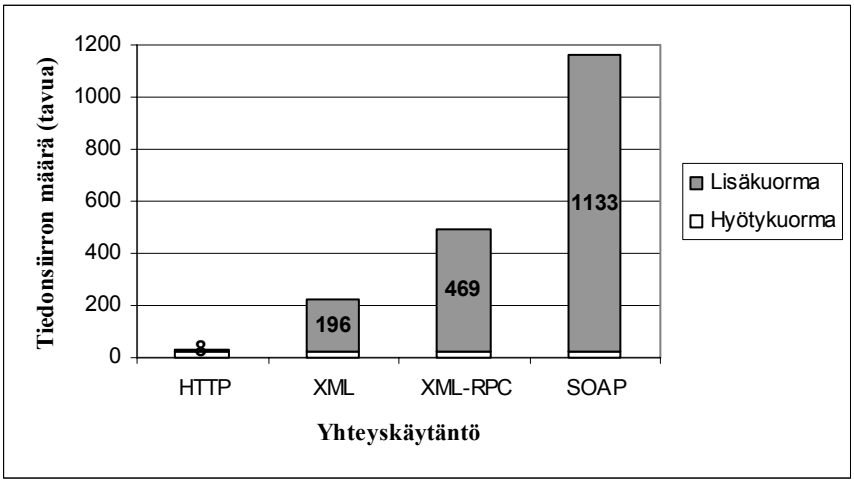
Taulukko 5.2: Perusaineiston viestien koko eri yhteyskäytäntöjä käytettäessä.

Yhteys-käytäntö	Pyyntöviesti				Vastausviesti			
	Hyöty-kuorma (tavua)	Lisä-kuorma (tavua)	Lisä-kuorma (%)	Konais-kuorma (tavua)	Hyöty-kuorma (tavua)	Lisä-kuorma (tavua)	Lisä-kuorma (%)	Kokonais-kuorma (tavua)
HTTP	2	2	100	4	23	6	26	29
XML	2	69	3450	71	23	127	552	150
XML-RPC	2	178	8900	180	23	291	1265	314
SOAP	2	524	26200	526	23	609	2647	632

Kuten taulukosta 5.2 havaitaan, pyyntö- ja vastausviestin kokonaiskuormat ovat suuresti riippuvaisia käytetystä yhteyskäytännöstä. Puhtaasti binäärimuodossa järjestelmien välillä tietoa siirtävä *HTTP*-pohjainen yhteyskäytäntö ei sisällä juurikaan lisäkuormaa pyyntö- ja vastausviesteissä. Lisäkuorman aiheuttaa *DataOutputStream*-luokan metodin *writeUTF* käyttö pyyntö- ja vastausviestien kirjoittamiseen kuvan 5.7 mukaisesti. *DataInputStream*-virta tekee automaattisesti konversiot primitiivityypin ja sen binääriesityksen välillä ja siitä aiheutuu tulostusvirtaan kaksi lisätavua. Tässä tutkielmassa toteutettu *XML*-pohjainen tiedonsiirto on pohjimmiltaan binäärimuotoinen tiedonsiirto, jossa asiakassovelluksen ja palvelinsovelluksen välinen tieto siirretään *XML*-dokumentissa kuvien 5.11 ja 5.14 mukaisesti. Tästä aiheutuu merkkauksen aiheuttamaa lisäkuomaa 69 tavua pyyntöviestiin ja 127 tavua vastausviestiin. *XML-RPC* –pohjaisessa toteutuksessa pyyntöviesti on kuvan 5.17 mukainen ja siinä tarvitaan kahden tavun hyötykuorman siirtämiseen 178 tavua lisäkuormaa. *XML-RPC* -vastausviesti on kuvattu kuvassa 5.20 ja se sisältää 23 tavun hyötykuorman lisäksi 291 tavua lisäkuormaa. *SOAP*-pohjaisessa yhteyskäytännössä hyötykuorman osuus kokonaiskuormasta on pienin. *SOAP*-pohjainen pyyntöviesti on kuvan 5.22 mukainen ja siinä tarvitaan kahden tavun hyötykuorman siirtämiseen 524 tavua lisäkuormaa. Kuvassa 5.25 kuvataan *SOAP*-toteutuksen vastaus-

viesti, jossa tarvitaan 23 tavun hyötykuorman siirtämiseen palvelinsovellukselta asiakassovellukselle lisäkuormaa 609 tavua.

XML-pohjaisten yhteyskäytäntöjen heikkoutena pidetään usein niiden resursseja kuluttavaa tiedon esitysmuotoa. Koska *XML*-dokumentit ovat tekstipohjaisia ja sisältävät lisäkuormana merkitsemistunnisteita sisällön joukossa, dokumenttien koko kasvaa suureksi. Dokumenttien suurella koolla on vaikutusta sekä tiedonsiirtoon että käsittelyn tehokkuuteen. Kuvassa 5.26 kuvataan graafisesti palvelussa tarvittava kokonaiskuorma eriteltynä hyöty- ja lisäkuormaksi yhteyskäytännöittäin laskettuna taulukosta 5.2. *HTTP*-pohjaisessa yhteyskäytännössä kokonaishyötykuorman 25 tavun siirtämiseen tarvitaan lisäkuormaa 8 tavua. *XML*-pohjaisessa yhteyskäytännössä vastaavan hyötykuorman siirtämiseen tarvitaan lisäkuormaa 196 tavua. *XML-RPC* -pohjaisessa yhteyskäytännössä lisäkuorman tarve on 469 tavua ja *SOAP*-pohjaisessa yhteyskäytännössä 1133 tavua.



Kuva 5.26: Perusaineiston palvelupyynnön koko yhteyskäytännöittäin.

5.2.2.2 Tiedonsiirron suorituskyvyn riippuvuus hyötykuorman koosta

Taulukossa 5.3 kuvataan perusaineiston lisäksi muut viisi laajennettua aineistoa, joiden avulla selvitetään hyötykuorman koon vaikutusta tiedonsiirron suorituskykyyn yhteyskäytännöittäin. Asiakassovellus lähettää palvelinsovellukselle kahden tavun suuruisen hyötykuorman, joka on kaikissa aineistoissa samankokoinen. Palvelinsovelluksen lähet-

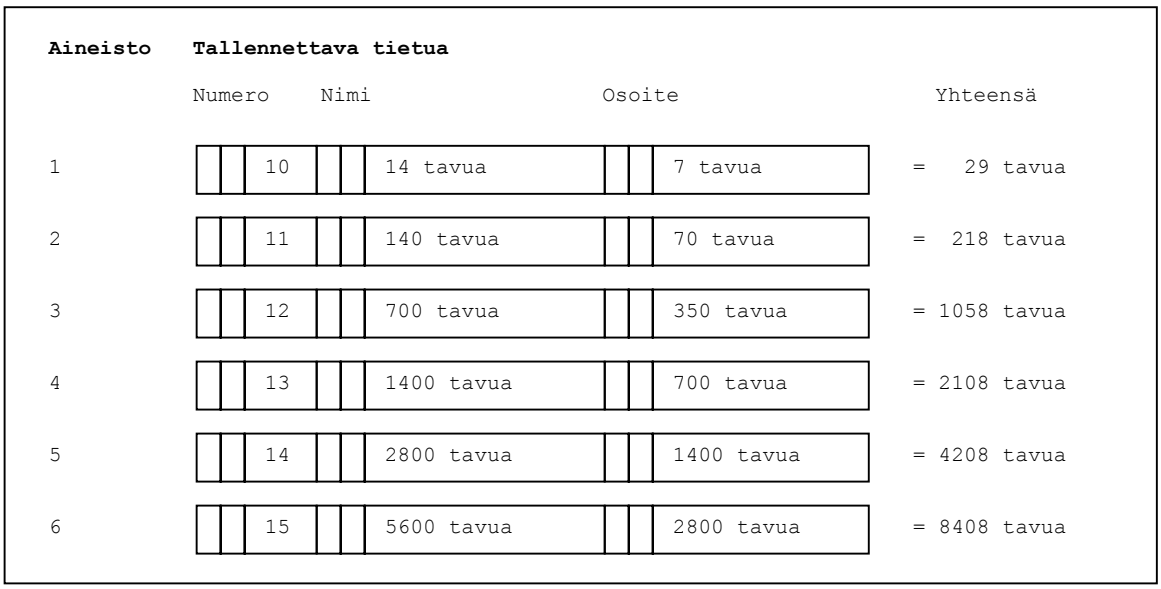
tämää vastausviestin hyötykuormaa kasvatetaan kuvan 5.6 mukaisesti. Aineistossa 1 eli perusaineistossa siirrettävä kokonaiskuorma on *HTTP*-pohjaisessa yhteyskäytännössä 33 tavua, *XML*-pohjaisessa yhteyskäytännössä 221 tavua, *XML-RPC* -pohjaisessa yhteyskäytännössä 494 tavua ja *SOAP*-pohjaisessa yhteyskäytännössä 1158 tavua. Viimeisenä laajennetussa aineistossa siirrettävä kokonaishyötykuorma on 8404 tavua. Tällöin asiakas- ja palvelinsovelluksen välinen kokonaiskuorma on *HTTP*-pohjaisessa yhteyskäytännössä 8412 tavua, *XML*-pohjaisessa yhteyskäytännössä 8600 tavua, *XML-RPC* –pohjaisessa yhteyskäytännössä 8873 tavua ja *SOAP*-pohjaisessa yhteyskäytännössä 9537 tavua.

Taulukko 5.3: Hyöty- ja kokonaiskuorma eri aineistoissa yhteyskäytännöittäin.

Aineisto		Hyötykuorma (tavua)	Kokonaiskuorma (tavua)			
			HTTP	XML	XML-RPC	SOAP
1	Pyyntö	2	4	71	180	526
	Vastaus	23	29	150	314	632
	Yhteensä	25	33	221	494	1158
2	Pyyntö	2	4	71	180	526
	Vastaus	212	218	339	503	821
	Yhteensä	214	222	410	683	1347
3	Pyyntö	2	4	71	180	526
	Vastaus	1052	1058	1179	1343	1661
	Yhteensä	1054	1062	1250	1523	2187
4	Pyyntö	2	4	71	180	526
	Vastaus	2102	2108	229	239	2711
	Yhteensä	2104	2112	2300	2573	3237
5	Pyyntö	2	4	71	180	526
	Vastaus	4202	4208	4329	4493	4811
	Yhteensä	4204	4212	4400	4673	5337
6	Pyyntö	2	4	71	180	526
	Vastaus	8402	8408	8529	8693	9011
	Yhteensä	8404	8412	8600	8873	9537

5.2.2.3 *RMS*-tietuevaraston suorituskyky

Tutkielman kokeellisessa osuudessa selvitetään myös *RMS*-tietuevaraston suorituskyky tiedon tallennuksen osalta. *RMS*-tietuevaraston suorituskyvyn arvioinnissa käytetään taulukossa 5.3 kuvattua kuutta aineistoa. Palvelimen palauttama vastausviestin hyötykuorma tallennetaan jokaisella aineistolla päätelaitteen paikalliseen *RMS*-tietuevarastoon ja mitataan tallennuksen kulunut aika. Palvelimen palauttama vastausviesti sisältää aina kolme merkkijonoa, jotka tallennetaan samaan tietueeseen. Merkkijonojen tallennus toteutetaan pakkaamalla merkkijonot *ByteArrayOutputStream*- ja *DataOutputStream*-tyyppisten olioiden avulla tavutaulukoksi, joka tallennetaan *RMS*-tietuevarastoon. Tavutaulukon muodostus ja tallennus kuvataan tarkemmin liitteessä 1.



Kuva 5.27: Aineiston rakenne *RMS*-tietuevaraston suorituskyvyn mittauksessa.

Kuvassa 5.27 esitetään tallennettavien tietueiden koot aineistoittain. Aineistossa 1 tallennettava hyötykuorma on 23 tavua ja kokonaiskuorma 29 tavua. Lisäkuormana tallennetaan jokaiseen tietueeseen kuusi tavua, koska merkkijonot esitetään *Unicode*-tekstimuodossa (UTF-8). Luokan *DataOutputStream* metodin *writeUTF* kutsu kuvan 3.7 mukaisesti tuottaa aina kaksi lisätavua. Aineistossa 6 tallennettava kokonaiskuorma on 8408 tavua, joka ylittää *MIDP*-määrityksen mukaisen vähimmäiskoon *RMS*-tietuevarastoille.

5.2.3 Tulokset

Tässä kohdassa käydään läpi kokeellisen osuuden tulokset. Alakohdat on jaoteltu ja nimetty mitattavan asian mukaisesti. Alakohdassa 5.2.3.1 tarkastellaan yhteyskäytäntöjen aiheuttamien lisäkuormien vaikutusta suorituskykyyn. Seuraavaksi alakohdassa 5.2.3.2 tarkastellaan hyötykuorman koon vaikutusta suorituskykyyn ja lopuksi alakohdassa 5.2.3.3 tarkastellaan tietojen tallennuksen suorituskykyä päätelaitteen paikalliseen *RMS*-tietuevarastoon. Liitteeseen 3 on koottu taulukoihin kaikki mitatut suoritusajat. Kukin taulukon solu ilmaisee allekkain ajan siirron ja tallennuksen osalta.

5.2.3.1 Lisäkuorman vaikutuksen selvittäminen

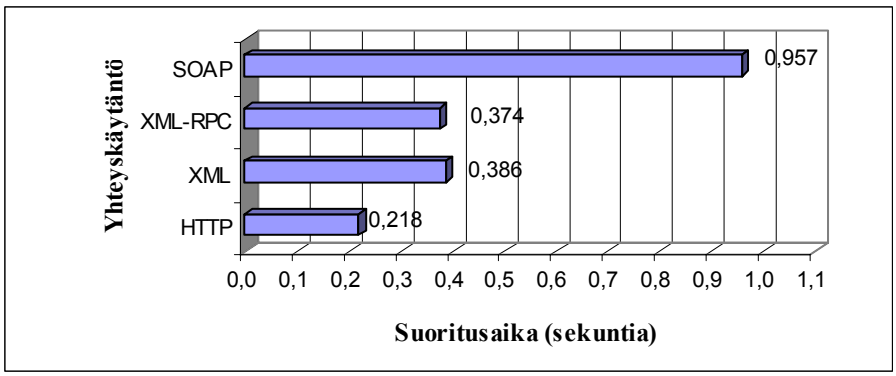
Tarkasteltavan yhteyskäytännön vaikutus lisäkuormana siirrettävän tiedon määrään on huomattava, kuten edellä alakohdassa 5.2.2.1 havaitsimme. *XML*:ään pohjautuvissa tiedonsiirtomenetelmissä kuljetetaan lisäkuormana metatietoa, joka lisää siirrettävän tiedon määrää huomattavasti. Tässä alakohdassa tarkastellaan tämän lisäkuorman vaikutusta tarkastelun kohteina olleiden yhteyskäytäntöjen suorituskykyyn eri päätelaitteilla. Tämä suorituskykyanalyysi ei anna mahdollisuutta vertailla eri laitevalmistajien laitteita keskenään, koska tutkimuksessa käytetyt laitteet ovat eri tehoisia, kuten taulukosta 5.1 voidaan päätellä. Analyysin tavoitteena on ainoastaan selvittää eri yhteyskäytäntöjen suorituskyky usealla päätelaitteella. Taulukossa 5.4 esitetään palvelupyynnön suorittamiseen kulunut aika perusaineiston osalta kullekin yhteyskäytännölle päätelaitteittain kymmenen kokeen tulosten keskiarvona.

Taulukko 5.4: Suorituskykyanalyysin tulokset yhteyskäytännöittäin perusaineistolle.

Yhteyskäytäntö	WTK		Nokia 6610i		Motorola Accompli 008	
	Keskiarvo (sekuntia)	Keskihajonta (sekuntia)	Keskiarvo (sekuntia)	Keskihajonta (sekuntia)	Keskiarvo (sekuntia)	Keskihajonta (sekuntia)
HTTP	0.218	0.023	4.435	0.294	10.038	0.396
XML	0.386	0.032	4.713	0.259	11.365	0.502
XML-RPC	0.374	0.036	5.032	0.352	25.569	1.057
SOAP	0.957	0.021	6.330	0.345	18.084	0.834

Kuten taulukosta 5.4 havaitaan, kaikilla päätelaitteilla *HTTP*-pohjaisella yhteyskäytännöllä toteutettu palvelupyynnön suoritusaika on pienin. *XML*:ään pohjautuvilla tiedonsiirtomenetelmillä palvelun suoritusaika on jokaisessa tapauksessa suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä. *XML*-pohjaisten menetelmien huonompaan suorituskykyyn voidaan katsoa vaikuttavan kaksi tekijää. Ensinnäkin, kuten alakohdassa 5.2.2 osoitettiin, *XML*-pohjaisilla yhteyskäytännöillä joudutaan siirtämään hyötykuorman lisäksi suhteellisen paljon lisäkuormaa. Toiseksi *XML*-pohjaiset viestit täytyy jäsentää sekä asiakaspuolella että palvelinpuolella. Erikoisesti niukkaresurssisilla *MIDP*-laitteilla *XML*-pohjaisten viestien jäsennys voi aiheuttaa merkittävän suorituskyvyn alennuksen. Bansalin ja Daltonin (2002) tutkimuksessa jäsennyksen osuudeksi palvelupyynnön suoritusajasta saatiin 20% niukkaresurssisilla päätelaitteilla.

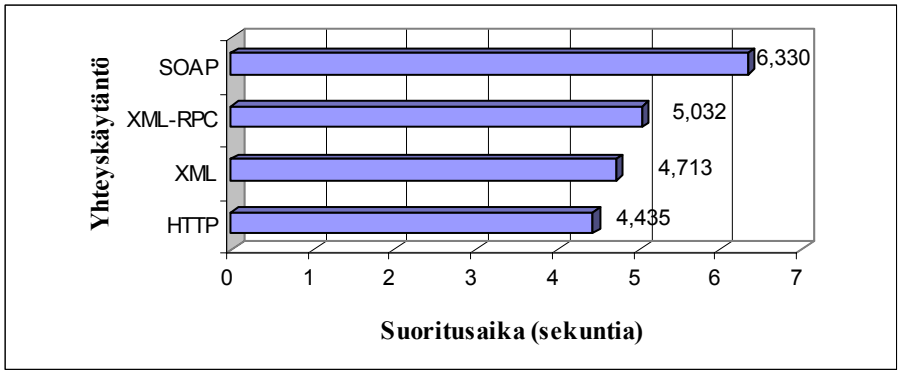
Taulukossa 5.4 esitetään keskiarvojen lisäksi myös kokeiden keskihajonnat. Suhteellinen keskihajonta eli vaihtelukerroin, suhteuttamalla keskihajonta keskiarvoon, vaihtelee *Nokia 6610i* matkapuhelimessa välillä 5,4% - 7,0%. Pienin suhteellinen keskihajonta on *SOAP*-yhteyskäytännöllä 5,4% ja suurin suhteellinen keskihajonta on *XML-RPC* -yhteyskäytännöllä 7,0%. *Motorola Accompli 00* matkapuhelimella suhteellinen keskihajonta vaihtelee välillä 4,0% - 4,6%. Pienin suhteellinen keskihajonta on *HTTP*-yhteyskäytännöllä 4,0% ja suurin suhteellinen keskihajonta on *SOAP*-yhteyskäytännöllä 4,6%.



Kuva 5.28: Suorituskyky yhteyskäytännöittäin: Emulaattori *WTK*.

Kuvassa 5.28 esitetään palvelupyynnön suoritusajan mittaustulokset *WTK*-kehitysympäristön emulaattorissa työasemaympäristössä, jossa on käytössä kiinteä laajakaistainen verkkoyhteys. Lisäksi työasema tarjoaa mittausympäristön, josta puuttuvat mobii-

lilaitteen ominaispiirteet kuten rajoitettu määrä muistia ja heikkotehoinen prosessori. Kuten kuvasta 5.28 havaitaan, *HTTP*-pohjaisella yhteyskäytännöllä on paras palvelupyynnön suoritusaika. Yleisesti kevytrakenteisena pidetyllä *XML-RPC* -yhteyskäytännöllä on lähes kaksinkertainen suoritusaika. *XML*- ja *XML-RPC* -yhteyskäytännöt osoittautuivat työasemaympäristössä likimain yhtä tehokkaiksi tiedonsiirtomenetelmiksi siirrettäessä pieni hyötykuorma asiakassovelluksen ja palvelinsovelluksen välillä. Kuva 5.28 paljastaa selkeästi *SOAP*-yhteyskäytännön tehottomuuden verrattaessa sitä muihin *XML*-pohjaisiin tiedonsiirtomenetelmiin ja *HTTP*-pohjaiseen yhteyskäytäntöön. *SOAP*-pohjaisella yhteyskäytännöllä on yli nelinkertainen suoritusaika verrattuna *HTTP*-pohjaiseen yhteyskäytäntöön ja yli kaksinkertainen suoritusaika verrattuna *XML-RPC*-pohjaiseen yhteyskäytäntöön.



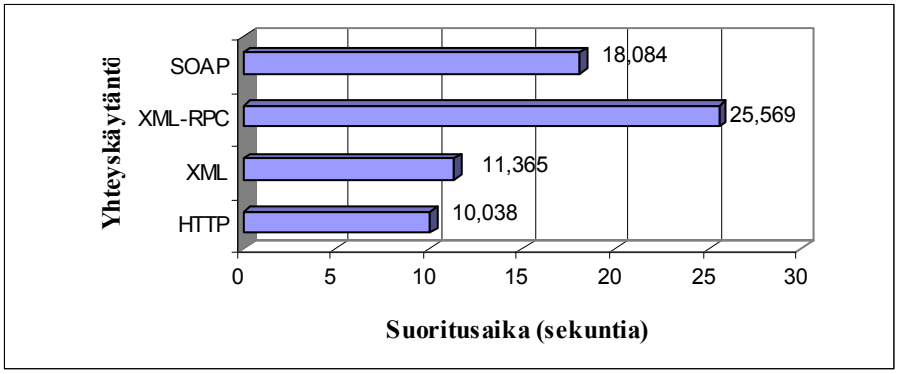
Kuva 5.29: Suorituskyky yhteyskäytännöittäin: Nokia 6610i.

Kuvassa 5.29 esitetään palvelupyynnön suoritusajan mittaustulokset *Nokia 6610i* matkapuhelimella. Kuten kuvasta havaitaan, *HTTP*-pohjaisella yhteyskäytännöllä on paras palvelupyynnön suoritusaika. *XML*-pohjaiselle yhteyskäytännöllä suoritusaika on noin 7% ja *XML-RPC* -pohjaiselle noin 13% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä. Eniten lisäkuormaa sisältävällä *SOAP*-pohjaiselle yhteyskäytännöllä palvelupyynnön suoritusaika on noin 43% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä ja 26% suurempi kuin *XML-RPC* -pohjaisella yhteyskäytännöllä.

Kuvassa 5.30 esitetään palvelupyynnön suoritusajan mittaustulokset *Motorola Accompli 008* matkapuhelimella. Kuten kuvasta havaitaan, *HTTP*-pohjaisella yhteyskäytännöllä on paras palvelupyynnön suoritusaika. *XML*-pohjaiselle yhteyskäytännöllä suoritusaika on noin 13% ja *XML-RPC* -pohjaiselle noin 155% suurempi kuin *HTTP*-pohjaisella

yhteyskäytännöllä. Eniten lisäkuormaa sisältävällä *SOAP*-pohjaiselle yhteyskäytännöllä palvelupyynnön suoritusaika on noin 80% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä.

Muista alustoista poiketen *Motorola Accompli 008* matkapuhelimella palvelupyynnön tiedonsiirron toteutus *XML-RPC* -yhteyskäytännön avulla vaati suoritusaikaa enemmän kuin toteutettaessa tiedonsiirto *SOAP*-yhteyskäytännön avulla. Yleisesti *XML-RPC* –yhteyskäytäntöä pidetään tehokkaampana yhteyskäytäntönä kuin *SOAP*-yhteyskäytäntöä (Yuan, 2003). Koska tässä tutkimuksessa ei mitattu suoritusaikoja palvelupyynnön eri vaiheille ei voida varmuudelle sanoa missä palvelupyynnön vaiheessa *Motorola Accompli 008* matkapuhelin tuhlasi suoritusaikaa. Todennäköistä on että, *XML-RPC* –viestin sarjallistuksen purku *Java*-tietotyypiksi asiakassovelluksella vaatii suoritusaikaa suhteellisen paljon heikkotehoisella prosessorilla varustetulla *Motorola Accompli 008* alustalla. Yi & al. (2002) päätyivät samansuuntaisiin tuloksiin vertaillessaan mobiilialustojen suorituskykyjä usean ominaisuuden suhteen. Tutkimukseen liittyi yhtenä osana *XML*-dokumentin jäsennyksen suorituskyvyn mittaus niukkaresurssisilla päätelaitteilla. Tulokset osoittivat, että jäsennyksen tehokkuuteen vaikutti voimakkaasti mittauksessa käytetyn laitteen prosessorin suorituskyky.



Kuva 5.30: Suorituskyky yhteyskäytännöittäin: Motorola Accompli 008.

5.2.3.2 Tiedonsiirron suorituskyvyn riippuvuus hyötykuorman koosta

Tässä alakohdassa esitetään tulokset mittauksista, joissa selvitettiin hyötykuorman koon vaikutusta palvelupyynnön suoritustehokkuuteen yhteyskäytännöittäin eri alustoilla. Mittauksissa käytettiin taulukossa 5.3 kuvattuja kuudella eri kokoisella hyötykuormalla varustettuja aineistoja. Jokaiselle aineistolle mittaus toistettiin kymmenen kertaa ja tulokset esitetään näiden toistojen keskiarvoina. Taulukossa 5.5 esitetään kuuden aineiston tulokset yhteyskäytännöittäin eri alustoilla. Aineiston 1 osalta tulokset ovat samat kuin taulukossa 5.4. Lisäksi tulokset esitetään graafisesti jokaiselle alustalle yhteyskäytännöittäin kuvissa 5.31 - 5.33.

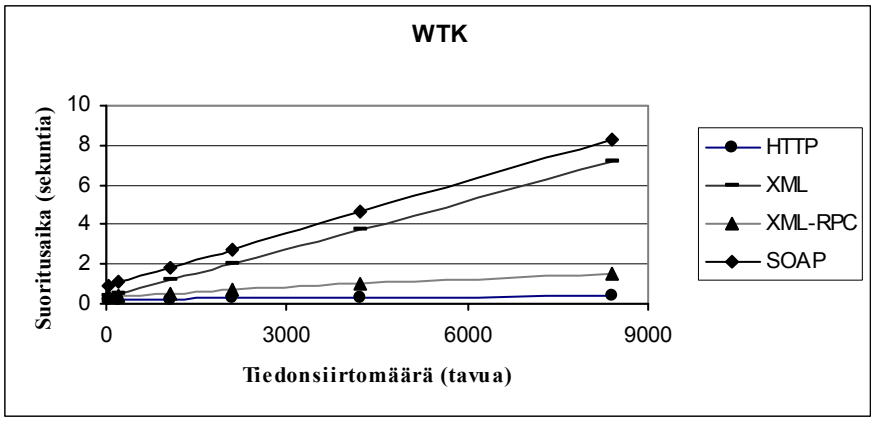
Taulukko 5.5: Suorituskykyanalyysin tulokset yhteyskäytännöittäin eri aineistoille.

Aineisto			Suoritusaika (sekuntia)					
			WTK		Nokia 6610i		Motorola Accompli	
n:o	Hyöty-kuorma (tavua)	Yhteyskäy-täntö	Keski-arvo	Keski-hajonta	Keski-arvo	Keski-hajonta	Keski-arvo	Keski-hajonta
1	25	HTTP	0.218	0.023	4.435	0.294	10.038	0.396
		XML	0.386	0.032	4.713	0.259	11.365	0.502
		XML-RPC	0.374	0.036	5.032	0.352	25.569	1.057
		SOAP	0.957	0.020	6.330	0.345	18.084	0.834
2	214	HTTP	0.214	0.017	4.507	0.265	10.243	0.476
		XML	0.545	0.029	4.994	0.300	12.661	0.442
		XML-RPC	0.405	0.016	4.988	0.197	26.094	2.246
		SOAP	1.147	0.038	6.082	0.205	18.884	0.349
3	1054	HTTP	0.236	0.024	4.575	0.216	11.583	0.463
		XML	1.214	0.036	6.786	0.337	19.413	0.689
		XML-RPC	0.528	0.026	6.291	0.431	25.867	0.808
		SOAP	1.867	0.033	8.140	0.441	25.680	0.630
4	2104	HTTP	0.253	0.019	5.828	0.479	13.117	0.588
		XML	2.065	0.034	8.130	0.190	26.765	0.501
		XML-RPC	0.665	0.029	6.790	0.529	25.966	0.555
		SOAP	2.772	0.048	10.496	0.775	32.042	0.409
5	4204	HTTP	0.310	0.021	7.921	0.656	17.651	1.647
		XML	3.732	0.024	12.601	0.457	42.054	1.123
		XML-RPC	0.960	0.027	9.542	0.470	41.835	0.318
		SOAP	4.607	0.085	13.981	0.716	49.045	1.155
6	8404	HTTP	0.425	0.045	11.427	0.421	23.452	1.781
		XML	7.179	0.055	19.902	0.835	73.165	0.908
		XML-RPC	1.554	0.029	13.727	0.589	68.905	1.506
		SOAP	8.253	0.053	21.489	0.515	81.208	2.082

Taulukossa 5.5 esitetään keskiarvojen lisäksi myös kokeiden keskihajonnat, joiden perusteella voidaan laskea suhteellinen keskihajonta eli vaihtelukerroin, joka on riippuma-

ton siirrettävän kuorman koosta. Kehitysympäristön emulaattorissa suhteellinen keskihajonta vaihtelee *HTTP*-yhteyskäytännölle välillä 7,5% - 11,0% ja *XML*-pohjaisille yhteyskäytännöille suhteellinen keskihajonta vaihtelee välillä 0,8% - 8,4%. *Nokia 6610i* matkapuhelimella suhteellinen keskihajonta vaihtelee *HTTP*-yhteyskäytännölle välillä 4,0% - 8,5% ja *XML*-pohjaisille yhteyskäytännöille suhteellinen keskihajonta vaihtelee välillä 2,2% - 7,2%. *Motorola Accompli 00* matkapuhelimella suhteellinen keskihajonta vaihtelee välillä 0,8% - 9,3% riippumatta yhteyskäytännöstä ja siirrettävän kuorman koosta.

Kuvassa 5.31 esitetään palvelupyynnön suoritusajan mittaustulokset *WTK*-kehitysympäristön emulaattorissa työasemaympäristössä hyötykuorman funktiona yhteyskäytännöittäin. Kuten kuvasta havaitaan, *HTTP*-pohjaisella yhteyskäytännöllä on paras palvelupyynnön suoritusaika riippumatta hyötykuorman määrästä. Kaikilla tarkastelluilla yhteyskäytännöillä palvelupyynnön suoritusaika kasvaa likimain lineaarisesti hyötykuorman funktiona. Siirrettävän hyötykuorman kasvaessa 25 tavusta 8404 tavuun kasvaa palvelupyynnön suoritusaika *HTTP*-pohjaisessa yhteyskäytännössä noin kaksinkertaiseksi, *XML*-pohjaisessa yhteyskäytännössä suoritusaika kasvaa noin 19 kertaiseksi, *XML-RPC* -pohjaisessa yhteyskäytännössä noin nelinkertaiseksi ja *SOAP*-pohjaisessa yhteyskäytännössä noin kahdeksankertaiseksi.

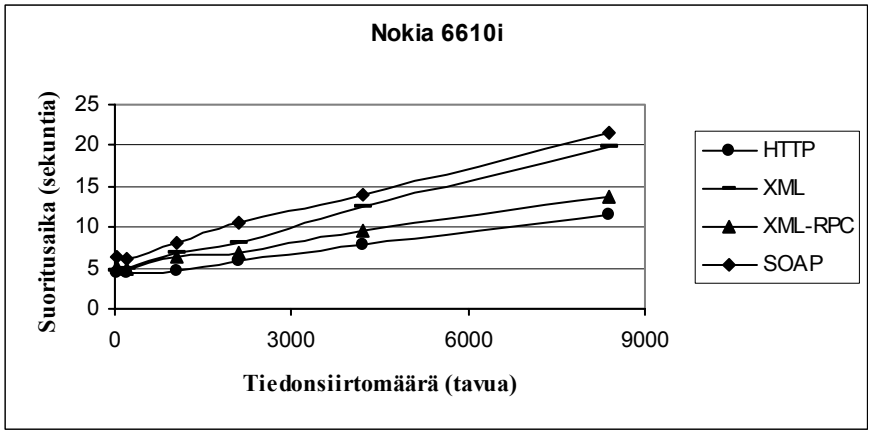


Kuva 5.31: Suorituskyky yhteyskäytännöittäin: Emulaattori WTK.

XML-RPC -yhteyskäytäntö osoittautui *XML*-pohjaisista palvelupyynnön yhteyskäytännöistä tehokkaimmaksi. *HTTP*- ja *XML-RPC* -pohjaiset yhteyskäytännöt ovat selvästi tehokkaampia kuin *XML*- ja *SOAP*-yhteyskäytännöt suuremmilla hyötykuorman määril-

lä. Esimerkiksi siirrettävän hyötykuorman määrän ollessa 8404 tavua palvelupyynnön toteutus *XML-RPC*-yhteyskäytännön avulla on yli viisi kertaa tehokkaampaa kuin toteuttaa se *SOAP*-yhteyskäytännön avulla.

Kuvassa 5.32 esitetään palvelupyynnön suoritusajan mittaustulokset *Nokia 6610i* matkapuhelimella hyötykuorman funktiona yhteyskäytännöittäin. Samoin kuin työasemaympäristössä myös todellisella matkapuhelimella, tässä tapauksessa *Nokia 6610i* alustalla, *HTTP*-pohjaisella yhteyskäytännöllä on paras palvelupyynnön suoritusaika riippumatta hyötykuorman määrästä. Toisaalta kuva 5.32 paljastaa, että *XML-RPC* – yhteyskäytäntö on varsin tehokas tiedonsiirtomenetelmä myös kapealla kaistanleveydellä varustetulla yhteydellä ja niukkaresurssisella alustalla suurillakin hyötykuormilla. Siirrettävän hyötykuorman ollessa 8404 tavua *XML-RPC* -pohjaisen yhteyskäytännön palvelupyynnön suoritusaika on vain 20% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä varustetulla palvelupyynnöllä. Vastaavalla hyötykuormalla *SOAP*-pohjaisella yhteyskäytännöllä suoritusaika on noin 88% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä ja *SOAP*-yhteyskäytännöllä noin 57% suurempi kuin *XML-RPC* -pohjaisella yhteyskäytännöllä

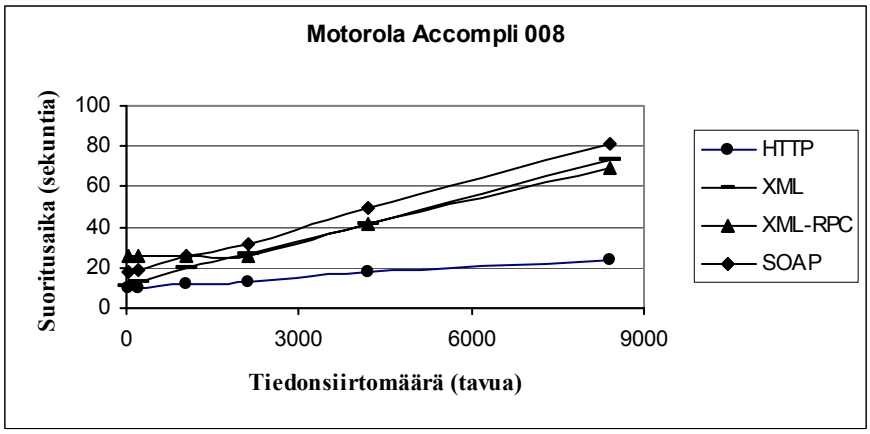


Kuva 5.32: Suorituskyky yhteyskäytännöittäin: Nokia 6610i.

Kuvan 5.32 mukaisesti kaikilla tarkastelluilla yhteyskäytännöillä palvelupyynnön suoritusaika kasvaa likimain lineaarisesti hyötykuorman funktiona käytettäessä mittausalustana *Nokia 6610i* matkapuhelinta. Siirrettävän hyötykuorman kasvaessa 25 tavusta 8404 tavuun kasvaa palvelupyynnön suoritusaika *HTTP*-pohjaisessa yhteyskäytännössä yli kaksinkertaiseksi, *XML*-pohjaisessa yhteyskäytännössä suoritusaika kasvaa yli nelinker-

taiseksi, *XML-RPC* -pohjaisessa yhteyskäytännössä yli kaksinkertaiseksi ja *SOAP*-pohjaisessa yhteyskäytännössä lähes nelinkertaiseksi.

Kuvassa 5.33 esitetään palvelupyynnön suoritusajan mittaustulokset *Motorola Accompli 008* matkapuhelimella hyötykuorman funktiona yhteyskäytännöittäin. Kuvan mukaisesti kaikilla tarkastelluilla yhteyskäytännöillä palvelupyynnön suoritusaika kasvaa likimain lineaarisesti hyötykuorman funktiona. Poikkeuksen muodostaa *XML-RPC* -pohjainen yhteyskäytäntö, jolla siirrettävän hyötykuorman ollessa alle 2000 tavua suoritusajan ei voida katsoa riippuvan lineaarisesti hyötykuorman määrästä. Tutkimuksessa ei selvitetty mistä tämä poikkeama johtuu. Siirrettävän hyötykuorman kasvaessa 25 tavusta 8404 tavuun kasvaa palvelupyynnön suoritusaika *HTTP*-pohjaisessa yhteyskäytännössä yli kaksinkertaiseksi, *XML*-pohjaisessa yhteyskäytännössä suoritusaika kasvaa yli kuusinkertaiseksi, *XML-RPC*-pohjaisessa yhteyskäytännössä lähes kolminkertaiseksi ja *SOAP*-pohjaisessa yhteyskäytännössä yli nelinkertaiseksi. Siirrettävän hyötykuorman ollessa 8404 tavua *XML*:ään pohjautuvilla yhteyskäytännöillä palvelupyynnön suoritusaika on noin 200% suurempi kuin *HTTP*-pohjaisella yhteyskäytännöllä.



Kuva 5.33: Suorituskyky yhteyskäytännöittäin: Motorola Accompli 008.

5.2.3.3 *RMS*-tietuevaraston suorituskyky

Tässä alakohdassa esitetään tulokset, joissa selvitetään *RMS*-tietuevaraston suorituskyky tallennettaessa palvelimelta vastaanotetut merkkijonot päätelaitteen *RMS*-tietuevarastoon. Jokaiselle kuvan 5.27 ja taulukon 5.3 esittämälle kuudelle aineistolle mittaus tois-

tettiin 40 kertaa ja tulokset esitetään näiden toistojen keskiarvoina. Taulukossa 5.6 esitetään kuudella aineistolla tallennuksen suoritusaika *RMS*-tietuevarastoon päätelaitteittain. Lisäksi tulokset esitetään graafisesti kuvassa 5.34.

Kuten taulukosta 5.6 ja kuvasta 5.34 havaitaan, paikallisen *RMS*-tietuevaraston suorituskyky tietojen tallennuksen osalta on suuresti riippuvainen tallennettavien tavujen määrästä ja käytetystä päätelaitteesta. Kaikilla tutkimuksessa käytetyillä alustoilla tietojen tallennuksen suoritusaika kasvaa likimain lineaarisesti tallennusmäärän funktiona. Kehitysympäristön emulaattorissa, jossa muistin määrän vähyys ja prosessorin suorituskyky eivät ole yhtä rajoittavia tekijöitä kuin mobiililaitteilla, tallennuksen suorituskyky on paras kaikilla tallennusmäärillä. Tallennettaessa *RMS*-tietuevarastoon pieni tavumäärä, 29 tavua, suoritusaika emulaattorilla on 7.5 millisekuntia. Tallennettavien tavujen määrän kasvaessa 8408 tavuun kasvaa suoritusaika lähes viisinkertaiseksi.

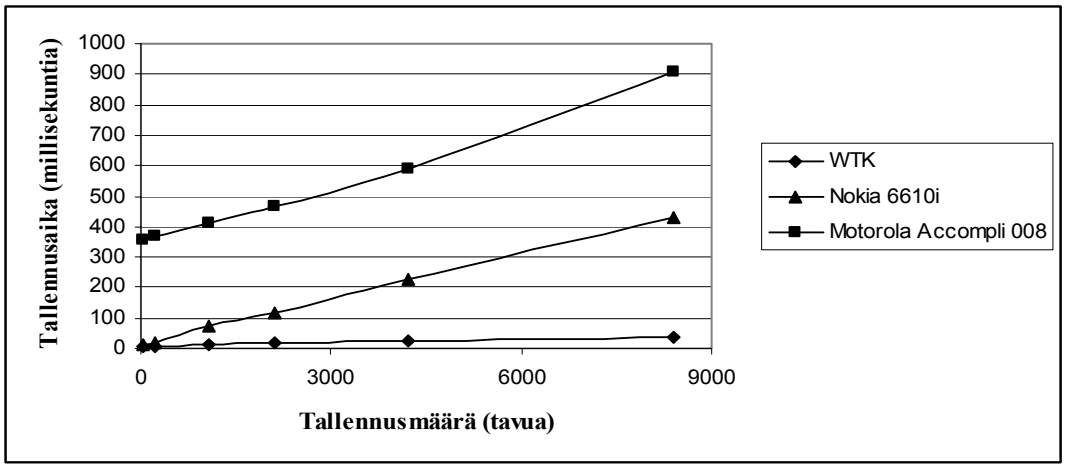
Taulukko 5.6: Tallennusaika *RMS*-tietuevarastoon alustoittain.

Tallennus määrä (tavua)	WTK		Nokia 6610i		Motorola Accompli 008	
	Keskiarvo (ms)	Hajonta (ms)	Keskiarvo (ms)	Hajonta (ms)	Keskiarvo (ms)	Hajonta (ms)
29	7,5	4,3	10,2	1,5	3572,1	462,7
218	7,8	4,7	20,1	2,4	3676,4	439,8
1058	11,5	3,5	73,2	4,8	4115,7	480,8
2108	15,8	4,9	118,8	10,7	4634,4	538,2
4208	21,5	3,6	228,8	18,7	5898,2	932,3
8408	37,3	5,9	430,4	30,8	9050,8	1286,9

Taulukosta 5.6 nähdään myös, että suoritusajan keskihajonta kasvaa tallennettavien tavujen määrän kasvaessa. Suhteellinen keskihajonta eli vaihtelukerroin, suhteuttamalla keskihajonta keskiarvoon, pienenee tallennettavien tavujen määrän kasvaessa. Kehitysympäristön emulaattorissa suhteellinen keskihajonta on noin 57% tallennusmäärän ollessa 29 tavua ja noin 16% tallennusmäärän ollessa 8408 tavua. *Nokia 6610i* matkapuhelimella suhteellinen keskihajonta on noin 15% tallennusmäärälle 29 tavua ja noin 7%

tallennusmäärälle 8408 tavua. *Motorola Accompli 008* matkapuhelimella suhteellinen keskihajonta vaihtelee välillä 13% - 16% riippumatta tallennettavien tavujen määrästä.

Todellisella matkapuhelimella *RMS*-tietuevaraston suorituskykyä rajoittaa mobiililaitteille ominainen muistinmäärän vähyys ja suhteellisen heikkotehoinen prosessori. Tässä tutkimuksessa käytetyllä *Nokia 6610i* matkapuhelimella tallennettaessa pieni tavumäärä *RMS*-tietuevarastoon suoritusaika on 10,2 millisekuntia, joka on vain 36% suurempi kuin tallennettaessa tiedot emulaattorissa. Tallennettavan tavumäärän kasvaessa resurssien niukkuus mobiilipäätelaitteilla alkaa rajoittaa *RMS*-tietuevaraston suorituskykyä. Tallennettavien tavujen määrän kasvaessa 8408 tavuun kasvaa suoritusaika *Nokia 6610i* matkapuhelimella yli 40 kertaiseksi, jolloin emulaattorilla on yli 10 kertaa parempi suoritusaika. *RMS*-tietuevaraston suorituskyky *Motorola Accompli 008* matkapuhelimella on selvästi heikompi kuin muilla alustoilla. Yi & al. (2002) saivat saman suuntaisia tuloksia paikallisen *RMS*-tietuevaraston suorituskyvylle.



Kuva 5.34: Tallennuksen suorituskyky alustoittain.[3]

[3] Matkapuhelimen Motorola Accompli 008 suoritusaika on jaettu luvulla 10.

5.3 Ohjelmointipanos

Tässä kohdassa pyritään osoittamaan sovelluskehittäjän kannalta olennaisimmat yhteyskäytäntöjen väliset eroavaisuudet. Yhteyskäytäntöjen välisien eroavaisuuksia tarkastelu pohjautuu kohtaan 5.1, jossa kuvattiin esimerkkisovellusten toteutus yhteyskäytännöittäin. Taulukossa 5.6 esitetään esimerkkisovelluksessa tarkasteltujen yhteyskäytäntöjen toteuttamiseen käytettyjen sovellusten osien toteuttamiseen tarvittavien koodirivien lukumäärä. Tietyn sovelluksen osan kohdalla koodirivien lukumäärän arvolla nolla osoitetaan, ettei ohjelmoijan tarvitse toteuttaa kyseistä sovelluksen osaa. Tällöin yhteyskäytäntö itse huolehtii kyseisen toiminnon toteuttamisesta. Vastaavasti koodirivien lukumäärän paikalla ollessa viiva osoitetaan, ettei kyseisen yhteyskäytännön toteutus vaadi kyseistä sovelluksen osan toteuttamista. Asiakassovelluksen vaatimaa työpanosta tarkastellaan alakohdassa 5.3.1 ja palvelinsovelluksen vaatimaa työpanosta alakohdassa 5.3.2 taulukon 5.6 jaottelun mukaisesti.

Taulukko 5.6: Ohjelmoinnin tehokkuus koodiriveinä yhteyskäytännöittäin.

Sovellusten osat	**Yhteyskäytäntö**			
Asiakassovellus	HTTP	XML	XML-RPC	SOAP
Ohjaus	79	79	79	79
Käyttöliittymä	36	36	36	36
Tietuevarasto	34	34	34	34
Verkkoyhteyssäie	40	40	40	40
Verkkoyhteyspyyntö	55	66	29	32
XML-jäsennys	-	38	0	0
Yhteensä	244	293	218	221
Palvelinsovellus				
Servlet	19	-	19	0
JSP-tekniikka	-	40	-	-
Web-palvelu	-	-	10	7
XML-jäsennys	-	33	0	0
Kuvaustiedosto	-	-	-	16
Tietovarasto	22	22	22	22
Tunnelointi	53	53	53	53
Yhteensä	94	148	104	98

5.3.1 Asiakassovellus

Asiakassovelluksen *Ohjaus*-toiminnon toteuttaa kuvan 5.2 asiakassovelluksen luokkakaavion mukaisesti *Ohjaus*-luokka. *Ohjaus*-toiminnolla hallitaan asiakassovelluksen elinkaarimallin mukaiset toiminnot. Lisäksi toiminto huolehtii asiakassovelluksen näytönhallinnasta. *Ohjaus*-toiminto on samanlainen kaikille yhteyskäytännöille, joten se on riippumaton käytetystä yhteyskäytännöstä. *Ohjaus*-toiminnon toteutus edellyttää ohjelmoijalta *MIDP*-pohjaisten sovellusten elinkaarimallin ymmärtämistä, jota on käsitelty kohdassa 3.1. Toiminnon toteutukseen tarvitaan esimerkkisovelluksessa 79 koodiriviä.

Käyttöliittymä-toiminnon toteuttaa kuvan 5.2 asiakassovelluksen luokkakaavion mukaisesti *PaaNaytto*-luokka. Toiminnolla muodostetaan sovelluksen käyttöliittymä. *Käyttöliittymä*-toiminto on samanlainen kaikille yhteyskäytännöille, joten se on riippumaton käytetystä yhteyskäytännöstä. *Käyttöliittymä*-toiminnon toteutus edellyttää ohjelmoijalta *MIDP*-pohjaisten sovellusten korkeantason käyttöliittymäohjelmoinnin ymmärtämistä, jota ei ole erityisesti tarkasteltu tässä tutkielmassa. Toiminnon toteutukseen tarvitaan esimerkkisovelluksessa 36 koodiriviä.

Tietuevarasto-toiminnon toteuttaa kuvan 5.2 asiakassovelluksen luokkakaavion mukaisesti *RMSToimittaja*-luokka. Toiminto huolehtii yhteyksistä päätelaitteen paikalliseen *RMS*-tietuevarastoon. *Tietuevarasto*-toiminto on samanlainen kaikille yhteyskäytännöille, joten se on riippumaton käytetystä yhteyskäytännöstä. Toiminnon toteutus edellyttää ohjelmoijalta *MIDP*-pohjaisten sovellusten tietueille perustuvan *RMS*-tallennusmekanismin ymmärtämistä, jota on käsitelty kohdassa 3.2. *Tietuevarasto*-toiminnon toteutukseen tarvitaan esimerkkisovelluksessa 34 koodiriviä.

Verkkoyhteys-toiminnon toteuttaa kuvan 5.2 asiakassovelluksen luokkakaavion mukaisesti *HttpVastaanotto*-luokka. Verkkoyhteyden toteutuksessa käytetään erillistä *säiettä*, jonka avulla elävöitetään käyttöliittymää ilmoittamalla käyttäjälle käynnissä olevasta verkkoyhteystoiminnosta. Erillisen säikeen toteutus on samanlainen kaikille yhteyskäytännöille, joten se on riippumaton käytetystä yhteyskäytännöstä. Esimerkkisovelluksessa säikeen toteutus vaatii 40 koodiriviä. *Verkkoyhteys*-toiminnossa muodostetaan

palvelinsovellukselle palvelupyyntö ja käsitellään palvelinsovelluksen lähettämä vastaus. Verkkoyhteyspyyntö on toteutettu kuvan 5.2 *HttpVastaanotto*-luokan metodissa *suoritaHaku*. Verkkoyhteyspyynnön osalta toteutuksessa on eroja yhteyskäytännöittäin.

HTTP-pohjaista yhteyskäytäntöä käyttävä esimerkkisovellus on perinteinen asiakas-palvelin -ohjelmisto, jossa viestinvälitys asiakassovelluksen ja palvelinsovelluksen välillä on toteutettu *HTTP*-yhteyskäytännön mukaisesti binaarimuodossa. Palvelinpuolen *Web*-komponenttina käytetään *servlettiä*. Asiakaspuolen toteutuksessa verkkoyhteyden ohjelmointi on helppoa ja vaatii ainoastaan *MIDP*-ohjelmoitiympäristön tuntemusta. Kuvassa 5.7 kuvataan miten asiakassovelluksessa muodostetaan palvelupyyntö ja yhteys palvelimelle sijoitettuun *web*-komponenttiin ja kuvassa 5.9 esitetään palvelinsovelluksen lähettämän vastauksen käsittely asiakassovelluksessa. Palvelupyynnön toteutus käytettäessä *HTTP*-yhteyskäytäntöä vaatii esimerkkisovelluksessa 55 koodiriviä.

XML-yhteyskäytäntöä käyttävässä esimerkkisovelluksessa viestinvälitys asiakassovelluksen ja palvelinsovelluksen välillä on toteutettu *HTTP*-yhteyskäytännön mukaisesti binaarimuodossa. Viestin esitysmuotona käytetään *XML*-dokumenttia kuvien 5.11 ja 5.14 mukaisesti. *XML*-tekniikan käyttö viestin välitykseen edellyttää, että ohjelmoijalla on tuntemus *XML*-kielen perusteista. Palvelinsovellus lähettää vastauksen *XML*-muodossa, joten asiakaspuolella täytyy olla *XML*-jäsennin. Esimerkkisovelluksessa käytetään pieniresurssisille päätelaitteille tarkoitettua *Enhydran kXML*-jäsennintä. *XML*-dokumenttien jäsennys joudutaan toteuttamaan ohjelmallisesti, joten se vaatii ohjelmoijalta *XML*-dokumenttien jäsennyksen osaamista. Palvelupyynnön toteutukseen esimerkkisovelluksessa tarvitaan 66 + 38 = 104 koodiriviä. Tämä on lähes kaksinkertainen koodirivien lukumäärä kuin toteutettaessa sama toiminnallisuus *HTTP*-yhteyskäytännön avulla. Suurempi koodirivien määrä johtuu *XML*-muotoisen palvelupyynnön muodostamisesta ja vastausviestin jäsennyksestä.

XML-RPC -yhteyskäytäntöä käyttävässä esimerkkisovelluksessa tehdään etäproseduurikutsuja palvelimelle. *XML-RPC* on hajautettujen sovellusten toteutustekniikka, jossa käytetään *HTTP*-yhteyskäytäntöä ja *XML*-dokumentteja komponenttien väliseen kommunikointiin. *XML-RPC* -tekniikka on ohjelmointikielestä ja alustasta riippumaton

ja yksinkertaisuuden ansiosta erittäin helppo käyttää. *XML-RPC* -tekniikan käyttö edellyttää, että asiakassovelluksella on *XML-RPC* -kirjasto. Esimerkkisovelluksessa käytetään pieniresurssisille päätelaitteille tarkoitettua *Enhydran kXML-RPC* -toteutusta. Asiakassovelluksessa etäproseduurikutsut sarjallistetaan *XML-RPC* -toteutuksen avulla kuljetusta varten *XML*-muotoon. Samoin palvelimen palauttamasta *XML*-muotoisesta vastausviestistä muodostetaan *Java*-tietotyyppi purkamalla sarjallistus *XML-RPC* -toteutuksen avulla. *XML-RPC* -tekniikan käyttö ei ohjelmoijalta periaatteessa edellytä *XML*-kielen syvällistä tuntemusta. Samoin *XML*-dokumenttia ei tarvitse ohjelmallisesti jäsentää, koska *XML-RPC* -toteutus huolehtii jäsennyksestä ja *XML*-dokumentin muodostamisesta. Kuvassa 5.16 esitetään *XML-RPC* -palvelupyynnön toteutus asiakassovelluksessa. Kuten kuvasta havaitaan *XML-RPC* -tekniikan käyttö on erittäin helppoa. *XML-RPC* -palvelupyynnön toteutukseen esimerkkisovelluksessa tarvitaan 29 koodiriviä. Toteutettaessa vastaava palvelupyyntö *HTTP*-yhteyskäytäntöä käyttäen tarvitaan lähes kaksinkertainen koodirivien määrä ja *XML*-yhteyskäytännöllä lähes nelinkertainen koodirivien määrä.

SOAP-yhteyskäytäntöä käyttävässä esimerkkisovelluksessa tehdään myös etäproseduurikutsuja palvelimelle. *SOAP* on yksinkertainen *XML*-pohjainen proseduurien etäkutsut mahdollistava tiedonsiirtomenetelmä. Tekstisisältöisen rakenteensa ansiosta *SOAP* on hyvin riippumaton käyttöympäristöstä ja helppokäyttöinen. *SOAP*-tekniikan käyttö vaatii, että asiakaspuolella on käytössä *SOAP*-kirjasto, joka osaa sarjallistaa etäproseduurikutsut kuljetusta varten *XML*-muotoon ja *XML*-muotoisista paluuviesteistä muodostaa *Java*-tietotyypin. Esimerkkisovelluksessa käytetään pieniresurssisille päätelaitteille tarkoitettua *Enhydran kSOAP*-toteutusta. Kuvassa 5.21 kuvataan palvelupyynnön toteutus asiakassovelluksessa. Kuten kuvasta nähdään palvelupyyntö muodostetaan *kSOAP*-toteutuksen tarjoamilla metodeilla, eikä ohjelmoijan tarvitse tuntea taustalla olevaa *XML*-kieltä. Kuljetusta varten *kSOAP*-toteutus sarjallistaa etäproseduurikutsusta kuvassa 5.22 esitetyn *XML*-muotoisen dokumentin. Vastaavasti asiakassovellus vastaanottaa palvelimelta kuvassa 5.25 kuvatun vastausviestin ja muodostaa siitä *Java*-tietotyypin purkamalla sarjallistuksen *kSOAP*-toteutuksen avulla. Asiakaspuolella palvelupyynnön toteutukseen esimerkkisovelluksessa tarvitaan 32 koodiriviä, joka on vain 10% enemmän kuin toteutettaessa sama toiminto *XML-RPC* -tekniikalla. Toteutettaessa

vastaava palvelupyyntö *HTTP*-yhteyskäytäntöä käyttäen tarvitaan lähes kaksikertainen koodirivien määrä ja *XML*-yhteyskäytännöllä lähes nelinkertainen koodirivien määrä.

5.3.2 Palvelinsovellus

Seuraavaksi tarkastellaan palvelinpuolen toteutuksien erityispiirteitä yhteyskäytännöittäin. Tässä tutkielmassa sovelluspalvelimena käytetään *Tomcat*-palvelinta, jonka hakemistorakenne on esitetty kuvassa 5.4. Alakohdassa 5.1.3 kuvatussa *HTTP*-pohjaista yhteyskäytäntöä käyttävässä esimerkkisovelluksessa käytetään palvelinpuolen *web*-komponenttina servlettiä. *Servlet*-määrityksen (Sun Microsystems, 2001) mukaan *servletin* tehtävänä *web*-sovelluksessa on palvella sovelluspalvelimelle tulevia kutsuja ja vastata niihin luomalla dynaamista sisältöä. *Servlettien* käyttö edellyttää ohjelmoijalta tuntemusta sovelluspalvelimen hakemistorakenteesta ja *web*-sovelluksen *web.xml*-tiedostosta, jonka avulla *servletit* otetaan käyttöön. Esimerkkisovelluksen *servletin* toteutukseen tarvitaan 19 koodiriviä.

XML-yhteyskäytäntöä käyttävässä esimerkkisovelluksessa asiakassovelluksen palvelupyyntö käsitellään *JSP*-tekniikan ja *JavaBeans*-komponentin avulla. *JPS*-komponentti vastaanottaa asiakassovelluksen lähettämän *XML*-muotoisen palvelupyynnön ja muodostaa *XML*-pohjaisen vastausviestin. Pyyntöviestin rakenne on kuvattu kuvassa 5.11 ja vastausviestin rakenne kuvassa 5.14. *JSP*-komponentin toteutukseen tarvitaan esimerkkisovelluksessa 40 koodiriviä. *XML*-muotoisen pyyntöviestin jäsennys täytyy toteuttaa ohjelmallisesti, joten ohjelmoijalta edellytetään tietämystä *XML*-kielen perusteista ja *XML*-dokumenttien jäsennyksestä. Esimerkkisovelluksessa *XML*-muotoisen pyyntöviestin jäsennys toteutetaan käyttämällä hyväksi *JavaBeans*-komponenttia, jonka toteutus on esitetty alakohdassa 5.1.4. *XML*-dokumentin jäsennykseen tarvitaan esimerkkisovelluksessa 33 koodiriviä. *XML*-yhteyskäytäntöä käyttävässä esimerkkisovelluksessa tarvitaan lähes nelinkertainen koodirivien määrä *HTTP*-yhteyskäytäntöön verrattuna.

XML-RPC -yhteyskäytäntöä käyttävän esimerkkisovelluksen palvelinpuolen toteutus on kuvattu tarkemmin alakohdassa 5.1.5. *XML-RPC* -tekniikan käyttöä varten tarvitaan

palvelinpuolelle *XML-RPC* -kirjasto, jonka sijoitus *Tomcat*-sovelluspalvelimelle on esitetty kuvassa 5.4. Ohjelmoijan tarvitsee toteuttaa *web*-palvelu, jota *XML-RPC* -maailmassa kutsutaan käsittelijäksi. Käsittelijä on *Java*-luokka, jonka metodeja asiakassovellus etäkutsuu *RPC*:n yli. Käsittelijäluokan toteutus on kuvattu kuvassa 5.18. Lisäksi ohjelmoijan täytyy toteuttaa kuvassa 5.19 esitetty *servletti*, joka vastaanottaa asiakassovelluksen lähettämät palvelupyynnöt ja välittää palvelupyynnöt edelleen käsittelijälle. *XML-RPC* -toteutus huolehtii *XML*-muotoisten pyyntö- ja vastausviestin sarjallistamisesta ja sarjallistamisen purkamisesta, joten tekniikan käyttö ei edellytä ohjelmoijalta *XML*-jäsennyksen tuntemusta. Pyyntöviestin rakenne on kuvattu kuvassa 5.17 ja vastausviestin rakenne kuvassa 5.20. Esimerkkisovelluksessa *web*-palvelun ja *servletin* toteutukseen tarvittiin yhteensä 29 koodiriviä. Vastaavan toiminnallisuuden toteuttaminen *XML*-yhteyskäytännön avulla vaatii ohjelmoijalta yli kaksinkertaisen määrän koodirivejä.

SOAP-yhteyskäytäntöä käyttävän esimerkkisovelluksen palvelinpuolen toteutus on kuvattu tarkemmin alakohdassa 5.1.6. *SOAP*-tekniikan käyttöä varten tarvitaan palvelinpuolelle *SOAP*-kirjasto, jonka sijoitus esimerkkisovelluksessa käytetylle *Tomcat*-sovelluspalvelimelle on esitetty kuvassa 5.4. Palvelinpuolen *SOAP*-toteutus vaatiikin toimiakseen *servlet*-palvelimen. Ohjelmoijan täytyy toteuttaa palvelimelle *web*-palvelu, johon toteutettuja metodeja asiakassovellus etäkutsuu. *SOAP*-ympäristössä *web*-palvelut ovat pohjimmiltaan *Java*-luokkia kuvan 5.23 mukaisesti. *SOAP*-toteutus sisältää sisäänrakennetun servletin *RPCRouterServlet*, joka välittää asiakassovelluksen palvelupyynnöt *SOAP*-toteutukselle, jossa *XML*-pohjainen *SOAP*-palvelupyyntö jäsennetään ja suoritetaan pyydetty etämetodikutsu. *Web*-palvelun lisäksi ohjelmoijan täytyy kirjoittaa palvelun kuvaava *XML*-pohjainen kuvaustiedosto kuvan 5.24 mukaisesti. Kuvaustiedoston avulla *web*-palvelu aktivoidaan käyttäen hyväksi *SOAP*-toteutuksen sisältämiä työkaluja. Esimerkkisovelluksessa *web*-palvelun ja kuvaustiedoston toteutukseen tarvittiin yhteensä 23 koodiriviä. Vastaavan toiminnallisuuden toteuttaminen *XML*-yhteyskäytännön avulla vaatii ohjelmoijalta yli kolminkertaisen määrän koodirivejä.

Tietovarastona esimerkkisovelluksissa palvelinpuolella ei käytetä todellista tietokantaa, vaan tietojen tallennusta varten toteutettiin *Java*-luokka kuvan 5.6 mukaisesti. Tieto-

varaston toteuttavan *Java*-luokan ohjelmoimiseen tarvitaan esimerkkisovelluksessa 22 koodiriviä. Jos tietovarasto toteutettaisiin tietokantana, muuttuisi koodirivimäärä vastaavasti.

Varsinaisena palvelimena toimiva *Tomcat*-sovelluspalvelin on palomuurilla suojattu. Tästä syystä asiakassovelluksen palvelupyynnöt lähetetään varsinaisella palvelimelle välittäjäpalvelimena toimivan *Apache*-palvelimen kautta. Välittäjäpalvelimen toteutusta on kuvattu alakohdassa 5.1.2. Välittäjä toteutettiin tunnelointina, jonka toteutukseen esimerkkisovelluksessa tarvitaan 53 koodiriviä.

6 YHTEENVETO

Kehitettäessä teknologiaa matkapuhelinten kautta käytettäviä lisäarvopalveluja varten on täytynyt ottaa huomioon pienten mobiilipäätelaitteiden ja matkapuhelinverkkojen rajoitukset. Päätelaitteiden osalta on pitänyt huomioida pienitehoiset prosessorit, muistin vähyys, rajoitettu virran kulutus, pienet näyttöresoluutiot ja pöytätietokoneista poikkeavat syöttölaitteet, esimerkiksi puhelimen näppäimistö ja osoitinlaitteiden puuttuminen. Edellä kuvatut langattomien tietoverkkojen ja päätelaitteiden luonteenpiirteet ovat luoneet tarpeen kehittää erityisesti näihin olosuhteisiin soveltuvaa teknologiaa lisäarvopalvelujen tuottamiseen.

Tutkielmassa haettiin vastauksia *XML*-pohjaisten yhteyskäytäntöjen käyttömahdollisuuksiin ohjelmistojen hajauttamisessa *J2ME*-ympäristössä. *XML*-pohjaisten yhteyskäytäntöjen suorituskykyä verrattiin *HTTP*-pohjaiseen yhteyskäytäntöön, jossa tiedonsiirto asiakassovelluksen ja palvelinsovelluksen välillä tapahtuu binaarimuodossa. Binaarimuodossa tapahtuva tiedonsiirto mahdollistaa tiiviin ja optimoidun tiedon esitysmuodon. Suorituskykymittaus suoritettiin *Nokia 6610i* ja *Motorolo Accompli 008* matkapuhelimilla, joissa on käytössä kapeakaistainen langaton *GPRS*-verkkoyhteys. Lisäksi mittaus suoritettiin kehitysympäristön emulaattorissa, jolloin on käytössä kiinteä laajakaistainen verkkoyhteys.

XML-pohjaisten yhteyskäytäntöjen ongelmana pidetään niiden tarvitsemaa suhteellisen suurta lisäkuorman määrää, joka voi vaikuttaa suorituskykyä alentavasti. Toinen *XML*-pohjaisten viestien suorituskykyyn vaikuttava tekijä on *XML*-viestien jäsennys, joka joudutaan tekemään sekä asiakaspuolella että palvelinpuolella. Varsinkin niukkaresurssisilla mobiililaitteilla *XML*-pohjaisten viestien jäsennys voi johtaa suorituskyvyn alentumiseen. Lisäkuorman vaikutuksen selville saamiseksi toteutettiin taulukon 5.2 kuvaaman, pienen vakiohyötykuorman sisältävän, perusaineiston avulla yksinkertainen pyynnön ja vastauksen sisältävä kysely asiakas- ja palvelinsovelluksen välillä. Taulukossa 6.1 esitetään tuloksista yhteenveto laskemalla suhteet taulukkojen 5.2 ja 5.4 perusteella siirrettävälle kuormalle ja suorituskyvylle suhteutettuna *HTTP*-yhteyskäytäntöön. Siirrettävä kuorma sisältää hyötykuorman sekä lisäkuorman. Kun *HTTP*-yhteyskäytännölle

skaalataan arvoksi yksi, lisäkuorman suhteellinen osuus kasvaa moninkertaiseksi siirryttäessä *HTTP*-yhteyskäytännöstä *XML*-pohjaisiin yhteyskäytäntöihin. Suurin lisäkuorma aiheutuu *SOAP*-yhteyskäytännölle.

Taulukon 6.1 suorituskyvyn tulokset lisäkuorman vaikutuksesta perustuvat *Nokia 6610i* matkapuhelimella saatuihin tuloksiin. Taulukossa suoritusajat on suhteutettu siten, että *HTTP*-yhteyskäytännön suoritusaika on skaalattu arvoon yksi ja *XML*-pohjaisten yhteyskäytäntöjen suoritusajat esittävät kuinka monta kertaa enemmän ne vievät suoritusaikaa. Taulukosta 6.1 nähdään palvelupyynnön suoritusajan riippuvuus yhteyskäytännön tarvitseman lisäkuorman määrästä. Huonoin palvelupyynnön suoritusaika saatiin *SOAP*-yhteyskäytännölle.

Taulukko 6.1: Yhteenveto yhteyskäytäntöjen lisäkuorman vaikutuksesta.

Ominaisuus	Yhteyskäytäntö			
	HTTP	XML	XML-RPC	SOAP
Hyötykuorma + lisäkuorma	1	6,7	15,0	35,1
Suorituskyky (Nokia 6610 i)	1	1,07	1,14	1,43

Mittaustulokset *Motorola Accompli 008* matkapuhelimelle on esitetty taulukossa 5.4 ja kuvassa 5.30. *Motorola Accompli 008* alustalla saatiin huomattavasti suuremmat palvelupyynnön suoritusajat kuin *Nokia 6610i* alustalla. Tutkimuksessa käytetyt matkapuhelimet ovat tulleet markkinoille eri aikoina, joten laitteiden suorituskyvyssä näkyy matkapuhelinalan voimakas kehitys. Kehitysympäristön emulaattorissa saadut mittaustulokset on esitetty taulukossa 5.4 ja kuvassa 5.28. Emulaattorissa on käytettävissä kiinteä laajakaistainen verkkoyhteys ja asiakassovelluksen käytössä on riittävästi muistia sekä tehokas prosessori.

Toisena empiirisen tutkimuksen osana selvitettiin hyötykuorman koon vaikutusta suorituskykyyn yhteyskäytännöittäin. Taulukossa 6.2 esitetään tuloksista yhteenveto laskemalla suhteet taulukon 5.5 perusteella. Tulokset perustuvat *Nokia 6610i* matkapuhelimella suoritettuihin mittauksiin. Taulukossa suoritusajat on suhteutettu siten, että *HTTP*-yhteyskäytännön suoritusaika on skaalattu ykköseksi ja *XML*-pohjaisten yhteys-

käytäntöjen suoritusajat esittävät kuinka monta kertaa enemmän ne vievät suoritusaikaa. Taulukosta 6.2 nähdään yhteenveto palvelupyynnön suoritusajan riippuvuudesta yhteyskäytännöstä ja hyötykuorman koosta. Hyötykuorman määrän kasvu vaikuttaa voimakkaimmin *XML*-yhteyskäytännön ja *SOAP*-yhteyskäytännön suorituskykyyn. *HTTP*-yhteyskäytännön ja *XML-RPC* -yhteyskäytännön välinen suhde on lähes riippumaton hyötykuorman määrästä.

Taulukko 6.2: Yhteenveto suorituskyvyn riippuvuudesta hyötykuorman koosta.

Hyötykuorma (tavua)	Yhteyskäytäntö			
	HTTP	XML	XML-RPC	SOAP
25	1	1,07	1,13	1,43
214	1	1,11	1,11	1,35
1054	1	1,48	1,37	1,78
2104	1	1,39	1,17	1,80
4204	1	1,59	1,20	1,76
8404	1	1,74	1,20	1,88

Empiirisen tutkimuksen kolmantena osana selvitettiin paikallisen *RMS*-tietuevaraston suorituskykyä tiedon tallennuksen osalta. Taulukossa 6.3 esitetään tuloksista yhteenveto vertaamalla mobiilin päätelaitteen tuloksia kehitysympäristön emulaattorin vastaaviin tuloksiin erisuurilla tiedon tallennusmäärillä. Yhteenveto on muodostettu taulukon 5.6 perusteella. Taulukossa suoritusajat on suhteutettu siten, että emulaattorilla saatu suoritusaika on skaalattu ykköseksi ja *Nokia 6610i* matkapuhelimella saatu suoritusaika esittää kuinka monta kertaa enemmän se vaati suoritusaikaa. Taulukosta 6.3 nähdään, että mobiililaitteiden vähäinen muistin määrä ja suhteellisen heikkotehoinen prosessori heikentävät *RMS*-tietuevaraston suorituskykyä tallennettaessa suuria tietomääriä paikalliseen *RMS*-tietuevarastoon.

Taulukko 6.3: Yhteenveto *RMS*-tietuevaraston suorituskyvystä.

Alusta	Tallennusmäärä (tavua)					
	29	218	1058	2108	4208	8404
WTK	1	1	1	1	1	1
Nokia 6610i	1,4	2,6	6,4	7,5	10,6	11,5

Tutkielman tavoitteena oli kuvan 1.2 mukaisesti suorituskyvyn lisäksi ohjelmointipanoksen selvittäminen yhteyskäytäntöjen osalta. Kohdassa 5.3 yhteyskäytäntöjä tarkasteltiin sovelluskehittäjän kannalta ja pyrittiin osoittamaan niiden väliset eroavaisuudet. Taulukossa 6.4 esitetään tuloksista yhteenveto taulukon 5.6 pohjalta, jossa esitettiin esimerkkisovelluksessa tarkasteltujen yhteyskäytäntöjen toteuttamiseen tarvittavien koodirivien lukumäärä. Taulukossa 6.4 toteutukseen tarvittavien koodirivien lukumäärä on suhteutettu siten, että *HTTP*-yhteyskäytännön toteuttamiseen tarvittavien koodirivien lukumäärä on skaalattu ykköseksi ja *XML*-pohjaisten yhteyskäytäntöjen toteuttamiseen tarvittavien koodirivien suhteelliset arvot esittävät kuinka paljon enemmän tai vähemmän ne tarvitsevat koodirivejä.

Taulukko 6.4: Yhteenveto ohjelmoinnin tehokkuudesta.

Ominaisuus	Yhteyskäytäntö			
	HTTP	XML	XML-RPC	SOAP
Asiakassovellus	1	1,20	0,89	0,91
Palvelinsovellus	1	1,57	1,11	1,04
Yhteensä	1	1,30	0,95	0,94

Taulukosta 6.4 havaitaan, että esimerkkitapauksessa *XML-RPC*- ja *SOAP*-yhteyskäytäntöjä käytettäessä asiakassovelluksen toteutukseen tarvitaan vähemmän koodirivejä kuin *HTTP*-yhteyskäytäntöön pohjautuvassa toteutuksessa. *XML-RPC*- ja *SOAP*-yhteyskäytäntöihin pohjautuvissa asiakassovelluksissa ohjelmoija käyttää valmiita kirjastoja muodostaessaan palvelupyyntöjä palvelimelle. Nämä kirjastot huolehtivat palvelupyyntöjen sarjallistamisesta ja sarjallistamisen purkamisesta. *XML*-yhteyskäytäntöön pohjautuvassa asiakassovelluksessa täytyy ohjelmallisesti huolehtia *XML*-viestien muodostamisesta ja jäsennyksestä, mikä selittää *XML*-yhteyskäytäntöön pohjautuvan asiakassovelluksen suuremman koodirivien lukumäärän.

Esimerkkitapauksessa palvelinpuolen toteutuksista *XML*-yhteyskäytäntöön pohjautuvassa toteutuksessa tarvitaan selvästi enemmän koodirivejä kuin toisiin tarkasteltuihin yhteyskäytäntöihin pohjautuvissa toteutuksissa. *XML*-yhteyskäytäntöön pohjautuvassa palvelinpuolen toteutuksessa täytyy ohjelmallisesti huolehtia *XML*-viestien muodostamisesta ja jäsennyksestä, mikä selittää *XML*-yhteyskäytäntöön pohjautuvan palvelinpuo-

len toteutuksen suuremman koodirivien lukumäärän. *XML-RPC* -yhteyskäytäntöön pohjautuvan toteutuksen suuremman koodimäärän selittää palvelupyyntöjen kuuntelun toteutus. *XML-RPC* -yhteyskäytäntöön pohjautuva toteutus sisältää ohjelmallisesti toteutetun servletin, joka kuuntelee asiakassovelluksen *XML-RPC* -palvelupyyntöjä. *SOAP*-yhteyskäytäntöön pohjautuvassa toteutuksessa *SOAP*-toteutus sisältää sisäänrakennetun servletin, joka kuuntelee *SOAP*-palvelupyyntöjä.

Saatujen tulosten perusteella suorituskyky on siis riippuvainen siirrettävästä kuormasta ja teknologioista. *XML*-pohjaisista yhteyskäytännöistä *XML-RPC* on suorituskykyisin ja ohjelmoijan kannalta yksinkertaisin toteuttaa. Ottamalla huomioon tässä tutkielmassa havaitun teknologian ikävaikutuksen suorituskykyyn, voidaan lähivuosina odottaa monipuolisemman *SOAP*-yhteyskäytännön tulevan varteenotettavammaksi vaihtoehdoksi myös mobiiliympäristössä.

VIITELUETTELO

Allman, M. (2003) An Evaluation of XML-RPC. *ACM SIGMETRICS Performance Evaluation Review*, 30(4), 2-11.

Apache (2003a) *WebServices - SOAP*. WWW-sivusto, The Apache Software Foundation, http://ws.apache.org/soap/ (5.3.2004).

Apache (2003b) *XML-RPC - Apache XML-RPC*. WWW-sivusto, The Apache Software Foundation, http://ws.apache.org/xmlrpc/ (25.2.2004).

Apache (2004a) *The Apache XML Project*. WWW-sivusto, The Apache Software Foundation, http://xml.apache.org/ (25.2.2004).

Apache (2004b) *The Jakarta Site - Apache Jakarta Tomcat*. WWW-sivusto, The Apache Software Foundation, http://jakarta.apache.org/tomcat/index.html (25.2.2004).

Bansal, V., Dalton, A. (2002) *A Performance Analysis of Web Services on Wireless PDAs*. CPS 214 Computer Networks, Duke University Computer Science, USA (Saatavana myös: http://www.cs.duke.edu/~vkb/advnw/project/index.html, 5.1.2005).

Hanslo, W., MacGregor, K. (2004) The Efficiency of XML as an Intermediate Data Representation for Wireless Middleware Communication. *Proceedings of SAICSIT* (toim. Maisden, G. Kotze, P. Adesina-Ojo, A.), 279-283.

Chu, H., You,C., Teng, C. (2004) Challenges: Wireless Web Services. *The Tenth International Conference on Parallel and Distributed Systems*, IEEE Computer Society, USA (Saatavana myös: http://ieeexplore.ieee.org/, 5.1.2005).

Enhydra (2003a) *The home of kXML at Enhydra.org*. WWW-sivusto, http://kxml.enhydra.org/ (4.6.2004).

Enhydra (2003b) *The home of kSOAP at Enhydra.org*. WWW-sivusto, http://ksoap.enhydra.org/ (4.6.2004).

Enhydra (2003c) *The home of kXML-RPC at Enhydra.org*. WWW-sivusto, http://-kxmlrpc.enhydra.org/ (4.6.2004).

Giguere, E. (2002) *Understanding J2ME Application Models*. Internet WWW-sivusto: http://wireless.java.sun.com/midp/articles/models/ (22.1.2005).

Harold, E.R. (2002) *Processing XML with Java*. Addison-Wesley, USA.

Holopainen, H. (2002) *Midlet-pohjaisen tietokantasovelluksen rakentaminen*. Tietojenkäsittelytieteen kandidaatintutkielma, Joensuun yliopisto, Joensuu.

Holopainen, H. (2003) *Midlet-pohjainen tietokantasovellus*. Tietojenkäsittelytieteen erikoistyö, Joensuun yliopisto, Joensuu.

Knudsen, J. (2002) *Understanding MIDlet Memory*. Internet WWW-sivusto: http://wireless.java.sun.com/midp/ttips/memory/ (10.12.2004).

Laurent, S., Johnston, J., Dumbill, E. (2001) *Programming Web Services with XML-RPC*. O'Reilly & Associates, USA.

Mahmoud, Q.H. (2000a) *MIDP Database Programming using RMS: a Persistent Storage for Midlet.* Internet WWW-sivusto: http://wireless.java.sun.com/midp/articles/persist/ (20.6.2004).

Mahmoud, Q.H. (2001a) *Learning Wireless Java*. Internet WWW-sivusto: http://www.oreilly.com/catalog/wirelessjava/chapter/ch05.html (15.12.2004).

Mahmoud, Q.H. (2001b) *MIDP Inter-Communication with CGI and Servlets*. Internet WWW-sivusto: http://wireless.java.sun.com/midp/articles/servlets/ (15.12.2004).

Mahmoud, Q.H. (2002) *Wireless Java Security*. Internet WWW-sivusto: http://wireless.java.sun.com/midp/articles/security/ (28.1.2005).

McLaughlin, B. (2001) *Java & XML*. Talentum Media Oy, Jyväskylä.

Muchow, J. (2002) *The Basic of J2ME*. Internet WWW-sivusto, http://wireless.java.sun.com/midp/chapters/muchowcore/ch1.pdf (5.2.2005).

Riggs, R., Taivalsaari, A., VandenBrink, M. (2001) *Midp User Interface Libraries*. Internet WWW-sivusto, http://wireless.java.sun.com/midp/chapters/pwdevices/ch09. pdf (13.12.2004).

Schiller, J. (2001) *Mobiilitietoliikenne*. Edita Oyj, Helsinki.

Sun Microsystems (2000a) *Connected, Limited Device Configuration Specification*. WWW-sivusto, Sun Microsystems, http://jcp.org/aboutJava/communityprocess/final/jsr-030/index.html (10.12.2004).

Sun Microsystems (2000b) *Mobile Information Device Profile Specification (JSR-37)*. JCP Specification, WWW-sivusto, Sun Microsystems, http://jcp.org/aboutJava/communityprocess/final/jsr030/index.html (10.12.2004).

Sun Microsystems (2001) *Java™ Servlet Specification Version 2.3*. WWW-sivusto, Sun Microsystems, http://www.jcp.org/aboutJava/commuityprocess/final/jsr053 (4.2.2004).

Sun Microsystems (2003) *Supporting Disconnected Operation in Wireless Enterprise Applications*, WWW-sivusto, Sun Microsystems, http://java.sun.com/blueprints/earlyaccess/wireless/disconnected/disconnected.pdf (4.2.2004).

Sun Microsystems (2005) *Java 2 Platform, Micro Edition (J2ME)*. WWW-sivusto, Sun Microsystems, http://java.sun.com/j2me/index.html (4.2.2004).

Topley, K. (2002) *J2ME in a Nutshell*. Internet WWW-sivusto, http://www.oreilly.com-/catalog/j2meanut/chapter/ch03.html (10.12.2004).

Tremblett, P. (2002) *Instant Wireless Java with J2ME*. McGray-Hill, USA.

W3C (1999a) *Hypertext Transfer Protocol - HTTP/1.1*. WWW-sivusto, World Wide Web Consortium, http:/www.w3.org/Protocols/rfc2616/rfc2616.html (10.12.2004).

W3C (1999b) *Namespaces in XML*. WWW-sivusto, World Wide Web Consortium, http:/www.w3.org/TR/REC-xml-names/ (12.12.2004).

W3C (2003a) *SOAP Version 1.2 Part 0: Primer*. WWW-sivusto, World Wide Web Consortium, http://www.w3.org/TR/2003/REC-soap12-part0-20030624/ (1.12.2004).

W3C (2003b) *SOAP Version 1.2 Part 1: Messaging Framework*. WWW-sivusto, World Wide Web Consortium, http://www.w3.org/TR/2003/REC-soap12-part1-2003-0624/ (1.12.2004).

W3C (2004) *Extensible Markup Language 1.1*. WWW-sivusto, World Wide Web Consortium, http://www.w3.org/REC-xml11-20040204 (10.12.2004).

White, J.P., Hemphill, D.A. (2002) *Java 2 Micro Edition*. Manning Publication Co, Greenwich.

Winer, D. (1999) *XML-RPC Specification*. WWW-sivusto, http://www.xmlrpc.com/-spec (10.12.2004).

Yi, W., Reddy, C.J., Ang, G. (2002) *J2ME Devices: Real-world performance*. WWW-sivusto, http://www.javaworld.com/javaworld/jw-10-2002/jw-1025-j2mebenchmark_p-.html (22.1.2005).

Yuan, M.J. (2002) *Access Web services from wireless devices*. WWW-sivusto, http://w-ww.javaworld.com/javaworld/jw-08-2002/jw-0823-wireless_p.html (22.1.2005).

Yuan, M.J. (2003) *Enterprise J2ME: Developing Mobile Java Applications*. Prentice Hall, USA.

Liite 1: Asiakassovelluksen lähdekoodi

Luokka: Ohjaus.java

Kuvaus: Sovelluksen pääluokka.

Toiminta: Ohjaa sovelluksen suoritusta ja luokan avulla toteutetaan näytönhallinta.

Historia: 7.4.2005 Heikki Holopainen

```
import javax.microedition.midlet.*;
import javax.microedition.lcdui.*;

public class Ohjaus extends MIDlet
  {
  private RMSToimittaja rmsToimittaja = null;
  private PaaNaytto paaNaytto = null;
  private HttpVastaanotto httpVastaanotto;
  private Form odotaLomake = new Form("Odota...");

  public Ohjaus() {
    rmsToimittaja = new RMSToimittaja();
    httpVastaanotto = new HttpVastaanotto(rmsToimittaja);
    paaNaytto = new PaaNaytto(this, rmsToimittaja, httpVastaanotto);
    }

  public void startApp() {
    Displayable current = Display.getDisplay(this).getCurrent();
    // Asetetaan näyttö.
    if (current == null) {
      Alert aloitusNaytto = new Alert("Gradu", "MIDlet-pohjainen\nTietokanta-"
                                    +"\nsovellus", null, null);
      aloitusNaytto.setTimeout(500);
      Display.getDisplay(this).setCurrent(aloitusNaytto, paaNaytto);
      }
    else {
      Display.getDisplay(this).setCurrent(current);
      }
    }

  public void pauseApp() {
    }

  public void destroyApp(boolean unconditional) {
    // Lopetetaan verkkoyhteysluokan säikeen suoritus.
    httpVastaanotto.lopeta();
    // Suljetaan paikallinen tietuevarasto.
    try {
      rmsToimittaja.suljeToimittajaRMS();
      }
    catch (Exception e) {
      // Ei käsitellä.
      }
    }

  // PaaNaytto-luokan takaisinkutsumetodit

  public void paaNayttoLopeta() {
    // Lopetetaan sovelluksen suoritus.
    destroyApp(false);
    notifyDestroyed();
    }

  public void paaNayttoVirhe(String viesti) {
    // Näytetään ilmoitus virheellisestä syötteestä.
    Alert a = new Alert("Viesti", viesti, null, null);
    a.setTimeout(Alert.FOREVER);
    Display.getDisplay(this).setCurrent(a, paaNaytto);
    }


  public void paaNayttoOdotaLomake(String viesti) {
    // Asetetaan aktiiviseksi näytöksi verkkoyhteyden ajaksi
```

```
  // odotusnäyttö.
  if (odotaLomake.size() > 0)
    odotaLomake.delete(0);
  odotaLomake.append(new StringItem(null, viesti));
  Display.getDisplay(this).setCurrent(odotaLomake);
  }

public void paaNayttoHttpVastaus(String viesti) {
  // Näytetään ilmoitus onnistuneesta verkkoyhteydestä.
  Alert a = new Alert("Viesti", viesti, null, null);
  a.setTimeout(Alert.FOREVER);
  Display.getDisplay(this).setCurrent(a, paaNaytto);
  }

public void paaNayttoHttpVirhe(String viesti) {
  // Näytetään ilmoitus epäonnistuneesta verkkoyhteydestä.
  Alert a = new Alert("Viesti", viesti, null, null);
  a.setTimeout(Alert.FOREVER);
  Display.getDisplay(this).setCurrent(a, paaNaytto);
  }
}
```

Luokka:	PaaNaytto.java
Kuvaus:	Muodostaa näytön, jonka avulla sovellus vastaanottaa käyttäjän syötteet.
Toiminta:	PaaNaytto-luokka on peritty Form-luokasta. Luokka muodostaa näytön, jossa on yksi tekstikenttä ja kaksi komentopainiketta.
Historia:	7.4.2005 Heikki Holopainen

```
import javax.microedition.lcdui.*;
import javax.microedition.rms.*;

public class PaaNaytto extends Form implements CommandListener, HttpKuuntelija
  {
  private Ohjaus ohjaus;
  private HttpVastaanotto httpVastaanotto;
  private RMSToimittaja rmsToimittaja;
  private TextField tfTunnus;
  private Command cmdPeruuta;
  private Command cmdHae;

  public PaaNaytto(Ohjaus ohjaus, RMSToimittaja rmsToimittaja,
    HttpVastaanotto httpVastaanotto)
    {
    super("Tietuehaku");
    // Viittaus Ohjaus-luokkaan takaisinkutsuja varten.
    this.ohjaus = ohjaus;
    // Viittaus RMSToimittaja-tyyppiseen olioon.
    this.rmsToimittaja = rmsToimittaja;
    // Viittaus HttpVastaanotto-tyyppiseen olioon.
    this.httpVastaanotto = httpVastaanotto;
    // Luodaan tekstikenttä tietojen syöttämistä varten.
    tfTunnus = new TextField("Numero: ", "", 5, TextField.ANY);
    // Luodaan komentopainikkeet.
    cmdPeruuta = new Command("Lopeta", Command.CANCEL, 1);
    cmdHae = new Command("Hae", Command.OK, 2);
    // Liitetään tekstikenttä ja komentopainikkeet lomakkeelle.
    append(tfTunnus);
    addCommand(cmdPeruuta);
    addCommand(cmdHae);
    // Asetetaan näyttö tapahtumien kuuntelijaksi.
    setCommandListener(this);
    }

  public void commandAction(Command c, Displayable d) {
    if (c == cmdPeruuta) {
      tfTunnus.setString("");
      ohjaus.paaNayttoLopeta();
      }
    else if (c == cmdHae) {
      if (tfTunnus.getString().trim().equals("")) {
        ohjaus.paaNayttoVirhe("Anna toimittajanumero");
        }
      else {
        String toimittajanro = tfTunnus.getString().trim();
        ohjaus.paaNayttoOdotaLomake("\n\nLataus...");
        httpVastaanotto.haeToimittaja(toimittajanro, this);
        }
      }
    }

  // HttpKuuntelija-rajapinnan toteuttavat metodit.

  public void httpVastaus(String vastaus) {
    ohjaus.paaNayttoHttpVastaus(vastaus);
    }

  public void httpVirhe(String viesti) {
    ohjaus.paaNayttoHttpVirhe(viesti);
    }

  }
```

Luokka:	RMSToimittaja.java
Kuvaus:	RMSToimittaja-luokka huolehtii yhteyksistä paikallisen *RMS*-tietuevaraston ja sovelluksen välillä.
Toiminta:	Luokan muodostimessa avataan yhteys paikalliseen *RMS*-tietuevarastoon ja luokkaan toteutettujen metodien avulla voidaan tietuevarastoon lisätä ja poistaa tietueita.
Historia:	7.4.2005 Heikki Holopainen

```
import javax.microedition.rms.*;
import java.io.*;

public class RMSToimittaja
  {
  private RecordStore recordStore = null;
  private String toimittajaRMS = "RMS-tietuevarasto";

  public RMSToimittaja() {
    try {
      // Avataan RMS-tietuevarasto.
      if (recordStore == null) {
        recordStore = RecordStore.openRecordStore(toimittajaRMS, true);
        }
      }
    catch (RecordStoreException rse) {
      }
    }

  // Lisätään toimittajatietue RMS-tietuevarastoon.
  public int lisaaTietue(Toimittaja toimittaja)
    throws IOException, RecordStoreNotOpenException,
           RecordStoreException, RecordStoreFullException
    {
    ByteArrayOutputStream stream = new ByteArrayOutputStream();
    DataOutputStream dataStream  = new DataOutputStream(stream);
    // Kirjoitetaan toimittaja-olion jäsenmuuttujan ByteArrayOutputStream-olioon.
    toimittaja.serialize(dataStream);
    dataStream.close();
    // Lisätään toimittajatietue RMS-tietuevarastoon.
    return recordStore.addRecord(stream.toByteArray(), 0, stream.size());
    }


  // Suljetaan ja poistetaan RMS-tietuevarasto.
  public void suljeToimittajaRMS()
    throws RecordStoreNotOpenException, RecordStoreException,
           NullPointerException
    {
    if (recordStore != null) {
      // Suljetaan RMS-tietuevarasto.
      recordStore.closeRecordStore();
      // Poistetaan RMS-tietuevarasto.
      RecordStore.deleteRecordStore(toimittajaRMS);
      recordStore = null;
      }
    }

  // Poistetaan tietue RMS-tietuevarastosta.
  public void poistaTietue(int tunnus)
    throws IOException, RecordStoreNotOpenException,
           InvalidRecordIDException, RecordStoreException
    {
    // Poistetaan toimittajatietue RMS-tietuevarastosta.
    recordStore.deleteRecord(tunnus);
    }

  }
```

Luokka:	HttpKuuntelija.java
Kuvaus:	Luokka määrittelee liittymän.
Toiminta:	Liittymä määrittelee kaksi takaisinkutsumetodia. Liittymän toteuttaa PaaNaytto-luokka ja liittymää käyttää HttpVastaanotto-luokka.
Historia:	7.4.2005 Heikki Holopainen

```
public interface HttpKuuntelija
  {
  public void httpVastaus(String vastaus);
  public void httpVirhe(String viesti);
  }
```

Luokka:	HttpVastaanotto.java
Kuvaus:	HttpVastaanotto on verkkoyhteysluokka, jolla hoidetaan toimittajatietojen haku palvelimelta paikalliseen RMS-tietuevarastoon.
Toiminta:	Luokka toteuttaa Runnable-rajapinnan ja se suoritetaan erillisessä säikeessä. Ohjelman käynnistyessä luokan muodostimessa käynnistetään säikeen suoritus. Säikeen suoritus keskeytetään run-metodissa wait-metodilla. Kutsuttaessa verkkoyhteyden käynnistävää metodia haeToimittaja, metodilla notify herätetään odotustilassa oleva säie. Tämän jälkeen run-metodissa kutsutaan metodia suoritaHaku, jossa suoritetaan toimittajatietueiden haku palvelimelta paikalliseen RMS-tietuevarastoon.
Historia:	7.4.2005 Heikki Holopainen

```
import javax.microedition.io.*;
import javax.microedition.rms.*;
import java.io.*;
import java.util.*;

public class HttpVastaanotto implements Runnable
  {
  private RMSToimittaja rmsToimittaja;
  private boolean lopetus = false;
  private Vector kuuntelijaVektori = new Vector();
  private String toimittajanro;
  private Vector params = new Vector();
  private String url = "http://ujocs.joensuu.fi/servlets/XMLTunneli";

  public HttpVastaanotto(RMSToimittaja rmsToimittaja){
    this.rmsToimittaja = rmsToimittaja;
    // Luodaan ja käynnistetään säie.
    Thread thread = new Thread(this);
    thread.start();
    }

  public synchronized void haeToimittaja(String toimittajanro,
    HttpKuuntelija kuuntelija)
    {
    this.toimittajanro = toimittajanro;
    // Sijoitetaan viittaus palvelua pyytävää olioon.
    kuuntelijaVektori.addElement(kuuntelija);
    // Herätetään run-metodin odotusalueella suoritusvuoroa odottava säie.
    notify();
    }

  public void run() {
    suoritus:
      while (!lopetus) {
        HttpKuuntelija kuuntelija;
        synchronized (this) {
          // Mennään odotusalueelle.
          while (kuuntelijaVektori.size() == 0) {
            try {
              wait();
```

```
            }
          catch (InterruptedException e) {
            }
          // Jos lopetus on tosi poistutaan molemmista silmukoista.
          if (lopetus)
            break suoritus;
          }
        kuuntelija = (HttpKuuntelija) (kuuntelijaVektori.elementAt(0));
        kuuntelijaVektori.removeElementAt(0);
        }
      suoritaHaku(kuuntelija);
      }
  }


public void suoritaHaku(HttpKuuntelija kuuntelija)
  {
  HttpConnection con = null;
  OutputStream os = null;
  InputStream in = null;
  String viesti = "";
  boolean virhe = false;
  String vastausViesti = "";
  String virheViesti = "";
  long alkuAika;
  long verkkoAika;
  long rmsAika;
  alkuAika = System.currentTimeMillis();
  try {
    // Luodaan yhteys palvelimelle.
    con = (HttpConnection) Connector.open(url);
    // Asetetaan tarvittavat otsakekentät.
    con.setRequestMethod(HttpConnection.POST);
    con.setRequestProperty("Content-Type", "application/x-www-form-urlencoded");
    con.setRequestProperty("User-Agent", "Profile/MIDP-1.0 Configuration/CLDC-1.0");
    con.setRequestProperty("Content-Language", "en-US");
    con.setRequestProperty("Accept", "application/octet-stream");
    con.setRequestProperty("Connection", "close");
    // Muodostetaan HTTP-pyynnön XML-dokumentti
    byte data[] = generoiXMLData();
    con.setRequestProperty("Content-Lengh", Integer.toString(data.length));
    // Avataan kirjoitusvirta ja lähetetään pyyntöviesti palvelimelle.
    os = con.openOutputStream();
    os.write(data);
    os.close();
    // Luetaan palvelimen lähettämä vastausviesti.
    int rc = con.getResponseCode();
    if (rc == HttpConnection.HTTP_OK) {
      in = con.openInputStream();
      // Välitetään XML-dokumentti DBXMLParser-luokalle.
      DBXMLParser p = new DBXMLParser(in);
      // Luetaan XML-vastausviestistä toimittajan tiedot
      String numero     = p.annaKentanArvo("toimittaja", "toimittajanro", 0);
      String nimi       = p.annaKentanArvo("toimittaja", "toimittajanimi", 0);
      String lahiosoite = p.annaKentanArvo("toimittaja", "lahiosoite", 0);
      Toimittaja toimittaja = new Toimittaja(numero, nimi, lahiosoite);
      verkkoAika = System.currentTimeMillis() - alkuAika;
      alkuAika = System.currentTimeMillis();
      // Tallennetaan toimittajatiedot RMS-tietuevarastoon.
      int tunnus = rmsToimittaja.lisaaTietue(toimittaja);
      rmsAika = System.currentTimeMillis() - alkuAika;
      vastausViesti = "Tietue " + Integer.toString(tunnus) + "\nSuoritusaika " +
                      "\n Haku " + verkkoAika + "\n Tallennus " + rmsAika;
      in.close();
      // Poistetaan edellä tallennettu toimittajatietue RMS-tietuevarastosta.
      rmsToimittaja.poistaTietue(tunnus);
      }
    else {
      // Virhe palvelimella
      virhe = true;
      virheViesti = "Palvelinvirhe";
      }
    } // try
  catch (Exception e) {
    virhe = true;
```

```
    virheViesti = "Virhe";
    }
  // Suljetaan yhteys palvelimelle.
  finally {
    try {
      if (os != null) os.close();
      if (con != null)  con.close();
      }
    catch(IOException ioe) {
      // Ei käsitellä
      }
    }
  // Paluu päänäyttöön.
  if (virhe) {
    kuuntelija.httpVirhe(virheViesti);
    }
  else {
    kuuntelija.httpVastaus(vastausViesti);
    }
  }

// Muostetaan HTTP-pyynnön XML-dokumentti.
public byte [] generoiXMLData() {
      StringBuffer sb = new StringBuffer();
  sb.append("<root>");
  sb.append("<toimittaja>");
  sb.append("<toimittajanro>");
  sb.append(toimittajanro);
  sb.append("</toimittajanro>");
  sb.append("</toimittaja>");
  sb.append("</root>");
  return sb.toString().getBytes();
  }

// Lopetetaan säie
public void lopeta() {
  lopetus = true;
  // Herätetään run-metodin odotusalueella suoritusvuoroa odottava säie.
  synchronized(this) {
    notify();
    }
  }
}
```

Luokka: DBXMLParser.java

Kuvaus: Luokka suorittaa XML-dokumentin jäsennyksen.

Toiminta: Luokan muodostimessa muodostetaan XML-dokumentista jäsennyspuu. Luokkaan toteutettu metodi annaKentanArvo palauttaa XML-dokumentista tietyn toimittajatietueen kentän arvon.

Historia: 7.4.2005 Heikki Holopainen

```
import java.io.*;
import org.kxml.*;
import org.kxml.parser.*;
import org.kxml.kdom.*;

public class DBXMLParser
  {
  private Document doc;

  public DBXMLParser(InputStream ins)
    throws Exception
    {
    // Muodostetaan XML-dokumentista jäsennyspuu.
    InputStreamReader reader = new InputStreamReader(ins);
    XmlParser parser = new XmlParser(reader);
    doc = new Document();
    doc.parse(parser);
    }


  public String annaKentanArvo(String tauluNimi, String kenttaNimi, int index)
    throws Exception
    {
    Element juuri = doc.getRootElement();
    Element rivi = null;
    Element kentta = null;
    int nykyinenIndex = 0;
    // Käydään läpi juurielementin lapsielementit
    for (int i = 0; i < juuri.getChildCount(); i++) {
      if (juuri.getType(i) == Xml.ELEMENT) {
        Element lapsi = (Element) juuri.getChild(i);
        if (tauluNimi.equals(lapsi.getName())) {
          if (nykyinenIndex == index) {
            rivi = (Element) juuri.getChild(i);
            break;
            }
          nykyinenIndex++;
          }
        }
      } //for
    //
    if (rivi == null) {
      return null;
      }
    // Käydään läpi toimittaja-elementin lapsi-elementit
    for (int i = 0; i < rivi.getChildCount(); i++) {
      if (rivi.getType(i) == Xml.ELEMENT) {
        Element kenttaElement = (Element) rivi.getChild(i);
        if (kenttaNimi.equals(kenttaElement.getName())) {
          kentta = kenttaElement;
          break;
          }
        }
      } //for
    if (kentta == null) {
      return null;
      }
    else {
      return kentta.getText(0);
      }
    }
  } //end
```

Luokka:	Toimittaja.java
Kuvaus:	Luokka pitää sisällään yhden toimittajan tiedot.
Toiminta:	Luokka sisältää get- ja set-metodit, joilla käsitellään luokan jäsenmuuttujia. Lisäksi luokka sisältää metodit serialize, jolla olio sarjallistaa itsensä lukuvirtaan ja deserialize, jolla olio purkaa sarjallistuksen.
Historia:	7.4.2005 Heikki Holopainen

```
import java.io.*;
public class Toimittaja
  {
  private String numero;
  private String nimi;
  private String osoite;

  public Toimittaja() {
    }

  public Toimittaja(String numero, String nimi, String osoite) {
    this.numero = numero;
    this.nimi = nimi;
    this.osoite = osoite;
    }

  public void setNumero(String numero) {
    this.numero = numero;
    }

  public void setNimi(String nimi) {
    this.nimi = nimi;
    }

  public void setOsoite(String osoite) {
    this.osoite = osoite;
    }

  public String getNumero() {
    return numero;
    }

  public String getNimi() {
    return nimi;
    }

  public String getOsoite() {
    return osoite;
    }

  // Toimittaja-olio sarjallistaa itsensä lukuvirtaan.
  public void serialize(DataOutputStream dataStream)
    throws IOException
    {
    dataStream.writeUTF(numero);
    dataStream.writeUTF(nimi);
    dataStream.writeUTF(osoite);
    }

  // Toimittaja-olio purkaa sarjallistuksen.
  public static Toimittaja deserialize(DataInputStream dataStream)
    throws IOException
    {
    Toimittaja toimittaja = new Toimittaja();
    toimittaja.numero = dataStream.readUTF();
    toimittaja.nimi = dataStream.readUTF();
    toimittaja.osoite = dataStream.readUTF();
    return toimittaja;
    }

  }
```

Liite 2: Palvelinsovelluksen lähdekoodi

Luokka:	Toimittajahaku.jsp
Kuvaus:	JSP-sovelma Toimittajahaku vastaanottaa asiakassovelluksen palvelupyynnön ja muodostaa vastausviestin.
Toiminta:	JSP-sovelma on tietovarastoon yhteydessä sovelluslogiikan muodostavan JavaBean-komponentin ToimittajaBean kautta.ToimittajaBean-komponentti palauttaa toimittajatiedot Toimittaja-tyyppisessä oliossa. Seuraavaksi JSP-sovelma muodostaa XML-dokumentin toimittajatietueesta ja lähettää sen asiakassovellukselle.
Historia:	7.4.2005 Heikki Holopainen

```
<%@ page import="JavaBeans.*" %>
<%@ page import="java.util.*" %>
<%@ page import="java.io.*" %>
<jsp:useBean id="parser" class="JavaBeans.ToimittajaBean" />

<%
  // Haetaan toimittajaolio.
  Toimittaja toimittaja = parser.getToimittaja(request.getReader());
  byte [] data = null;
  // Muodostetaan XML-dokumentti vastaukselle.
  ByteArrayOutputStream byteOut = new ByteArrayOutputStream();
  data = "<root>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "<toimittaja>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "<toimittajanro>".getBytes();
  byteOut.write(data, 0, data.length);
  data = toimittaja.getNumero().getBytes();
  byteOut.write(data, 0, data.length);
  data = "</toimittajanro>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "<toimittajanimi>".getBytes();
  byteOut.write(data, 0, data.length);
  data = toimittaja.getNimi().getBytes();
  byteOut.write(data, 0, data.length);
  data = "</toimittajanimi>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "<lahiosoite>".getBytes();
  byteOut.write(data, 0, data.length);
  data = toimittaja.getOsoite().getBytes();
  byteOut.write(data, 0, data.length);
  data = "</lahiosoite>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "</toimittaja>".getBytes();
  byteOut.write(data, 0, data.length);
  data = "</root>".getBytes();
  byteOut.write(data, 0, data.length);
  // Muunnetaan ByteArrayOutputStream-olio tavutaulukoksi.
  data = byteOut.toByteArray();
  // Lähetään response.
  response.setStatus(response.SC_OK);
  response.setContentLength(data.length);
  response.setContentType("application/octet-stream");
  OutputStream os = response.getOutputStream();
  os.write(data);
  os.close();
%>
```

Luokka: ToimittajaBean.java

Kuvaus: ToimittajaBean-luokka suorittaa XML-dokumentin jäsennyksen.

Toiminta: Luokan muodostimessa luodaan ilmentymä tietovarastoluokasta ToimittajaDB. Luokkaan toteutettu metodi getToimittaja palauttaa haetun toimittajan tiedot Toimittaja-tyyppisessä oliossa.

Historia: 7.4.2005 Heikki Holopainen

```
package JavaBeans;

import java.io.*;
import org.xml.sax.Attributes;
import java.io.IOException;
import java.util.*;
import org.xml.sax.InputSource;
import org.xml.sax.SAXException;
import org.xml.sax.XMLReader;
import org.xml.sax.helpers.DefaultHandler;
import org.apache.xerces.parsers.SAXParser;

public class ToimittajaBean extends DefaultHandler
  {
  ToimittajaDB db = null;
  Toimittaja toimittaja = null;
  boolean lue = false;

  public ToimittajaBean() {
    db = new ToimittajaDB();
    }

  public Toimittaja getToimittaja(Reader reader)
    throws SAXException, IOException
    {
    // Luodaan lukija-olio.
    XMLReader xmlReader = new SAXParser();
    //Rekisteröidään sisällönkäsittelijä lukija-olioon.
    xmlReader.setContentHandler(this);
    // Aloitetaan XML-dokumentin jäsennys.
    xmlReader.parse(new InputSource(reader));
    // Palautetaan toimittajatiedot.
    return toimittaja;
    }

  public void startElement(String uri, String localName, String rawName,
    Attributes attrs)
    // Jäsennin kutsuu metodia, kun kohdataan XML-dokumentin elementin alkutunniste.
    {
    if (rawName.equals("toimittajanro")) {
      lue = true;
      }
    }

  public void characters(char[] characters, int start, int length)
    {
    // Tekstimuotoinen data siirretään sovellukselle.
    if (lue) {
      String toimittajanro = (new String(characters, start, length)).trim();
      toimittaja = db.getToimittaja(toimittajanro);
      lue = false;
      }
    }

  public void endElement(String uri, String localName, String rawName)
    // Jäsennin kutsuu metodia, kun kohdataan XML-dokumentin elementin lopputunniste.
    {
    lue = false;
    }
  }
```

Luokka:	ToimittajaDB.java
Kuvaus:	ToimittajaDB-luokka muodostaa palvelinpuolen tietovaraston.
Toiminta:	Luokan muodostimessa luodaan kuusi erikokoista tutkimusaineistoa. Luokkaan toteutettu metodi getToimittaja palauttaa toimittajatiedot Toimittaja-tyyppisessä oliossa.
Historia:	7.4.2005 Heikki Holopainen

```
package JavaBeans;
import java.util.*;

public class ToimittajaDB
  {
  private Hashtable db;
  private String nimi = "Matti J Mainio";
  private String osoite = "Joensuu";

  public ToimittajaDB() {
    db = new Hashtable();
    lisaa(new Toimittaja("10", "Matti J Mainio", "Joensuu"));
    lisaa(new Toimittaja("11", kasvata(10,nimi), kasvata(10, osoite)));
    lisaa(new Toimittaja("12", kasvata(50,nimi), kasvata(50, osoite)));
    lisaa(new Toimittaja("13", kasvata(100,nimi), kasvata(100, osoite)));
    lisaa(new Toimittaja("14", kasvata(200,nimi), kasvata(200, osoite)));
    lisaa(new Toimittaja("15", kasvata(400,nimi), kasvata(400, osoite)));
    }

  private void lisaa(Toimittaja toimittaja) {
    db.put(toimittaja.getNumero(), toimittaja);
    }

  public Toimittaja getToimittaja(String numero) {
    return (Toimittaja) db.get(numero);
    }

  private String kasvata(int lkm, String param) {
    StringBuffer sb = new StringBuffer();
    for (int i = 0; i < lkm; i++) {
      sb.append(param);
      }
    return sb.toString();
    }
  }
```

Liite 3: Mitatut suoritusajat

Taulukko 1: *WTK*-emulaattori.

Aineisto	Suorituskykykokeiden tulokset (millisekuntia)									
	1	2	3	4	5	6	7	8	9	10
HTTP-yhteyskäytäntö										
1	231 10	210 10	241 10	200 10	191 10	221 10	270 10	211 0	200 10	210 10
2	240 10	200 10	210 10	241 10	220 10	230 0	200 10	200 10	200 0	201 0
3	271 10	230 10	220 10	230 10	250 10	200 10	271 10	231 10	210 10	250 10
4	240 20	281 20	250 20	270 20	251 20	230 10	260 11	230 20	240 20	280 20
5	341 20	301 20	290 20	340 20	300 20	281 20	330 20	301 20	321 20	300 20
6	380 50	431 40	391 40	401 40	431 40	391 40	461 40	531 40	431 30	400 31
XML-yhteyskäytäntö										
1	441 10	400 10	371 10	360 0	370 10	361 10	351 0	361 10	431 0	410 10
2	540 11	551 20	581 10	531 10	550 10	520 0	581 10	581 0	501 10	510 10
3	1272 20	1191 10	1252 10	1172 10	1232 20	1192 10	1242 10	1161 10	1222 10	1201 11
4	2033 20	2103 10	2083 20	2043 20	2103 10	2023 10	2050 10	2113 10	2032 20	2063 20
5	3745 20	3725 20	3735 30	3686 20	3746 20	3705 30	3755 20	3756 30	3755 20	3715 20
6	7100 41	7231 30	7221 30	7200 40	7130 20	7240 40	7210 40	7130 40	7220 30	7111 40
XML-RPC -yhteyskäytäntö										
1	440 10	350 10	431 10	341 0	370 0	361 10	391 10	351 0	360 10	340 10
2	391 10	400 10	421 0	420 10	400 10	411 10	400 10	380 10	391 10	431 0
3	541 10	541 10	501 20	531 10	521 10	541 10	581 10	501 10	491 10	531 10
4	641 20	671 10	661 10	641 10	691 10	681 10	620 20	701 10	641 20	701 10
5	952 20	921 30	971 20	1011 20	972 20	921 20	951 20	971 20	962 20	972 20
6	1562 30	1563 50	1552 30	1522 40	1583 40	1512 40	1613 40	1532 30	1562 30	1542 40
SOAP-yhteyskäytäntö										
1	991 10	981 0	961 10	922 10	951 10	971 0	962 0	952 10	940 10	942 10
2	1142 0	1122 10	1252 10	1131 10	1141 10	1131 0	1132 10	1142 10	1132 0	1141 10
3	1913 10	1833 10	1823 10	1852 10	1863 10	1833 20	1913 10	1862 10	1893 20	1883 20
4	2834 10	2834 20	2724 20	2834 20	2794 20	2714 20	2754 20	2734 10	2754 10	2744 20
5	4536 20	4747 20	4487 20	4687 30	4677 20	4617 20	4527 20	4657 20	4536 30	4597 20
6	8242 30	8212 40	8212 40	8171 40	8282 40	8372 40	8252 30	8252 40	8262 40	8272 40

Taulukko 2: Matkapuhelin *Nokia 6610i*.

Aineisto	Suorituskykykokeiden tulokset (millisekuntia)									
	1	2	3	4	5	6	7	8	9	10
HTTP-yhteyskäytäntö										
1	4627 12	4327 11	4816 10	4183 9	5000 8	4076 9	4431 11	4203 11	4333 11	4353 9
2	4226 21	4918 19	4353 21	4914 21	4244 20	4732 21	4361 22	4329 20	4603 20	4386 21
3	5039 75	4421 75	4559 76	4495 75	4486 75	4884 77	4425 74	4389 78	4590 75	4458 75
4	5386 125	6536 125	6552 126	6354 140	5442 131	5912 130	5383 124	5566 143	5676 144	5477 127
5	7299 239	8735 239	8978 241	7333 240	7463 245	7833 244	7860 239	7422 241	7514 252	8773 252
6	10644 492	11683 477	11730 484	11661 486	11543 480	11043 477	11077 486	11247 483	11549 484	12091 475
XML-yhteyskäytäntö										
1	4763 10	4932 8	4478 9	4641 11	4640 9	4444 11	4942 10	4548 9	4490 9	5247 12
2	5269 18	4733 18	4837 20	5568 18	4920 18	4877 19	4795 18	5391 19	4823 22	4731 18
3	6695 77	6559 67	6662 67	6649 75	7343 63	6511 67	6634 67	6772 68	6565 67	7473 68
4	7978 106	8092 106	7890 107	8003 108	8137 106	8593 106	8118 105	8242 105	8109 105	8136 120
5	12228 203	13189 218	13304 207	12079 205	12521 204	12377 225	13115 206	12502 204	12644 203	12055 217
6	19228 418	19134 420	19876 410	20872 400	20524 411	20594 399	20399 418	20654 402	19356 418	18382 419
XML-RPC -yhteyskäytäntö										
1	5316 10	5377 11	5638 9	4736 10	4594 11	4802 9	5162 9	4653 11	5182 9	4857 9
2	4944 19	4925 20	5337 20	4974 19	4776 22	4878 20	4958 19	5294 22	5061 31	4731 20
3	6304 75	6241 70	6284 68	6050 72	6116 73	5998 70	6939 82	7169 74	5937 72	5870 70
4	7846 120	6651 130	6492 122	6366 123	6480 128	6458 117	6438 124	7331 112	7397 120	6440 123
5	9031 247	9944 242	10023 260	9949 243	8859 245	9448 259	9609 248	9056 243	9325 255	10179 244
6	13024 403	14351 406	13715 411	13519 411	14017 403	12908 409	14030 412	12988 405	14307 415	14410 419
SOAP-yhteyskäytäntö										
1	5828 12	6334 10	6240 15	6459 11	6462 10	6684 10	5999 11	6062 9	7006 14	6225 9
2	6410 18	6256 18	5867 25	6007 18	6030 20	6054 18	6388 19	5919 19	6052 24	5832 20
3	7610 84	8969 77	7813 71	8732 74	8221 84	8362 79	7946 71	7815 79	7786 72	8148 71
4	9631 106	10225 120	9969 107	10218 120	11754 110	10608 122	11309 109	11551 122	9777 108	9913 121
5	13677 209	14600 221	13375 200	14749 232	14389 205	14417 218	13236 205	13277 220	13103 208	14983 225
6	20521 418	21427 415	21729 433	21531 420	21502 415	21449 414	22254 412	21498 419	22122 418	20856 417

Taulukko 3: Matkapuhelin *Motorola Accompli 008*.

Aineisto	Suorituskykykokeiden tulokset (millisekuntia)									
	1	2	3	4	5	6	7	8	9	10
HTTP-yhteyskäytäntö										
1	10205 3249	9680 3249	9370 4059	10595 3004	9460 2936	10224 3971	10261 3443	10247 4259	10197 3358	10137 3362
2	10334 3603	10419 3600	10325 4474	9397 3743	10413 3763	10989 4514	10735 4327	10250 3265	9667 3270	9904 4174
3	11777 3830	11761 4993	12281 4492	11842 4280	11783 3763	11486 3672	11837 4531	10620 3956	11302 4025	11149 4951
4	12338 4668	13163 4582	13114 5609	13128 4846	14355 4804	12882 5648	13456 4484	13573 5062	12500 4616	12661 4592
5	16409 6909	17164 6378	17417 6402	16804 7024	17856 8506	16490 6539	21185 6738	20010 7425	16860 7063	16315 6895
6	22780 10029	23320 10488	25645 9635	22530 10588	23056 9731	22454 10578	27609 9383	21856 10284	22855 9454	22418 10386
XML-yhteyskäytäntö										
1	10957 3414	11149 4091	10975 2903	12256 3401	11410 3399	10919 3526	10992 3524	12192 3529	11603 2966	11206 2906
2	12634 3487	12272 3684	12801 3679	12587 3674	12516 2890	12281 3013	13687 3705	12350 3139	12385 3353	13101 3933
3	18733 3592	19228 3752	19612 3757	18880 3746	21007 4353	19406 4405	18979 5151	19215 4211	20111 4256	18959 4933
4	27835 4666	26771 5471	26927 4354	26868 4573	26464 5814	26423 4657	26801 4961	26138 5645	27197 4780	26226 5065
5	41706 6863	44610 5927	41185 5833	42057 5907	41321 6016	40969 6941	41357 6004	41567 6947	43145 6299	42621 6515
6	72474 10223	72579 9289	75147 11190	72702 9592	72624 8872	73772 9393	73620 9651	72120 9956	73784 10116	72825 9058
XML-RPC -yhteyskäytäntö										
1	25723 3347	26925 3358	25295 3545	24922 3493	26026 4289	26228 2895	26965 2899	25336 3733	23450 3332	24820 3240
2	27528 3515	24208 3040	26284 3041	26074 3880	24295 3549	26876 3553	24278 4378	31558 3391	24852 3325	24989 3980
3	25353 3597	26299 3506	26653 4467	25343 3867	25524 3882	27002 4668	24466 3427	25198 3534	26648 3606	26187 3719
4	25006 4712	26130 3927	26254 4694	26750 4298	25226 4316	26451 5264	26368 4473	25535 4605	25974 4074	25969 3773
5	41274 5197	41779 5860	41837 4744	42080 5516	41813 5268	42131 4759	41992 5151	41462 5190	42330 6215	41650 5228
6	68545 7396	68258 6574	72429 7152	68382 7372	67591 6827	69430 7541	68386 6395	67634 7662	70447 6654	67952 7544
SOAP-yhteyskäytäntö										
1	18035 3783	17402 3783	17210 4735	17333 3642	18820 3682	17257 4547	18968 3961	17563 3975	19198 4309	19055 3785
2	18711 4388	18686 3717	19492 3159	18694 4448	18805 3349	18587 3348	18399 4381	19288 3699	19241 3815	18945 3811
3	26425 4012	24936 4025	24365 4747	25878 3679	25698 3666	25533 4095	26322 4100	26159 4036	25853 4141	25629 5206
4	32481 3939	32472 4066	31513 4080	31672 4911	32204 3730	31722 3745	32409 4815	31840 4100	31602 4951	32508 4005
5	50947 5142	48055 5123	49084 4528	51161 4462	48523 5639	47655 4622	49285 5741	48451 4880	48666 4793	48623 4740
6	82711 8074	81642 10280	79750 9284	84672 9271	79715 10176	78933 8380	78121 8765	78725 8967	79029 9854	79338 9971