Tommi Flink

SIIRTYMINEN MONITASOARKKITEHTUURIIN: MICROSOFT .NET:IN TARJOAMAT MAHDOLLISUUDET

Tietotekniikan pro gradu -tutkielma

Ohjelmistotekniikan linja

19.11.2001

Jyväskylän yliopisto

Tietotekniikan laitos

Tekijä: Tommi Flink

Yhteystiedot: tomflin@st.jyu.fi

Työn nimi: Siirtyminen monitasoarkkitehtuuriin: Microsoft .NET:in tarjoamat mahdollisuudet

Title in English: Transition to multi-tier architecture: Possibilities provided by Microsoft .NET

Työ: Pro gradu -tutkielma

Sivumäärä: 65+20

Linja: Ohjelmistotekniikka.

Teettäjä: Jyväskylän yliopisto, tietotekniikan laitos

Avainsanat: Monitasoarkkitehtuurit, Microsoft .NET

Keywords: Multi-tier architecture, Microsoft .NET

Tiivistelmä: Tutkielmassa tarkastellaan monitasoarkkitehtuurin tarjoamia mahdollisuuksia hajautettujen sovellusten rakentamisessa. Arkkitehtuuria verrataan asiakas/palvelin –arkkitehtuuriin ja siinä ilmenneisiin ongelmiin. Tarkasteluissa painotetaan näkökulmaa, että on olemassa asiakas/palvelin –mallin mukaisesti toteutettu sovellus, joka halutaan siirtää monitasoarkkitehtuuriin. Tarkastelemme miten siirtyminen monitasoarkkitehtuuriin on mahdollista, milloin se on perusteltua ja mitä siinä tulisi erityisesti huomioida.

Tutkielmassa käydään läpi monitasoarkkitehtuurin mukaisten tietojärjestelmien rakentamiseen tarjolla olevia teknologioita ja työvälineitä, joista pääpaino on Microsoft .NET:in tarjoamilla mahdollisuuksilla.

Abstract: Possibilities provided by multi-tier architecture in implementation of distributed applications is studied in this thesis. This architecture is compared to client/server – architecture and the problems involved with it. The case where an existing client/server – application is to be transited into multi-tier architecture is emphasized. We will examine how the transition to multi-tier architecture is possible, when it is reasonable, and what should be notified especially.

Tools and technologies available in the building of multi-tier architecture based information systems are studied. The focus is on the possibilities provided by Microsoft .NET.

Termiluettelo

.NET	Microsoftin uusi teknologia mm. sovellusten hajauttamiseen.
ASP	Active Server Page, Microsoftin dynaamisten www-sivujen rakentamiseen kehittämä teknologia.
ASP.NET	ASP:n seuraaja.
Assembly	.NET sovellusten asennuspaketti.
CORBA	Common Object Request Broker Architecture, teknologia ohjelmistojen hajauttamiseen, käytetään paljon varsinkin Java / J2EE –sovelluksissa.
CLR	Common Language Runtime, .NET sovellusten ajonaikainen suoritusympäristö.
COM	Component Object Model, Microsoftin komponenttimalli.
COM+	COM:n ja MTS:n yhteenliittämisen tuloksena syntynyt teknologia sovellustenkomponenttipohjaiseen hajauttamiseen.
CTS	Common Type System, .NET:n kaikille ohjelmointikielille yhteisiä tietotyyppejä.
C++	Ohjelmointikieli.
C#	Microsoftin uusi ohjelmointikieli.
DCOM	Distributed Component Object Model, tapa kutsua COM-komponentteja.

HTTP	HyperText Transfer Protocol, internetissä tiedonsiirtoon usein käytetty tiedonsiirtoprotokolla.
Java	Ohjelmointikieli.
J2EE	Java 2 Enterprise Edition.
Käyttöliittymäkerros	Kerros monitasoarkkitehtuurissa, joka hoitaa tiedon esittämisen ja käsittelee käyttäjän suorittamat toiminnot. Käyttöliittymäkerrokselta on yhteys sovelluslogiikkakerrokseen.
MTS	Microsoft Transaction Server, palvelinmalli mm. DCOM:in avulla kutsuttavien COM-komponenttien hallintaan.
MSIL	Microsoft Intermediate Language, .NET sovellusten välikieli, joka käännetään prosessorin ymmärtämään muotoon.
MSMQ	Microsoft Message Queue Server.
SOAP	Simple Object Access Protocol, protokolla jonka avulla mm. .NET-sovellukset kommunikoivat keskenään.
Tilaton yhteys	Kun yhteys esim. asiakassovelluksen ja sovelluspalvelimen välille otetaan ainoastaan palvelukutsun tai vastauksen ajaksi, eikä sovelluspalvelimelle jää mitään tilaa asiakkaasta (esim. oliota), yhteys on tilaton.
Sovelluslogiikkakerros	Kerros monitasoarkkitehtuurissa, jossa sijaitsee sovelluksen toimintalogiikka.
Sovelluspalvelin	Palvelin, jolla monitasoarkkitehtuurissa toimii sovelluslogiikkakerros.
XML	Extensible Markup Language, tiedon kuvauskieli.

Sisältö

1 Johdanto

90-luvulla tietojärjestelmät rakennettiin usein asiakas/palvelin -arkkitehtuurin mukaisiksi. Asiakas/palvelin –mallissa asiakaskoneena toimii usein PC-työasema, jolta otetaan yhteys palvelimelle, yleensä tietokantapalvelimelle. Ohjelmistojen kaikki toiminnallisuus on sijoitettu asiakaskoneelle, palvelin huolehtii yleensä vain esimerkiksi tiedon varastoinnista.

Internetin yleistyminen on tuonut mukanaan uudenlaisia tarpeita ja mahdollisuuksia. Nykyisten järjestelmien tulee kyetä käsittelemään yhä suurempia käyttäjämääriä, joka ei aina asiakas/palvelin -ympäristössä onnistu. Asiakas/palvelin-, eli kaksitasoarkkitehtuurin, yhtenä ongelmana on sen skaalautuvuus. Kaksitasoarkkitehtuurin tietojärjestelmää on vaikea laajentaa käsittelemään uusia käyttäjiä, edes konekapasiteettia lisäämällä. Ylärajan käyttäjien määrälle asettaa lähinnä palvelimen resurssit, oli käytösssä sitten tietokanta- tai esim. tiedostopalvelin, joissa yhtäaikaisten käyttäjien määrä on teknisten syiden vuoksi rajallinen. Lisäksi kaksitasoarkkitehtuurissa on järjestelmien hallinnointi hankalaa, koska sovelluslogiikka sijaitsee useissa eri työasemakoneissa, tehden päivitykset työläiksi.

Monitasoarkkitehtuurissa, jossa mahdollisimman paljon sovelluslogiikkaa on siirretty omalle palvelimelleen, on pyritty korjaamaan kaksitasoarkkitehtuurin ongelmia, sekä lisäämään uusia mahdollisuuksia. Monitasoarkkitehtuurissa asiakaskoneella sijaitsee lähinnä vain käyttöliittymä, sovelluslogiikka on omalla palvelimellaan, joiden lisäksi käytetään vielä usein erillistä tietokantapalvelinta.

Monitasoarkkitehtuuria, jossa järjestelmä on jaettu kahdelle eri palvelimelle sekä asiakaskoneisiin, voidaan kutsua kolmitasoarkkitehtuuriksi. Periaatteena kolmitaso- ja muissa monitasoarkkitehtuureissa on, että yksittäistä sovelluskerrosta on mahdollista muokata ilman muutoksia muiden kerrosten toimintaan, mikäli kerrosten välinen kommunikointitapa ei muutu.

Kolmi- ja monitasoarkkitehtuurin toteuttamiseen komponentti- ja oliotekniikoita hyödyntäen on olemassa useita välineitä, joista tunnetuimpina mainittakoon Java (+CORBA), sekä Microsoftin teknologiat COM, DCOM ja .NET.

Kaksitasoarkkitehtuuriin rakennettujen tietojärjestelmien siirtäminen toimimaan monitasoarkkitehtuureissa saattaa aiheuttaa eriasteisia ongelmia. Kaksitasoarkkitehtuuriin toteutettuja järjestelmiä on olemassa lukemattomia, joista kaikissa joudutaan samojen kysymysten eteen. Siksi on tärkeää tutkia, mitä vaaditaan järjestelmän siirtämiseksi monitasoarkkitehtuuriin ja mitä apuvälineitä siihen on olemassa.

Tässä tutkielmassa on tarkoituksena selvittää, kuinka työasema/palvelin -arkkitehtuurin mukainen tietojärjestelmä voitaisiin siirtää monitasoarkkitehtuuriin ja saavutetaanko siirtymisellä ylipäätään riittäviä etuja siihen vaadittavan työmäärän suhteen. Tarkasteluissa painotetaan sitä näkökulmaa, että on olemassa kaksitasomallin mukainen nykyjärjestelmä, joka haluttaisiin uudistaa vastaamaan nykyajan vaatimuksia.

Teknisen toteutuksen kannalta tässä tutkielmassa tarkastellaan lähinnä Microsoftin .NET arkkitehtuurin suomia mahdollisuuksia.

2 Työasema/palvelin –arkkitehtuuri

Työasema/palvelin –arkkitehtuurilla, jota kutsutaan myös asiakas/palvelin -arkkitehtuuriksi, tarkoitetaan yleensä tapausta, jossa on yksi tai useampia asiakaskoneita, jotka kaikki ottavat yhteyden samaan palvelinkoneeseen. Palvelinkone voi olla esim. tietokantapalvelin. Tässä kappaleessa käsittelemme työasema/palvelin -arkkitehtuurin historiaa, toteutustapoja, sekä siinä ilmenneitä ongelmia.

2.1 Historiaa

Työasema/palvelin –arkkitehtuuri siinä muodossa kuin se nykyään useimmiten käsitetään alkoi muotoutua 80-luvun lopussa, kun PC-koneiden muodostamat verkot yleistyivät [Dar97b]. Ensimmäiset työasema/palvelin –järjestelmät rakennettiin 90-luvun alkupuolella, jonka jälkeen arkkitehtuuri on yleistynyt voimakkaasti. Etenkin 90-luvun puolivälin tienoilla työasema/palvelin -arkkitehtuuri, josta käytetään myös nimeä kaksitasoarkkitehtuuri, oli tyypillinen tapa rakentaa tietojärjestelmiä.

Monen käyttäjän tietojärjestelmiä on rakennettu jo kymmeniä vuosia ennen asiakas/palvelin –arkkitehtuuria. Ennen PC-koneiden yleistymistä asiakaskoneina toimivat usein eritasoiset päätteet, jotka olivat suoraan yhteydessä palvelimeen. Ns. tyhmät päätteet, joita on paikoin käytössä vieläkin, eivät hoida lainkaan tiedon käsittelyä, vaan kaikki toiminnallisuus on palvelimella, jopa näytön piirtämistä myöten. Hieman enemmän toiminnallisuutta sisältävät ns. lohkopäätteet (engl. block mode terminal), jotka jo osaavat itse hoitaa näytön käsittelyä. Lisäksi mainittakoon ns. X-päätteet, jotka ovat ikäänkuin lohkopäätteitä graafisella käyttöliittymällä.

Käytettäessä eriasteisia päätteitä asiakaskoneina, palvelimena toimii ns. keskuskone, joka hoitaa kaiken tiedon käsittelyn ja varastoinnin. Keskuskoneen voidaan ajatella jakautuvan useisiin kerroksiin, kuten nykyinen kaksi- tai monitasoarkkitehtuuri. Keskuskoneella on oma osa-alue, joka hoitaa näytön piirtämisen asiakaspäätteelle tai asiakaspäätteen kanssa kommunikoinnin. Tiedon varastointiin on oma kerroksensa ja näiden kerrosten välillä on kerros, joka hoitaa varsinaisen tiedon prosessoinnin.

70-luvun monitasoarkkitehtuuri-sovelluksissa käytettiin usein ns. tapahtumankäsittelymonitoria (engl. Transaction Processing Monitor), jonka kautta asiakaskoneet suorittivat tietokantahaut yms. Tapahtumankäsittelymonitorit, eli TP-monitorit, hoitavat mm. tietokantojen ja muiden resurssien käytön koordinoinnin, palvelupyyntöjen reitittämisen ja tapahtumien suorituksen priorisoinnin ja valvonnan.

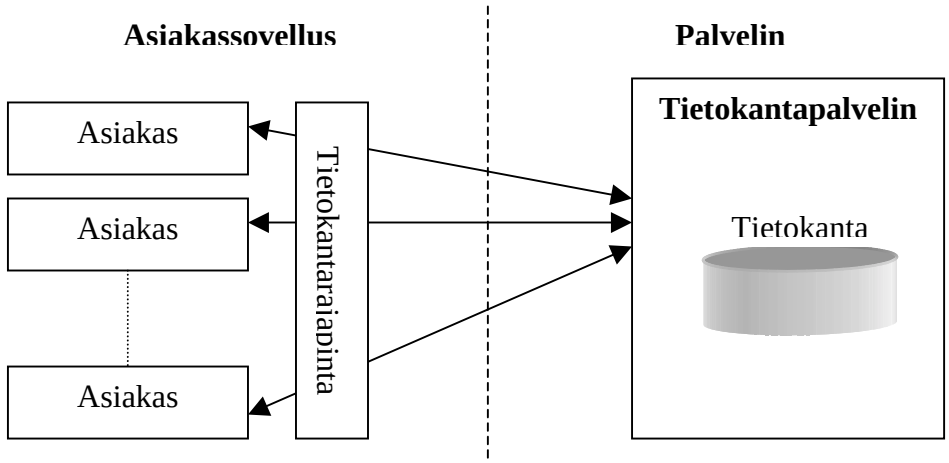
PC-koneiden yleistyessä ja verkottuessa, siirryttiin kohti työasema/palvelin –arkkitehtuuria, mm. sovelluskehityksen helpottumisen ja graafisten käyttöliittymien vuoksi. Ennen esim. TP-monitorien suorittamat toiminnot suurelta osin sivuutettiin keskuskoneiden käytön vähentyessä.

Työasema/palvelin –arkkitehtuurimallin katsotaan olevan lähtöisin yksinkertaisesta tiedostopalvelimen käytöstä, jossa palvelinkoneelta ladataan tiedosto asiakaskoneelle ja kaikki tietojen käsittely suoritetaan asiakaskoneessa [Sch96]. Tiedostopalvelin toimii kuitenkin hyvin vain tilanteissa, jossa käyttäjiä on vähän ja tietojen päivitys tapahtuu harvoin. Usean yhtäaikaisen käyttäjän päivittäessä tietoja tiedon eheyden tarkistaminen on vaikeaa, koska sama tieto voi olla usean käyttäjän päivitettävissä samaan aikaan.

Kun lähiverkot yleistyivät ja käyttäjämäärät kasvoivat tarvittiin tiedostopalvelimen tilalle jotain uutta. Tähän tarpeeseen sopi asiakas/palvelin –malli, jossa palvelimena on tietokantapalvelin. Tietokantapalvelimen ansiosta jaettu tieto on helpompi pitää ajantasalla, vaikka olisi useita samanaikaisia käyttäjiä. Lisäksi tiedostopalvelimeen verrattuna liikenne verkossa väheni huomattavasti, koska aiemman kokonaisten tiedostojen siirtämisen sijaan verkossa liikkui lähinnä vain tietokantakyselyt sekä kyselyjen tulokset. Palvelin ei välttämättä ole aina tietokantapalvelin, vaikkakin se on asiakas/palvelin –arkkitehtuurissa yleisin palvelinmalli. Palvelin voi tietokantapalvelimen sijasta olla esim. tulostuspalvelin, tietoliikennepalvelin jne.

2.2 Teknologioista

Tässä kappaleessa käsittelemme lähinnä asiakas/palvelin –arkkitehtuuria, jossa palvelimena on tietokantapalvelin. Kuvassa 1 näemme tyypillisen asiakas/palvelin ratkaisun.



Kuva 1. Asiakas/palvelin –arkkitehtuuri

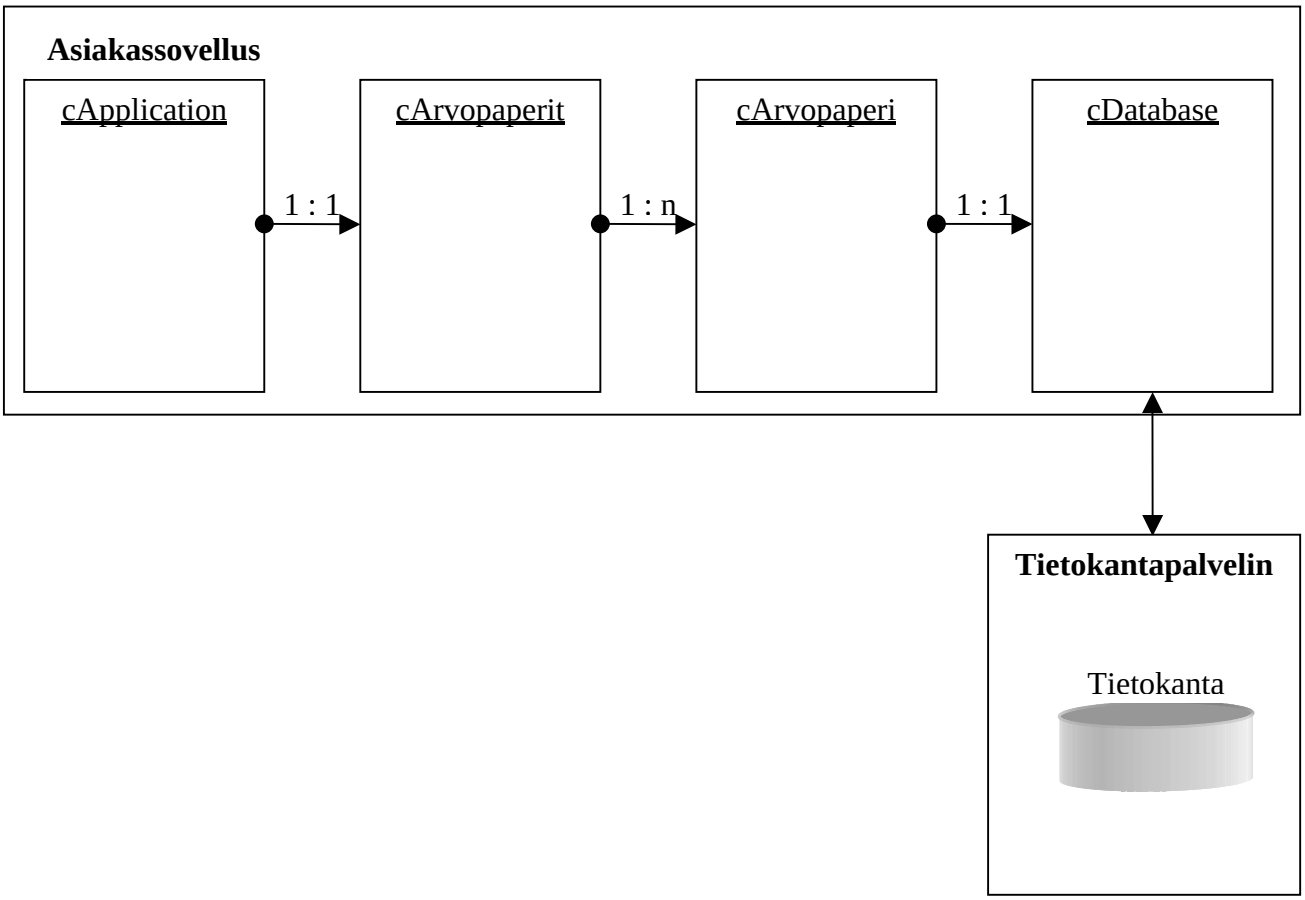
Kuvan 1 tilanne on tyypillinen asiakas/palvelin –malli. Siinä on yksi tietokantapalvelin, johon useat asiakaskoneet ovat suoraan yhteydessä tietokantarajapinnan kautta.

Asiakassovelluksen rakentamiseen on tarjolla useita kehitysvälineitä. Asiakassovellus voidaan toteuttaa periaatteessa samoilla välineillä, kuin tehtäisiin pelkkä asiakaskoneella toimiva sovellus, ilman yhteyttä palvelimelle. Kehitysvälineistä Windows ympäristöön mainittakoon esim. Borlandin C++ Builder ja Delphi, sekä Microsoftin Visual Studio.

Kehitysvälineet tukevat usein tiettyjä rajapintoja tietokantaan esim. ODBC (Open DataBase Connectivity) tai JDBC (Java DataBase Connectivity) -rajapintaa [Ble00, s.398] Rajapinnan lisäksi tarvitaan käytettävälle tietokantapalvelimelle rajapintaa tukevat ajurit.

Tietokantatoimittajia on myös lukuisia, tunnetuimpina mainittakoon Oracle ja Microsoft.

Kappaleessa 4.2.5. rakennamme esimerkkisovelluksen, joka toteuttaa kaksitasomallin mukaisen rakenteen. Kuvassa 2 on esimerkkisovelluksen luokkakaavio.



Kuva 2. Esimerkkisovelluksen luokkakaavio

Esimerkkisovelluksessamme haetaan arvopapereiden perustietoja tietokannasta. cApplication-luokka sisältää metodit käyttäjän käyttöliittymän hallintaan, sisältäen mm. käyttäjän komentojen tulkitsemisen sekä tietojen tulostamisen näytölle. cArvopaperit-luokka hallinnoi ja luo cArvopaperi–luokkia luodaan tarpeen mukaan. cArvopaperi–luokka luo tarvittaessa cDatabase–luokan ilmentymän, jonka kautta hoidetaan yhteys mahdollisesti toisella palvelimella sijaitsevaan tietokantaan.

Kaikki sovelluslogiikka toimii siis asiakaskoneella, palvelimelta käytetään ainoastaan tietokantaa. Kaksitasoarkkitehtuurin asiakassovellusta voidaan kutsua myös ”lihavaksi asiakkaaksi”, engl. fat client. Seuraavassa kappaleessa tarkastelemme mitä ongelmia kaksitasoarkkitehtuurista ilmenee.

2.3 Ongelmia työasema/palvelin –arkkitehtuurissa

Työasema/palvelin –arkkitehtuuri asettaa käytännössä ylärajan käyttäjien määrälle. Esim. palvelimen ollessa tietokantapalvelin, jokaiselta asiakaskoneelta on suora yhteys tietokantaan, joka kuormittaa tietokantapalvelinta sen varatessa resursseja jokaisen asiakkaan käyttöön. Satojen, tai viimeistään tuhansien käyttäjien asiakas/palvelin -järjestelmissä tietokantapalvelin helposti kuormittuu liikaa ja skaalaaminen tietyn pisteen yli on vaikeaa [Dar97a]. Yhteyden avaaminen tietokantaan ainoastaan kyselyä suoritettaessa olisi periaatteessa mahdollista, mutta ei tarkoituksenmukaista, koska yhteyden avaaminen vie liian kauan aikaa.

Jokainen tietokantahaku vaatii useita viestejä asiakas- ja palvelinkoneen välillä, lisäksi asiakas– ja palvelinkoneen välillä liikkuu runsaasti tietoa, jolla ne pitävät yhteyttä yllä, vaikka varsinaisia tietokantakutsuja ei lähetettäisikään. Verkossa liikkuu siis suuri tietoliikennekuorma erinäisten viestien muodossa.

Sovellusten asennus ja ylläpito käy käyttäjämäärän noustessa työlääksi, koska sovellukset on asennettava kaikkiin asiakaskoneisiin erikseen. Koska jokaiselta asiakaskoneelta on myös suora yhteys tietokantaan, on kaikilla asiakaskoneilla oltava sopivat tietokanta-ajurit. Tämä osaltaan teettää lisää työtä sovellusten asentamiseen.

On myös tilanteita, jolloin sovellusten täytyy päästä käsiksi useampaan, mahdollisesti jopa eri palvelimilla sijaitsevaan, tietokantaan, jotka vaativat eri tietokanta-ajurit [Ble00]. Tällaisessa tapauksessa asiakaskoneeseen pitää asentaa useita eri ajureita tukemaan tietokantayhteyksiä.

Sovelluslogiikkaa on periaatteessa mahdollista osittain sijoittaa myös palvelinkoneelle tietokantaproseduurien yms. muodossa, mutta se kuormittaa palvelinta entisestään, ja näin ollen rajoittaa käyttäjämäärää. Jos sovelluslogiikkaa sijoitetaan tietokantapalvelimelle, saadaan eräänlainen 2½-tasomalli, joka saattaa osin helpottaa sovellusten hallintaa, mutta ei kuitenkaan ratkaise suuren käyttäjämäärän tuomia ongelmia.

Sovelluslogiikan sijoittaminen tietokantapalvelimelle hankaloittaa myös mahdollista sovellusten siirtoa toiseen ympäristöön. Koska tietokantapalvelimille kirjoitetut proseduurit toimivat usein vain saman palvelintoimittajan tietokannoissa, tietokantapalvelinta vaihdettaessa voidaan joutua kirjoittamaan sovelluslogiikkaa uudestaan. Lisäksi tietokantaproseduurien kirjoittaminen ja testaaminen saattaa jossain määrin olla vaikeampaa, kuin olisi saman logiikan testaaminen asiakassovelluksessa.

Seuraavassa kappaleessa tarkastelemme monitasoarkkitehtuuria, jota voidaan pitää kaksitasoarkkitehtuurin seuraajana. Monitasoarkkitehtuurissa on pyritty ratkaisemaan kaksitasoarkkitehtuurissa ilmenneitä ongelmia, joita olemme tässä luvussa käsitelleet.

3 Monitasoarkkitehtuurit

Monitasoarkkitehtuurilla voidaan ratkaista useita ongelmia, joita ilmenee asiakas/palvelin –arkkitehtuurissa [Dar97c]. Keskeisenä ajatuksena monitasoarkkitehtuureissa on, että sovelluslogiikka eriytetään käyttöliittymästä omalle palvelimelleen, jolloin asiakaskoneelle jää vain käyttöliittymä. Kun vielä tiedon varastointi on omalla palvelimellaan, on toiminnassa vähintään kolme sovelluskerrosta. Monitasoratkaisua, jossa sovellus jaetaan kolmeen eri kerrokseen, voidaankin kutsua kolmitasoarkkitehtuuriksi.

Vaikka monitasoarkkitehtuuriin ollaankin käytännössä siirtymässä asiakas/palvelin –arkkitehtuurista, on monitasoarkkitehtuuri asiakas/palvelin –arkkitehtuurin tapaan ideana vanha. Sovelluksia on rakennettu monitasoarkkitehtuuriin jo ennen kuin asiakas/palvelin –malli tuli suosituksi 1990-luvun puolivälissä. Sovelluspalvelimena käytettiin ns. tapahtumankäsittelymonitoria (engl. Transaction Processing Monitor), jonka kautta suoritettiin tietokantahaut yms. Sovellukset monitasoarkkitehtuuriin ovat kuitenkin olleet työläämpiä rakentaa kuin kaksitasoarkkitehtuurisovellukset, joka on osaltaan vaikuttanut kaksitasoarkkitehtuurisovellusten yleistymiseen. Nykyään monitasoarkkitehtuurisovellusten yleistyttyä myös niiden rakentaminen on käynyt helpommaksi uusien kehitysvälineiden ja tekniikoiden myötä.

3.1 Sovelluskerrokset

Monitasoarkkitehtuurissa on vähintään kolme sovelluskerrosta: käyttöliittymä-, sovelluslogiikka- ja esim. tietokantakerros.

Käyttöliittymäkerros toimii asiakaskoneella ja sen tulisi sisältää vain kaikki tiedon esittämiseen ja käyttäjän komentojen käsittelyyn vaadittavat toiminnot, kaikki pidemmälle menevä toiminta tulisi sisällyttää sovelluslogiikkakerrokseen. Asiakassovellusta, jossa on vain käyttöliittymätoiminnot voidaan kutsua ”ohueksi asiakkaaksi”, engl. thin client (vrt. fat client kappaleessa 2.2). Käyttöliittymäkerros kommunikoi ainoastaan sovelluspalvelimen kanssa, josta tarvittaessa otetaan yhteys palvelimelle, joka hoitaa tiedon varastoinnin, esim. tietokantapalvelimelle.

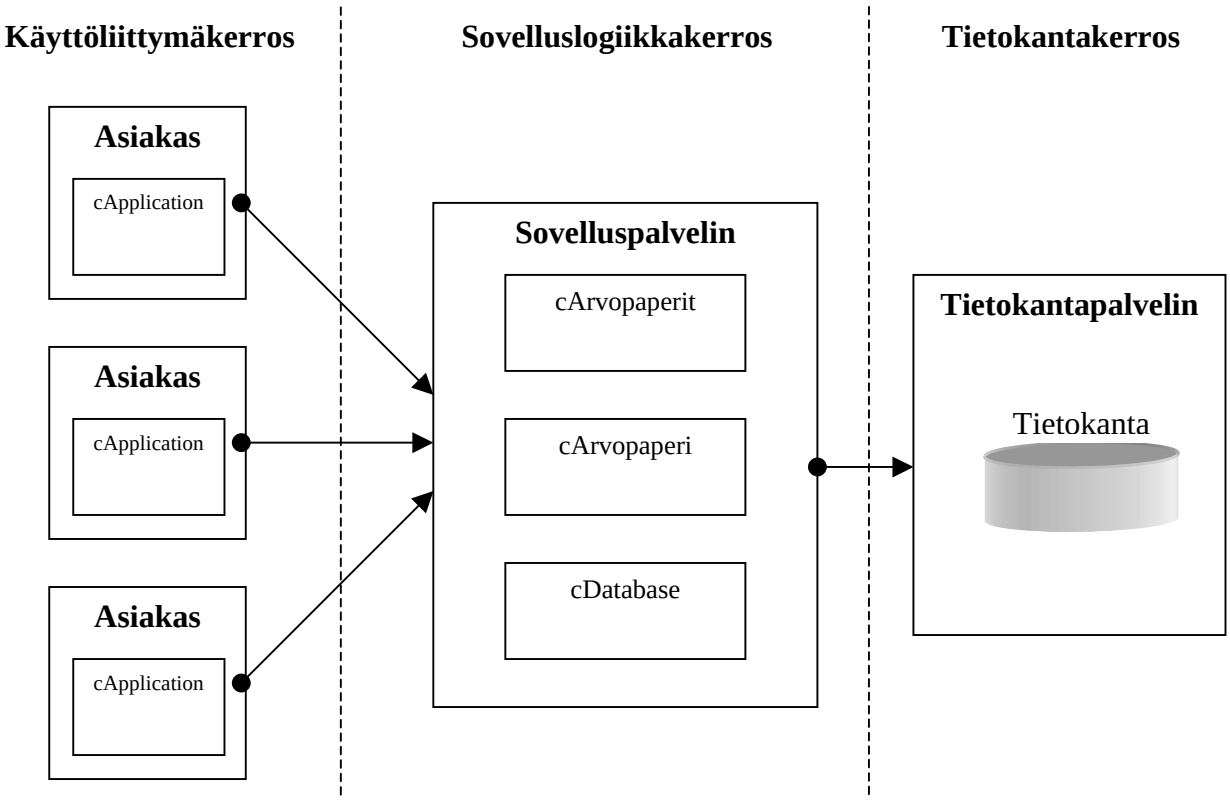
Sovelluslogiikkakerrokseen kuuluu kaikki ns. sovelluslogiikka, joka asiakas/palvelin –järjestelmässä kuului asiakaskoneelle (kts. kappale 2.2). Tämä helpottaa mm. ohjelmistojen päivitystä, koska sovelluslogiikan päivitys ja mahdolliset rekisteröinnit yms. tarvitsee tehdä ainoastaan yhdelle koneelle, sovelluspalvelimelle. Sovelluslogiikan päivittäminen on usein mahdollista ilman muutoksia käyttöliittymään, kunhan sovelluskerrosten välinen kommunikointitapa ei muutu. Näin esimerkiksi sovelluslogiikassa olevia laskusääntöjä ja tiedon prosessointia voidaan muuttaa asiakassovelluksen siitä tietämättä. Asiakaskoneen ja sovelluspalvelimen välisen yhteyden toteuttamiseen on useita eri vaihtoehtoja, joita käsitellään kappaleessa 3.4.

Usein joustavin järjestelmä saadaan aikaan, jos asiakaskoneen ja sovelluspalvelimen välinen yhteys on ns. tilaton. Tilattomalla tarkoitetaan, että esim. ohjelmakutsu sovelluspalvelimelle luo olion, joka suorittaa halutun toiminnon, jonka jälkeen olion varaama muisti vapautetaan. Asiakaskoneelta ei siis voi luoda oliota ja myöhemmin kutsua sen metodeja. Tällä tavoin liikenne verkossa pysyy pienempänä ja sovelluspalvelimen resursseja säästyy, koska muistia ei varata turhaan.

Sovelluslogiikan eriyttäminen omaksi kerroksekseen mahdollistaa myös erilaisten käyttöliittymien rakentamisen samalle sovelluslogiikalle. Kun kaksitasomallissa (kts. kappale 2) uuden käyttöliittymän rakentaminen samalle sovelluslogiikalle vaatisi sovelluslogiikan ohjelmointia uudelleen, monitasoarkkitehtuurissa käyttöliittymän tarvitsee vain tietää kuinka kommunikoida sovelluspalvelimen kanssa.

Monitasoarkkitehtuurin kolmantena kerroksena voi toimia esim. tietokantapalvelin. Yhteydet käyttöliittymäkerrokselta muihin kerroksiin tapahtuvat aina sovelluslogiikkakerroksen kautta, niin myös yhteydet tietokantapalvelimelle. Tietokantapalvelimen tapauksessa tietokantayhteyksien keskittetty hallinta takaa joustavan resurssien jaon, jossa yhteyksiä ei varata jokaiselle asiakkaalle yhtä aikaa. Sovelluslogiikkakerros osaa avata riittävän määrän tietokantayhteyksiä ja jakaa niitä asiakakaskoneille tarpeen mukaan. Tämä on eräs tärkeimmistä monitasoarkkitehtuurin ominaisuuksista. Kun kaksitasomallissa jokaiselta asiakaskoneelta on suora yhteys tietokantaan (kts. kappale 2.3), tietokantapalvelimen resurssit loppuvat käyttäjämäärän kasvaessa kesken. Yleensä kuitenkaan kaikki asiakaskoneet eivät samaan aikaan käytä tietokantayhteyttä, jolloin sovelluspalvelimen on mahdollista jakaa tietokantayhteyksiä ainoastaan asiakaskoneen tarvitsemaksi ajaksi. Näin, sovelluksesta riippuen, tietokantayhteyksiä tarvitaan vain murto-osa asiakaskoneiden lukumäärästä.

Kun kappaleessa 2.2. käsittelemämme esimerkkisovelluksen luokkakaavio sijoitetaan toimimaan kolmitasoarkkitehtuurissa, luokkamalli jakautuu palvelimille kuvan 3 mukaisesti.



Kuva 3. Esimerkkisovellus kolmitasoarkkitehtuurissa

cApplication luokka jää asiakaskoneelle (vrt. kaksitasomalli kappaleessa 2.2), sovelluspalvelimelle siirretään cArvopaperit, cArvopaperi ja cDatabase luokat. Lisäksi tarvitaan yhteystapa sovelluspalvelimen ja asiakaskoneiden välille. Sovelluspalvelimella sekä asiakaskoneessa pitää olla luokka, joka hoitaa yhteyden muodostamisen, sekä tiedon lähettämisen ja vastaanottamisen verkon yli.

3.2 Yleisiä vaatimuksia sovelluspalvelimille monitasoarkkitehtuurissa

Koska monitasoarkkitehtuurissa asiakaskoneet ottavat siis yhteyden sovelluspalvelimelle, ei suoraan esim. tietokantapalvelimeen, sovelluspalvelimen tulee kyetä käsittelemään useita samanaikaisia yhteyksiä asiakaskoneilta. Toisin kuin kaksitasoarkkitehtuurissa, jossa asiakkaiden määrä on rajoitettu tietokantayhteyksen rajoitetun lukumäärän vuoksi, monitasoarkkitehtuurissa asiakkaiden määrä voi periaatteessa olla rajaton. Monitasoarkkitehtuurissa tietokantapalvelimen ei tarvitse hallinnoita tuhansia yhteyksiä, vaan ainoastaan ennalta säädettyä määrää. Yhteydet asiakaskoneilta sovelluspalvelimelle on paras hoitaa tilattomasti, eli otetaan yhteys ainoastaan kun sitä tarvitaan (kuten tavallisesti esim. www-palvelimelle).

Monitasoarkkitehtuurissa pelkkä käyttäjien määrä ei sinänsä kuormita tietokantapalvelinta. Käytännössä yläraja muodostuu mm. käyttäjien suorittamien toimintojen lukumäärän perusteella, ei kuitenkaan pelkän käyttäjämäärän.

Asiakkaiden suuren määrän mahdollistaa sovelluspalvelimen skaalautuvuus. Skaalautuvuudella tarkoitetaan tilannetta, jossa palvelimen resursseja (esim. prosessoritehoa tai muistia) lisäämällä asiakasmäärää pystytään lisäämään. Esimerkkinä ajatellaan kappaleessa 3.1. esitetyn kuvan 3 tilannetta. Oletetaan että samanaikaisia asiakkaita, eli käyttäjiä, on yli 1000 kappaletta ja sovelluspalvelimelta on tietokantapalvelimelle koko ajan 100 kappaletta yhteyksiä auki. Parhaassa tapauksessa asiakaskoneiden ja sovelluspalvelimen välinen yhteys on tilaton, jolloin niiden välillä verkossa ei liiku turhaa tietoa. Asiakaskoneesta käsin vaikuttaa siltä, että yhteys on koko ajan auki, mutta käytännössä yhteys sovelluspalvelimeen otetaan vain tarvittaessa.

Oletetaan että asiakas suorittaa yhden tietokantaa vaativan toimenpiteen minuutissa. Tällöin 1000 käyttäjällä tietokantatoimintoja syntyy noin 17 kappaletta sekunnissa. Sovellus- ja tietokantapalvelimen tulisi pystyä suorittamaan kyseinen määrä toimintoja. Jos toimintojen suoritus toimii liian hitaasti, lisäämällä palvelimien suorituskykyä toimintojen tulisi onnistua. Kun kaksitasoarkkitehtuurissa käyttäjien määrän kasvattaminen suorituskykyä lisäämällä yli tietyn pisteen on vaikeaa, monitasoarkkitehtuurissa tilattomat yhteydet mahdollistavat tilanteen, jossa tuplaamalla palvelimien suorituskyky voidaan periaatteessa tuplata myös suoritettavien toimenpiteiden määrä, joka taas suoraan mahdollistaa myös suuremman asiakkaiden määrän. Tarvittaessa sovelluslogiikka voidaan myös jakaa useammalle palvelimelle, mikä taas ei samoissa määrin asiakas/palvelin -arkkitehtuurissa onnistu.

Skaalautuvuutta voidaan pitää yhtenä tärkeimmistä sovelluspalvelimien vaatimuksista, mutta ei suinkaan ainoana. Sovelluspalvelimien tulee myös olla luotettavia, eli niiden tulee kyetä palvelemaan asiakkaita lähes koko ajan. Esimerkiksi Microsoft markkinoi uusien palvelintuotteiden kykenevän toimimaan 99,999% ajasta, joka tarkoittaa noin 5 minuutin käyttökatkosta vuodessa. Näin hyvään toimivuuteen kuitenkaan harvoin ylletään.

Sovelluspalvelimien avulla tulee olla myös mahdollista hallita tapahtumankäsittelyä. Tällä tarkoitetaan tilannetta, jossa suoritettava tapahtuma koostuu tapahtumaketjusta. Jos joku ketjun tapahtumista epäonnistuu, koko ketju perutaan. Klassinen esimerkki aiheesta on tilisiirron tekeminen. Oletetaan että tililtä A siirretään tietty summa tilille B. Ensin vähennetään summa tililtä A, jonka jälkeen se on tarkoitus lisätä tilille B, mutta ennen kirjaamista tilille B tapahtuu virhe. Tällöin tililtä A on kadonnut rahat, mutta niitä ei ole tullut tilille B. Tässä tapauksessa tililtä A tehty nosto tulisi automaattisesti perua. Sovelluspalvelimen tulee kyetä hoitamaan vastaavat tilanteet.

Sovelluspalvelimen tulee kyetä hoitamaan myös ns. kuormantasausta (engl. load balancing), jossa toimintoja siirretään mahdollisuuksien mukaan vähemmän kuormitetuille palvelimille. Palvelimen tulisi kyetä myös ns. hajautettuun tapahtumankäsittelyyn, jossa tietoa joudutaan päivittämään useisiin eri tietokantoihin, mahdollisesti eri palvelimille.

3.3 Mitä hyötyjä saadaan monitasoarkkitehtuurista

Kappaleessa 3.2. esitellyt vaatimukset palvelimille ovat sellaisenaan parannuksia verrattuna asiakas/palvelin –arkkitehtuuriin. Monitasoarkkitehtuurista saadaan myös muita hyötyjä, joita käsittelemme tarkemmin tässä kappaleessa. [Sch96]

Monitasoarkkitehtuuri helpottaa sovellusten asennusta ja ylläpitoa [Ble00, s.16]. Asentaminen helpottuu varsinkin tilanteissa, joissa käyttöliittymänä toimii selainkäyttöliittymä, jolloin asiakaskoneelle ei usein tarvita lainkaan lisäasennuksia.

Sovellusten ylläpito helpottuu päivitysten keskittämisen myötä. Koska kaikki sovelluslogiikka sijaitsee samassa paikassa, sovelluspalvelimella, asiakaskoneissa mahdollisesti sijaitseviin sovelluksiin tarvitsee harvemmin tehdä muutoksia. Asiakaskoneiden sovelluksia pitää päivittää ainoastaan siinä tapauksessa, että käyttöliittymä muuttuu. Tämäkin vain tapauksissa, joissa käyttöliittymä on oma asiakaskoneella toimiva sovellus, ei selainkäyttöliittymä.

Sovelluslogiikan keskittäminen sovelluspalvelimelle parantaa myös tietoturvaa, koska sovelluksia on helpompi kontrolloida. Kaikki tietokantahaut ja tiedon käsittelyt suoritetaan sovelluspalvelimelta käsin, jolloin väärinkäytöksiä on helpompi ehkäistä, kuin kaksitasoarkkitehtuuriin rakennetuissa sovelluksissa.

Kun sovelluslogiikka on eriytetty käyttöliittymästä, sen uudelleenkäyttö on helpompaa. Uudelleenkäyttöä voi olla esim. uuden käyttöliittymän rakentaminen samalle sovelluslogiikalle tai sovelluslogiikan siirtäminen toimimaan uudessa ympäristössä.

Mahdollinen tietokantapalvelimen vaihto on myös helpompaa monitasoarkkitehtuurissa kuin kaksitasoarkkitehtuurissa. Koska kaikki tietokantayhteydet hoidetaan sovelluspalvelimelta, asiakas ei välttämättä näe mitään muutosta tietokantapalvelimen vaihtuessa, koska kaikki tiedonsiirto pysyy asiakkaan kannalta muuttumattomana, eikä esim. uusia tietokanta-ajureita tarvita asiakaskoneisiin.

Sen lisäksi, että monitasoarkkitehtuuri on helposti skaalautuva (kts. kappale 3.2.) palvelimien suorituskykyä lisäämällä, sovelluslogiikka on mahdollista jakaa myös useammalle palvelimelle.

3.4 Teknologioista

Tässä luvussa tarkastelemme lähinnä teknologioita, joilla on mahdollista toteuttaa käyttöliittymä- ja sovelluskerroksen vaatimat järjestelmät. Tietokantapalvelin on periaatteessa vastaava kuin kaksitasoarkkitehtuurissa, myös yhteydet sovelluskerrokselta eteenpäin voidaan hoitaa vastaavilla tekniikoilla, kuin kaksitasoarkkitehtuurissa suoraan asiakaskoneelta (esim. ODBC- tai JDBC-rajapinnat, katso kappale 2.2.).

Sovelluskerroksen ja käyttöliittymien toteutukseen ja niiden väliseen viestintään monitasoarkkitehtuuriratkaisussa löytyy runsaasti eri teknologiavaihtoehtoja. Vaihtoehdoista tällä hetkellä suosituimpina voidaan pitää Java:an ja CORBA:an (Common Object Request Broker Architecture) pohjautuvaa ratkaisua, Enterprise JavaBeans –teknologiaa (EJB), sekä Microsoftin WinDNA (Windows Distributed interNet Architecture) -teknologiaa. Lisäksi Microsoft on julkaisemassa uuden .NET-arkkitehtuurin, jota tässä tutkimuksessa tarkastelemme kappaleessa 4 vielä laajemmin.

Kaikkia Java-teknologiaan pohjautuvia ratkaisuja voidaan pitää alustariippumattomina. Javalla tehdyt sovellukset toimivat periaatteessa kaikissa ympäristöissä, joissa on tarvittavat Java Virtual Machine (JVM) –suoritusympäristöt. Toteutettaessa hajautettuja sovelluksia monitasoarkkitehtuuriin Java:lla, tai tarkemmin J2EE:llä (Java 2 Enterprise Edition), valittavana on lähinnä kaksi vaihtoehtoa. Joko toteutetaan sovellukset Java:lla käyttäen hajauttamiseen CORBA:a, tai käytetään Enterprise Java Beans:ja, jotka muistuttavat Microsoftin COM+ -teknologiaa. Java:a ja CORBA:a käyttäen sovellukset ovat EJB-teknologiaa yksinkertaisempia rakentaa [Gra99], mutta EJB sisältää useita hajautetuissa sovelluksissa tarvittavia ominaisuuksia. EJB:kin käyttää CORBA:a hajautusteknologiana, mutta EJB sisältää myös mm. tapahtumienhallintaan ja tietoturvaan tarvittavat tekniikat.

Vaikka Java-teknologiat ovatkin suosittuja ja monelta osin hyviksi havaittuja, tässä tutkimuksessa emme tarkastele niitä sen enempää. J2EE:llä ja Microsoftin .NET-teknologialla on paljon yhteistä [Far00] ja tässä tutkimuksessa keskitymme .NET-teknologiaan, jota voidaan pitää uusimpana hajautettujen sovellusten rakentamiseen kehitettynä teknologiana. Ennen tarkempaa .NET-teknologian käsittelyä käymme kuitenkin läpi WinDNA (Windows Distributed interNet Architecture) teknologiaa, mistä .NET:kin on osaltaan saanut alkunsa.

WinDNA perustuu komponenttipohjaisiin teknologioihin Windows-alustalla. WinDNA koostuu suuresta määrästä eri osa-alueita, mutta pääasiassa se on yhdistelmä Microsoftin COM+ - ja Windows 2000 teknologiaa. COM+ -teknologia (kts. Kappale 4.2.7) taas koostuu Microsoftin COM (Component Object Model) –arkkitehtuurista, DCOM (Distributed Component Object Model) -tekniikasta ja MTS (Microsoft Transaction Server) –palvelinmallista.

COM-tekniikka mahdollistaa erillisten komponenttien käytön sovelluksessa. Komponentit voivat olla itse valmistettuja tai esim. ulkopuolelta ostettuja. COM-tekniikan jälkeen Microsoft on kehittänyt DCOM-tekniikan, joka mahdollistaa etäkutsut toisilla palvelimilla sijaitseviin COM-komponentteihin. MTS-palvelimet mahdollistavat mm. tapahtumienhallinnan sekä tietokantayhteyksien hallinnoimisen kutsuttaessa COM-komponentteja DCOM-etäkutsujen avulla.

WinDNA siis perustuu suurelta osin COM-komponenttien hyödyntämiseen. COM-komponentit on suunniteltu alunperin siten, että sovellus on koko ajan yhteydessä komponenttiin, joko DCOM:n avulla tai ilman. Vaikka COM-komponentteja pystyy käyttämään tilattomallakin (kts. kappale 3.1) yhteydellä, niitä ei ole varsinaisesti suunniteltu siihen. Microsoftin uusin .NET-teknologia taas suosii enemmän ns. tilattomia yhteyksiä sovellusten välillä, joka on suurilla käyttäjämäärillä osoittautunut tehokkaammaksi tavaksi.

Kun projektia suunnitellessa mietitään, mitä tekniikoita sen toteuttamisessa tulisi käyttää, kannattaa huomioida ainakin seuraavia asioita. Täysin uutta projektia monitasoarkkitehtuuriin luotaessa voidaan vaihtoehtoina pitää ainakin Java-pohjaisia (J2EE) ratkaisuja tai EJB-teknologiaa, sekä Microsoftin COM-komponentteihin perustuvaa Windows DNA -teknologiaa, tai .NET-teknologiaan pohjautuvaa ratkaisua.

Valittaessa Microsoftin teknologioiden välillä, pitää huomioida, että .NET on vielä varhaisessa vaiheessa ja sen sisältö saattaa muuttua. Toisaalta Windows DNA –teknologialla valmistettu komponenttipohjainen sovellus on mahdollista siirtää myöhemmin .NET-ympäristöön, esim. rakentamalla COM-komponenteista .NET-luokkia.

Jos lähdetään siirtämään olemassaolevia sovelluksia monitasoarkkitehtuuriin, teknologian valintaan vaikuttaa luonnollisesti se, millä siirrettävät sovellukset on rakennettu. Jos aiemmin on käytetty Microsoftin kehitysvälineitä, .NET tarjoaa monipuolisen alustan sovellusten nykyaikaistamiseksi.

4 Microsoft.NET -arkkitehtuuri

.NET on Microsoftin uusin pyrkimys siirtyä monitasoarkkitehtuuriin. .NET- arkkitehtuurin päällimmäisenä ideana on ohjelmistojen hajauttaminen toimimaan verkon yli hyödyntäen ns. tilattomia palveluita (engl. stateless) ja mahdollistaen eri alustoilla toimivien sovellusten välisen kommunikoinnin [And01]. Microsoftin pääjohtaja Bill Gates on sanonut .NET:n olevan suurin uudistus sitten graafisten käyttöliittymien ja Windows95-käyttöjärjestelmän ilmestymisen.

Tässä kappaleessa tarkastellaan Microsoftin .NET -arkkitehtuurin rakennetta ja sen tarjoamia mahdollisuuksia. Kappaleessa huomioidaan erityisesti, mitä .NET tarjoaa hajautettujen monitasoarkkitehtuuristen järjestelmien toteutukseen.

4.1 Taustaa

.NET-arkkitehtuuri perustuu suurelta osin sovellusten välisen kommunikoinnin toteuttamiseen XML (Extensible Markup Language) –pohjaisilla (kts. kappale 4.3.1) SOAP (Simple Object Access Protocol) –viestellä (kts. kappale 4.3.2). .NET-arkkitehtuurin avulla voi mm. toteuttaa tilattomia SOAP-viesteillä kutsuttavia verkkopalveluita, mahdollistaen mm. eri alustoilla toimivien sovellusten kommunikoinnin keskenään.

Uusien arkkitehtuuristen mallien lisäksi .NET sisältää myös muita uudistuksia, joista aluksi mainittakoon Java-maailmasta tuttu ajonaikainen suoritusympäristö, CLR (Common Language Runtime). CLR:n luvataan mm. tekevän sovelluksista turvallisempia uusien muistinhallinta- ja roskienkeruuominaisuuksien vuoksi. Jokaiselle sovellukselle varataan oma muistialue, jolla yritetään välttää pääsyä varatuille muistialueille. Roskienkeruu hoidetaan automaattisesti, eli esim. oliot, joihin ei enää viitata, tuhotaan automaattisesti.

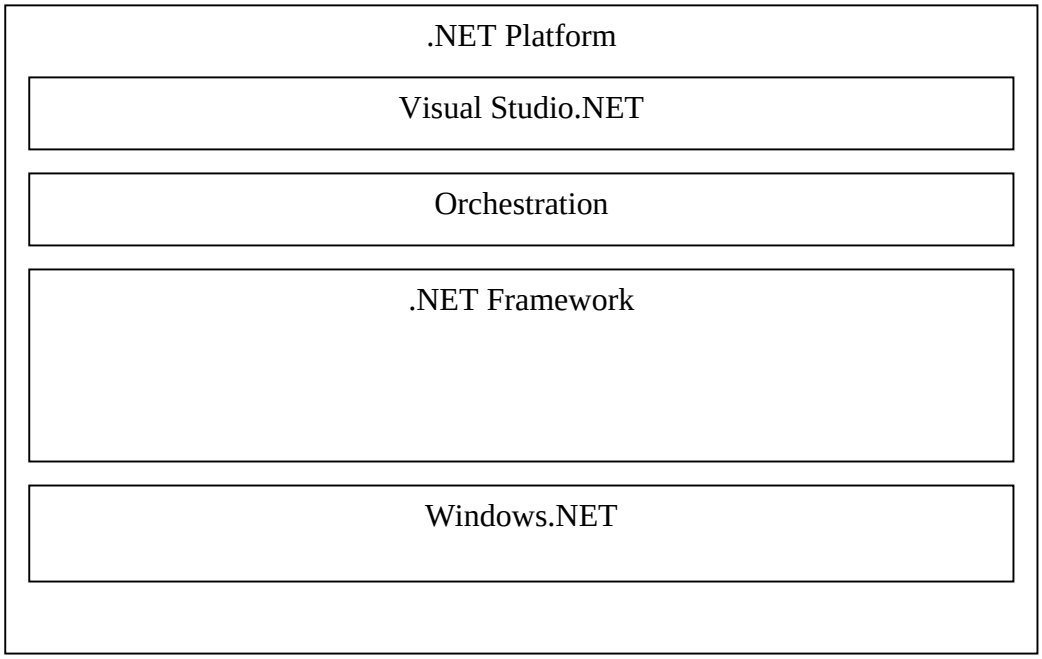
CLR on periaatteessa mahdollista rakentaa myös muille kuin Windows-alustoille, jolloin .NET-sovellukset olisivat Java-sovellusten tapaan alustariippumattomia, mutta mitään päätöksiä asiasta ei vielä ole tehty.

.NET:n luvataan myös helpottavan ohjelmistokehitystä, koska suuri osa toimintoja on rakennettu valmiiksi .NET-ympäristössä käytettäviin luokkiin. Nämä luokat osaltaan mahdollistavat erään .NET-uudistuksista, riippumattomuuden ohjelmointikielistä. .NET-ympäristössä on mahdollista rakentaa sovelluksia useilla eri ohjelmointikielillä, käyttää eri kieliä jopa saman sovelluksen sisällä, ja saada aikaan ohjelmointikielistä riippumatta saman lopputuloksen.

Lisäksi .NET:n luvataan mm. helpottavan ohjelmien asennusta, jossa ei enää tarvita rekisteröintiä tai jaettuja .dll-tiedostoja. .NET-sovellukset on mahdollista asentaa uuteen hakemistoon ns. XCOPY-toiminnolla, jolloin mitään vanhoja tiedostoja ei päivitetä. COM-komponentteja käytettäessä ne piti erikseen rekisteröidä, .NET-sovelluksissa vastaavia rekisteröintejä ei enää ole. Myöskään käyttöjärjestelmän (Windows) rekisteriin ei kirjoiteta ohjelmaan liittyviä tietoja.

4.2 .NET Platform

Platform on .NET -arkkitehtuurin alusta. Se sisältää kaikki .NET:n osa-alueet, joiden avulla .NET -sovelluksia on mahdollista rakentaa ja suorittaa. Kuvassa 4 on kuvattu .NET Platformin osa-alueet.



Kuva 4. .NET Platform

Platformin alin kerros on käyttöjärjestelmä, Windows.NET. .NET -sovellukset on mahdollista saada toimimaan ainakin Windows CE, ME, 95, 98, NT 4.0 ja 2000 -käyttöjärjestelmissä asentamalla näihin Common Language Runtime (CLR) -ympäristö, jonka päällä .NET-sovellukset toimivat. Tulevaisuudessa on mahdollista, että .NET CLR toteutetaan myös muihin kuin Microsoftin käyttöjärjestelmiin. Microsoft on myös ilmoittanut julkaisevansa uuden täysin .NET:iä tukevan Windows-käyttöjärjestelmän, nimeltään Windows XP.

Seuraava kerros on .NET Framework, joka on koko .NET -arkkitehtuurin ydin. Framework kattaa kaikki ohjelmistokehityksen alueet käyttöjärjestelmästä alkaen [Con00]. Framework-kerros voidaan jakaa pääpiirteittäin kolmeen osa-alueeseen, jossa Framework:in pohjalla on Common Language Runtime (CLR), .NET-sovellusten ajonaikainen suoritusympäristö. CLR huolehtii mm. sovelluksen kääntämisestä prosessorin ymmärtämään muotoon, natiivikoodiksi, sekä muistinhallinnasta.

CLR:n päällä toimii Framework Base Classes, joka sisältää kaikille .NET:n ohjelmointikielille yhteisiä luokkakirjastoja.

Framework:in ylin kerros sisältää mm. käyttöliittymäkomponentit, Windows Forms:it (WinForms) ja ASP.NET:n. ASP.NET koostuu selainpohjaisten käyttöliittymien rakentamiseen käytettävistä Web Forms -komponenteista, Web Services -osasta, sekä ASP.NET –palveluista (ks. kappale 4.2).

Ylin kerros .NET Platformissa on .NET-sovellusten kehitysvälineet, Visual Studio.NET. Visual Studion suurin uudistus on riippumattomuus ohjelmointikielestä. Siinä on mahdollista käyttää myös muiden kuin Microsoftin tukemia ohjelmointikieliä, jotka on integroitu .NET:iin [Con00]. .NET Type System mahdollistaa kääntäjien tekemisen lähes mille tahansa ohjelmointikielelle. Sovellus käännetään aina samalle välikielelle (MSIL), riippumatta siitä, millä ohjelmointikielellä se on tehty.

.NET Platform sisältää myös Microsoftin palvelinperheen, .NET Enterprise Servers, sekä Building Block Services ”rakennuspalikat”.

Building Block Service:ihin pohjautuu mm. palvelu jota kutsutaan nimellä “Hailstorm”. Hailstorm sisältää mm. Microsoftin uuden käyttäjäntunnistusjärjestelmän, Passport:n [Man01]. Passport:n ideana on, että käyttäjä voi tallentaa tietonsa verkossa olevaan ”passiin” ja päästä niihin käsiksi myös muualta kuin omalta koneelta. Lisäksi käyttäjän tarvitsee tunnistautua vain kerran istunnon yhteydessä Passport:iin, jolloin eri www-sivut ja muut verkossa olevat sovellukset kaikki toimisivat saman tunnistautumisen pohjalta. Passport:n tekee käyttäjän näkökulmasta arveluttavaksi ainakin seikka, että hänen tietonsa olisivat palvelimella verkossa. Tietomurtojen yhteydessä henkilökohtaiset tiedot saattaisivat joutua vääriin käsiin, olkoonkin, että käyttäjä itse valitsee, mitä tietoja hän Passport:iin antaa. Lisäksi voidaan miettiä, haluaako palveluntarjoaja tilannetta, että käyttäjän pitää aina tunnistautua Microsoftin palvelun kautta, päästäkseen hänen sivuilleen tai palveluihin.

.NET –arkkitehtuuri tarjoaa periaatteessa kaksi eri tapaa hajautettujen sovellusten rakentamiseen, Web Service –verkkopalvelut, sekä Remoting:in.

Web Services -verkkopalvelut ovat verkkoon URI- (Universal Resource Identifier) osoitteeseen rakennettu palvelu. Web Service:n kanssa kommunikointi tapahtuu XML- pohjaisina viesteinä (XML, kts. kappale 4.3.1), HTTP-protokollaa käyttäen. Palvelut voivat olla kolmannen osapuolen tekemiä esim. laskentasovelluksia. Web Service:jä käsittelemme kappaleessa 4.2.6.

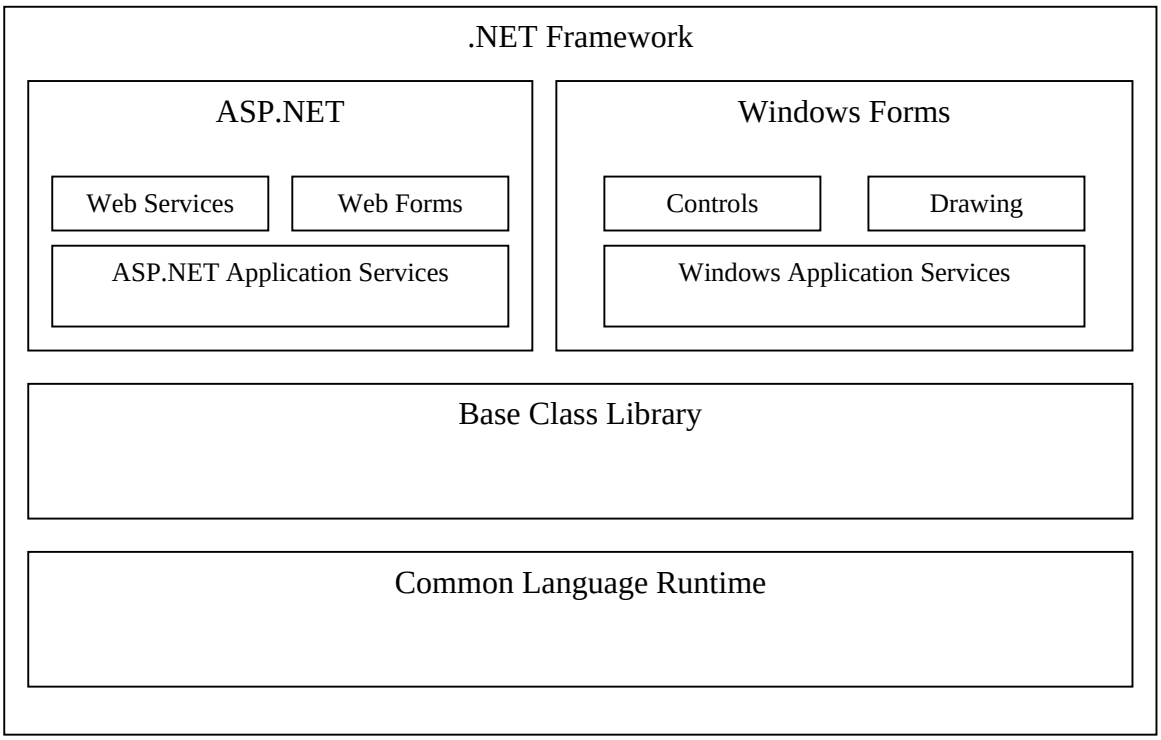
Luotaessa asiakas- ja sovelluspalvelimen välinen yhteys käyttämällä Remoting-yhteystapaa, saadaan Web Serviceä tarkemmin määriteltyä välitettävän viestin sisältö ja välitystapa. Remoting:ia tarkastelemme enemmän kappaleessa 4.3.3.

Seuraavassa luvussa tarkastelemme .NET-arkkitehtuurin ja Platformin ydintä, .NET Framework:iä.

4.3 .NET Framework

.NET Framework -ympäristö sisältää kaikki .NET-sovelluskehitykseen, julkaisemiseen ja ohjelman suorittamiseen tarvittavat osa-alueet. Kuten jo edellisessä kappaleessa mainittiin, .NET-kehys voidaan jakaa pääpiirteittäin kolmeen eri osa-alueeseen, Common Language Runtime:en (CLR), Framework Base Classes:in, sekä käyttöliittymäkomponentit sisältävään kerrokseen (kuva 5).

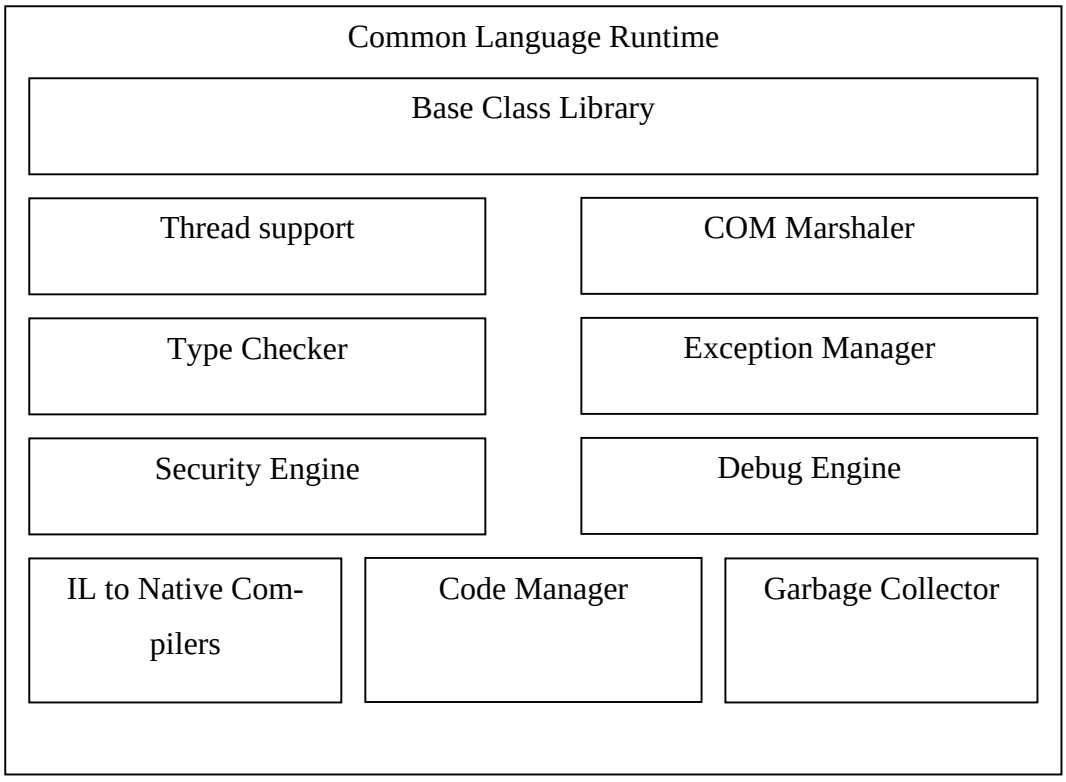
Framework-ympäristöön on koottu useita komponentteja, jotka aikaisemmin ovat olleet ohjelmointikielikohtaisia. Ohjelmointikielen osuus on näin yritetty supistaa mahdollisimman pieneen, lähinnä kielikohtaiseen syntaksiin. Seuraavassa käsittelemme tarkemmin Frameworkin osa-alueita.



Kuva 5 .NET Framework

4.3.1 Common Language Runtime (CLR)

.NET Frameworkin alimpana kerroksena on Common Language Runtime (CLR), ajonaikainen suoritusympäristö. Kuvassa 6 on esitelty CLR:n osa-alueet. CLR on osittain tuttu Java-teknogian puolelta, jossa ohjelma suoritetaan Java Virtual Machinen (JVM) avulla. CLR toimii vastaavasti .NET:ssä, kääntäen sovelluksen Microsoft Intermediate Languagesta (MSIL) suoritettavaan muotoon. Erona näiden kahden kääntäjän välillä on kuitenkin se, että Java-sovellusta ajettaessa JVM voi toimia ikäänkuin tulkkina koodin ja prosessorin välillä, kun taas CLR kääntää aina välikielen natiivikoodiksi ennen suoritusta. CLR:n luvataan myös osaavan optimoida koodi eri prosessorityypeille. Vaikka CLR kääntää koodin ennen suoritusta, eivät sen tehtävät kuitenkaan rajoitu pelkkään koodin kääntämiseen. Common Language Runtimellä on runsaasti myös muita tehtäviä, joista eräs tärkeimpiä on muistinhallinta ja roskienkeruu (garbage collection).



Kuva 6. Common Language Runtime

Roskienkeruu poistaa automaattisesti oliot, joihin ei enää viitata. Roskienkeruu ja muistinhallinta minimoivat mahdollisia sovelluskehittäjän tekemiä virheitä muistin varaamisessa ja vapauttamisessa, tehden näin ohjelmista hieman turvallisempia suorittaa. Varsinkin C++ -ohjelmointikieleen tämä on tervetullut uudistus. Käyttäjä tosin pystyy C++:aa kirjoittaessaan edelleenkin itse hoitamaan muistinvaraukset yms., mutta oletuksena CLR pitää niistä huolen. CLR käyttää myös ns. Application memory isolation -toimintoa, joka estää sovellusta pääsemästä varatuille muistialueille ja niin aiheuttamaan toimintahäiriöitä.

Common Language Runtime ei ole rajattu toimimaan ainoastaan Microsoftin käyttöjärjestelmissä, vaan sen voisi periaatteessa rakentaa lähes mille alustalle tahansa. Microsoft ei kuitenkaan ole tehnyt päätöstä CLR:n siirtämisestä muihin ympäristöihin.

CLR:n ensisijainen tavoite on ohjelmistokehityksen helpottaminen, ei suorituskyvyn parantaminen [Con00]. Common Language Runtime:n ja sen sisältämän Common Type System:in avulla eri ohjelmointikielillä tehdyt sovelluksen osat ymmärtävät toisiaan [Ric00b]. On mahdollista rakentaa luokka esimerkiksi C++:lla tai vaikkapa Microsoftin kehittämällä uudella C#:lla ja periä luokka tai luoda sen ilmentymä Visual Basic -koodissa. Common Type System sisältää ohjelmointikielille yhteiset tietotyypit [Con00, s.38].

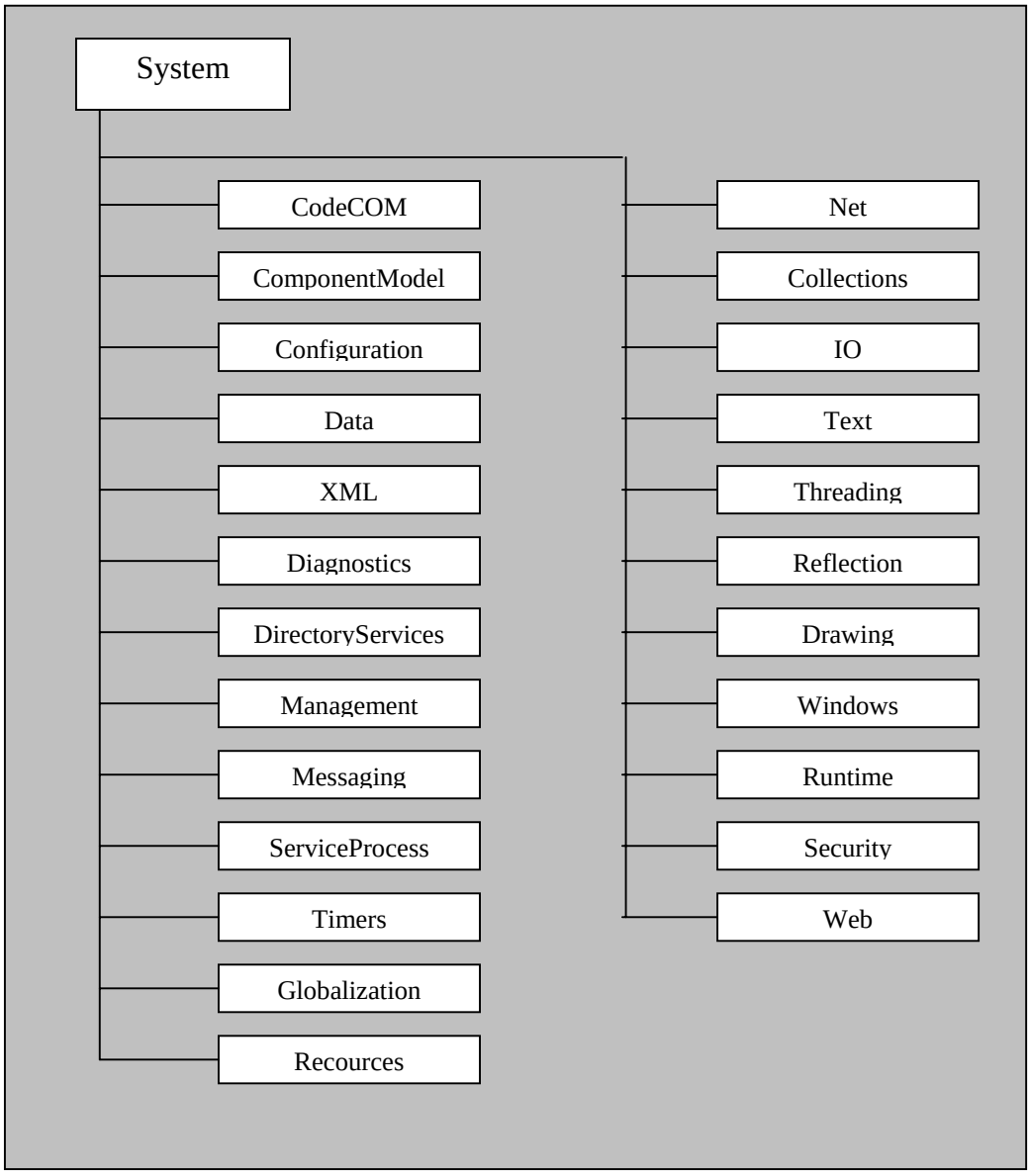
CLR sisältää myös koodin oikeellisuuden tarkastuksen, sekä tietotyyppitarkistukset, jolla varmistetaan, ettei koodi sisällä vääriä tyyppimuunnoksia tai alustamattomia muuttujia.

Ohjelmakoodia, jonka CLR kääntää suoritettavaan muotoon kutsutaan myös nimellä "managed code". Visual Studio.NET:llä (kts.luku 4.2.2) voidaan kirjoittaa myös koodia, jota CLR:n ei enää tarvitse kääntää. Koodia kutsutaan "unmanaged code:ksi" ja sitä pystyy kirjoittamaan ainoastaan C++ -ohjelmointikielellä [Ric00a].

Common Type System on osa Common Language Runtimea. Se yhdistää eri ohjelmointikielten tietotyypit ja täten mahdollistaa mm. kielten välisen perinnän.

4.3.2 Yleiset luokkakirjastot

.NET:n yleiset luokkakirjastot, Framework Classes (kts. kuva 5), sisältävät lukuisia kaikkien ohjelmointikielten käytettävissä olevia luokkia ja metodeja. Suuri osa entisistä ohjelmointikielten ominaisuuksista on nyt Framework Classes -luokissa [Con00, s.20]. Framework Classes sisältää abstrakteja luokkia, joita voidaan periä sovelluksessa, sekä luokkia, joista voi luoda ilmentymän, olion. Luokat on nimetty tietyn nimiavaruuden (engl. namespaces) mukaan [Con00, s.171]. Nimiavaruuden juurena (engl. root) on *system*-nimiavaruus, jonka alapuolelta löytyy 25 seuraavaa nimiavaruutta (kuva 7), joista kukin sisältää vielä useita alempia nimiavaruuksia.



Kuva 7 .NET nimiavaruus

Esim. *System.IO* sisältää tiedostojen ja tietovirtojen ja *System.XML* XML-dokumenttien kirjoittamiseen, lukemiseen ja rakentamiseen tarvittavat luokat. *System.Threading* nimiavaruus sisältää säikeiden käyttämiseen tarvittavat luokat jne.

4.3.3 Windows-käyttöliittymät

Frameworkin ylin kerros sisältää mm. käyttöliittymien rakentamiseen vaadittavat komponentit. .NET:ssä on periaatteessa kaksi tapaa rakentaa sovellukselle käyttöliittymä. On mahdollista tehdä Windows-sovellus Windows Forms:ien (lyhennetään WinForms) avulla, tai sovellukselle voidaan rakentaa selainpohjainen käyttöliittymä WebForms:eja apuna käyttäen. Sekä WinForms että WebForms ovat yhteisten luokkakirjastojen ja tietotyyppien tapaan myös kaikkien .NET-ohjelmointikielten käytettävissä.

.NET Framework mahdollistaa myös täysin tekstipohjaisten käyttöliittymien rakentamisen, mutta tässä tutkielmassa tutustumme vain edellä mainittuihin graafisiin käyttöliittymiin. Tässä kappaleessa tarkastelemme tapoja rakentaa Windows-käyttöliittymä.

WinForms perustuu Microsoftin Java-kehitysvälinettä (kts. Java 3.4) J++:aa varten kehitettyihin Windows Foundation Classes –luokkiin (WFC) [Con00, s.21]. .NET-arkkitehtuurissa WinForms on osa Framework Classes -luokkia. Esim. lomake luodaan perimällä System.Windows.Forms -luokka.

Windows Form -lomake on mahdollista myös periä edelleen, jolloin voidaan esim. luoda pohja, jota käytetään useammalla lomakkeella.

Rakennamme esimerkkinä Windows-sovelluksen, jonka luokkakaavion rakensimme kappaleessa 2.2. Sovelluksella haetaan arvopaperin perustietoja tietokannasta arvopaperin tunnuksen perusteella.

Esimerkkisovelluksessa painettaessa lomakkeen *Hae tiedot*-painiketta (Kuva 8) luodaan c*Arvopaperit* –luokan olio. Arvopaperit –olio taas luo tarvittavan määrän cArvopaperi –luokan olioita, tässä esimerkissä yhden. Arvopaperi –olio hakee tarvittavat tiedot tietokannasta *cDatabase* -luokan avulla, sille parametrinä viedyn tunnuksen perusteella.

Kuva 8. WinForms-käyttöliittymä

Tietojen hakemisen tietokannasta voisi periaatteessa hoitaa suoraan käyttöliittymäluokan kautta, jolloin olisi pärjätty kokonaan ilman cArvopaperit- ja cArvopaperi –luokkia. Tässä esimerkissä yritämme kuitenkin rakentaa helposti uudelleenkäytettävää ohjelmakoodia, joka voidaan liittää eri ympäristöihin. Sovelluksen ohjelmakoodi on liitteissä 8.1 (cArvopaperi), 8.2 (cArvopaperit), 8.3 (cDatabase) ja 8.4 (WinForm.vb).

Tässä esimerkissä olemme rakentaneet sovelluksen, jossa on sovelluslogiikka eriytetty kokonaan käyttöliittymästä. Sovellus toimii sellaisenaan asiakas/palvelin -ympäristössä, mutta on helppo siirtää toimimaan myös hajautettuna sovelluksena useammalla koneella. Seuraavissa kappaleissa rakennamme esimerkit, joissa samaa sovelluslogiikkaa käytetään sellaisenaan. Ensin liitämme sovelluslogiikan toimimaan www-sivun kautta (kappale 4.2.5), jonka jälkeen rakennamme siitä Web Service:n (kappale 4.2.6).

4.3.4 www-käyttöliittymät

.NET-sovelluksiin voidaan rakentaa käyttöliittymäksi myös ASP.NET WebForm, selainpohjainen käyttöliittymä. ASP.NET on uudistettu Active Server Pages (ASP), Microsoftin dynaamisten www-sivujen rakentamiseen kehittämä teknologia. ASP.NET sivujen rakentaminen Visual Studio .NET:llä (kts. kappale 4.5.2) on hyvin samanlaista kuin Windows-käyttöliittymien. Ohjelmoijan ei välttämättä tarvitse osata HTML:ää, vaan Visual Studio .NET generoi HTML-koodin automaattisesti.

ASP.NET -sivuilla on mahdollista erottaa ohjelmakoodi omaksi tiedostokseen, jolloin itse käyttöliittymä-sivun (tiedostopääte .aspx) muokkaaminen on helpompaa. Rakennettaessa aspx-sivua Visual Studio.NET:n avulla (kts. kappale 4.4.2), ohjelmakoodi tehdään oletuksena erilliseen tiedostoon, aspx-sivulle tulee vain sivun muotoilu ja kontrollit HTML:nä (Hypertext Markup Language). Kontrollin takana oleva koodi ei ole aspx-sivulla, vaan siellä on viittaus, mistä ohjelmakoodi löytyy. Esimerkkinä painikkeen lisäämiseen vaadittu ohjelmakoodi aspx-sivulla.

```
<asp:Button id=Button1 runat="server" Width="112" Height="24" Text="Hae
            tiedot"></asp:Button>
```

ASP.NET -sivujen taustalla oleva ohjelmakoodi voi olla tehty millä tahansa .NET:n ohjelmointikielellä. Ohjelmakoodi voidaan joko kääntää kokonaan ennen suoritusta, tai käyttää ajonaikaista kääntäjää (JIT). Pääsääntöisesti koodi suoritetaan palvelimella.

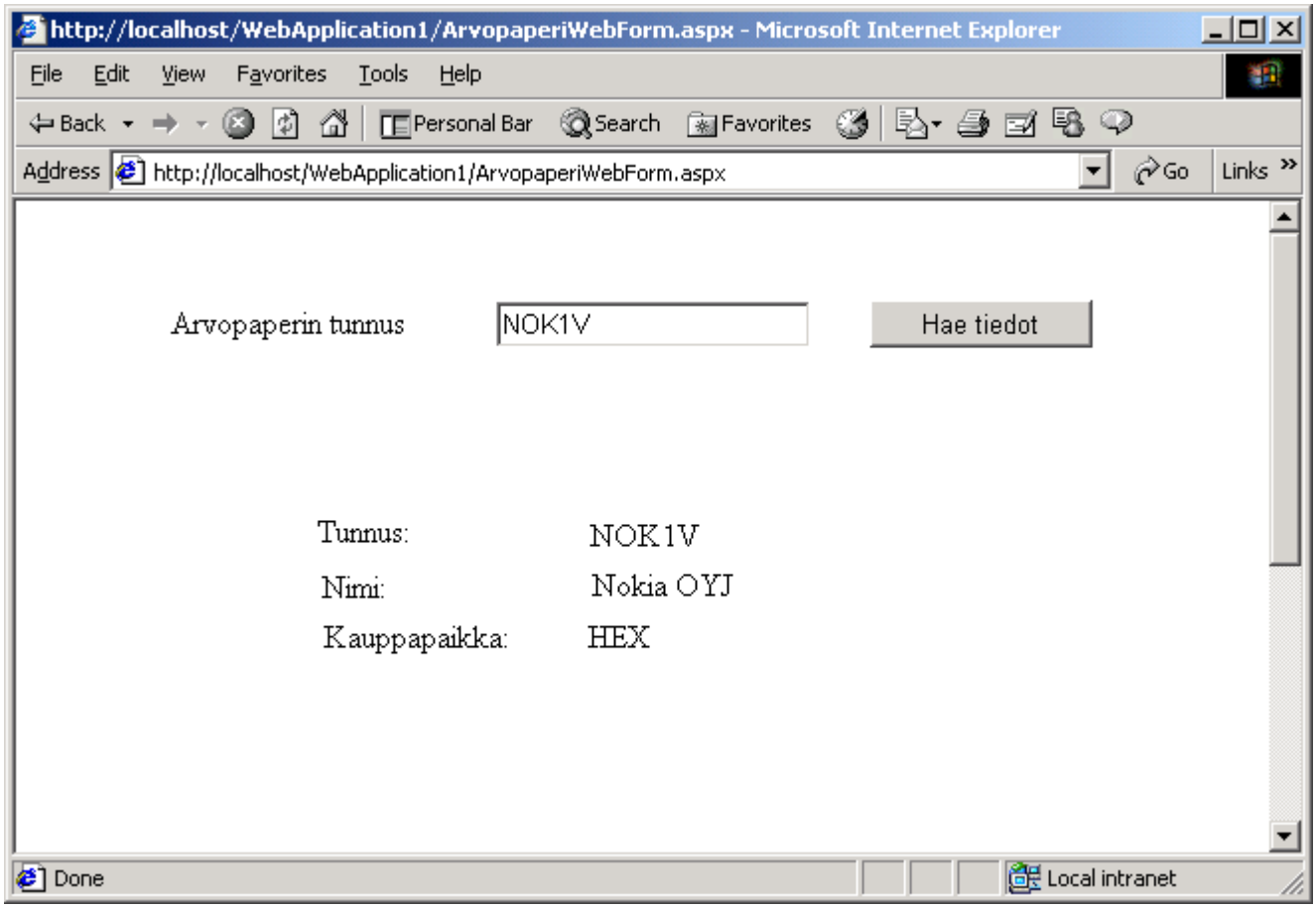
Esimerkkinä rakennamme edellisen kappaleen sovelluslogiikkaan, vastaavat kontrollit kuin WinForms-käyttöliittymässä (4.2.4) sisältävän, ASP.NET-käyttöliittymän. Käytämme *cArvopaperi-*, *cArvopaperit* ja *cDatabase* –luokkia sellaisenaan, joiden lisäksi teemme yhden .aspx –sivun (ASP.NET sivu, jota kutsutaan selaimella), joka toimii käyttöliittymänä, sekä luokan, jossa sijaitsee käyttöliittymän toimintalogiikka. Käyttöliittymän toimintalogiikan sisältävä luokka on omana tiedostonaan ja se sisältää lähes identtistä koodia, kuin WinForms-käyttöliittymä, poikkeuksena web -komponenttien käsittely WinForms-komponenttien sijaan. Esimerkkinä ohjelmakoodia, joka käsittelee ohjelman toiminnot käyttäjän painettua *Hae tiedot* –painiketta.

```
‘Web Form:in painikkeen käsittelyn hoitava ohjelmakoodia(cArvopaperi)
   Public Sub Button1_Click(ByVal sender As Object, ByVal e As _
   System.EventArgs)
        Dim A As New cArvopaperi()
        Dim sText As String
        Dim APt As New cArvopaperit()
        Dim Result As New System.Collections.ArrayList()

        APt.LataaArvopaperi(textbox1.Text)
        Result = APt.Arvopaperit
            ‘Tässä tulostetaan saadut tiedot…
```

Kun tekstikenttään nyt asetetaan arvopaperin tunnus ja painetaan hae tiedot-painiketta, luodaan *cArvopaperit*-luokan ilmentymä (olio) ja kutsutaan sen LataaArvopaperi –metodia, jolle viedään tunnus parametrinä.

Tuloksena saadaan kuvan 9 kaltainen www-sivu.



Kuva 9. Arvopaperi Web sovellus

Uutena ominaisuutena ASP.NET-sivuissa on vanhoihin asp-sivuihin verrattuna mm. ominaisuus, että tekstikenttien arvot säilyvät, vaikka sivulla oleva mahdollinen tuloskenttä päivittyisikin. Päivitetään tuloskenttä pyytämällä Arvopaperit-oliolta arvopaperin tiedot ja tulostamalla ne.

Paitsi että ASP.NET -sivut muistuttavat ulkonäöltään aiempia ASP-sivuja enemmän Win32-sovelluksia, myös niiden rakentaminen muistuttaa enemmän Windows-sovellusten rakentamista. Visual Studio.NET tukee myös ASP.NET -sivujen rakennukssessa ns. drag'n'drop -menetelmää, jossa kontrollit ”raahataan” lomakkeelle.

4.3.5 Web Services

Web Services on Microsoftin ja IBM:n yhteistyössä kehittämä ”verkkopalvelu”, jossa verkossa olevaa ohjelmaa kutsutaan HTTP-protokollan kautta. Web Service:n käyttö ei ole rajattu ainoastaan .NET-sovelluksille, vaan SOAP-protokollan (kts. kappale 4.3.2) avulla se on käytettävissä myös muissa ympäristöissä [How01].

Web Services toimii hajaustusteknologiana, jolla sovellukset voidaan hajauttaa toimimaan muuallakin kuin lähiverkossa [Con00, s.246] ja se onkin tärkeä osa .NET-arkkitehtuuria. Web Service:jä on mahdollista toteuttaa itse, esim. yrityksen sisäiseen käyttöön, tarjota palveluja muille, tai käyttää valmiiksi rakennettuja palveluja.

Web Service:t kuvataan Web Services Description Languagen (WDSL) avulla. WDSL on XML-kuvaus tarjottavista rajapinnoista, sisältäen kuvaukset mm. funktioista, tietotyypeistä jne.

UDDI, Universal Description, Discovery and Integration, on eräänlainen hakemisto tai hakukone, jossa tarjotaan Web Service -palveluita. Web Service:n etsiminen UDDI:sta ei kuitenkaan onnistu suoraan esim. hakusanoilla, vaan Web Service:jä haetaan SOAP-viestien avulla [Lef01]. Nähtäväksi jää, kuinka suosituiksi UDDI:n kautta julkaistut Web Service:t tulevat. Kolmannen osapuolen valmistamien Web Service:ien liittämistä osaksi omaa sovellusta saatetaan pitää suurempana riskinä, kuin esim. valmiiden komponenttien ostamista omaan käyttöön.

Web Service:jä voidaan käyttää sekä www-sivuilta, että asiakaskoneessa toimivasta sovelluksesta käsin.

Web Service on käytännössä .asmx-päätteinen tiedosto (vrt. aspx kappaleessa 4.2.5). Seuraavassa esimerkissä rakennamme kappaleessa 4.2.4 tekemästämme Windows-sovelluksen sovelluslogiikasta Web Servicen.

Voisimme lisätä suoraan *cArvopaperit* –luokkaan määrittelyn, joka kertoo, että kyseessä on Web Service, sekä määritellä metodit, joita Web Servicestä halutaan ”julkaista”, Web metodeiksi. Lisäksi luokka tulisi peritä *System.WebService* luokasta. Luokka määritellään Web Service:ksi ”tagilla”:

```
<%@ WebService Language=”VB” Class=”cArvopaperi” &>
```

Lisäksi metodit määritellään verkosta kutsuttaviksi ”tagilla”:

```
<WebMethod()>
```

Luokan muuttaminen Web Service:ksi aiheuttaisi kuitenkin muutoksia ohjelmakoodiin, jolloin uudelleenkäytettävyys heikentyisi. Lisäksi luokka tarvitsisi uusia metodeja, jotta metodien kutsuminen tilattoman (kts. kappale 3.1) yhteyden kautta olisi mahdollista.

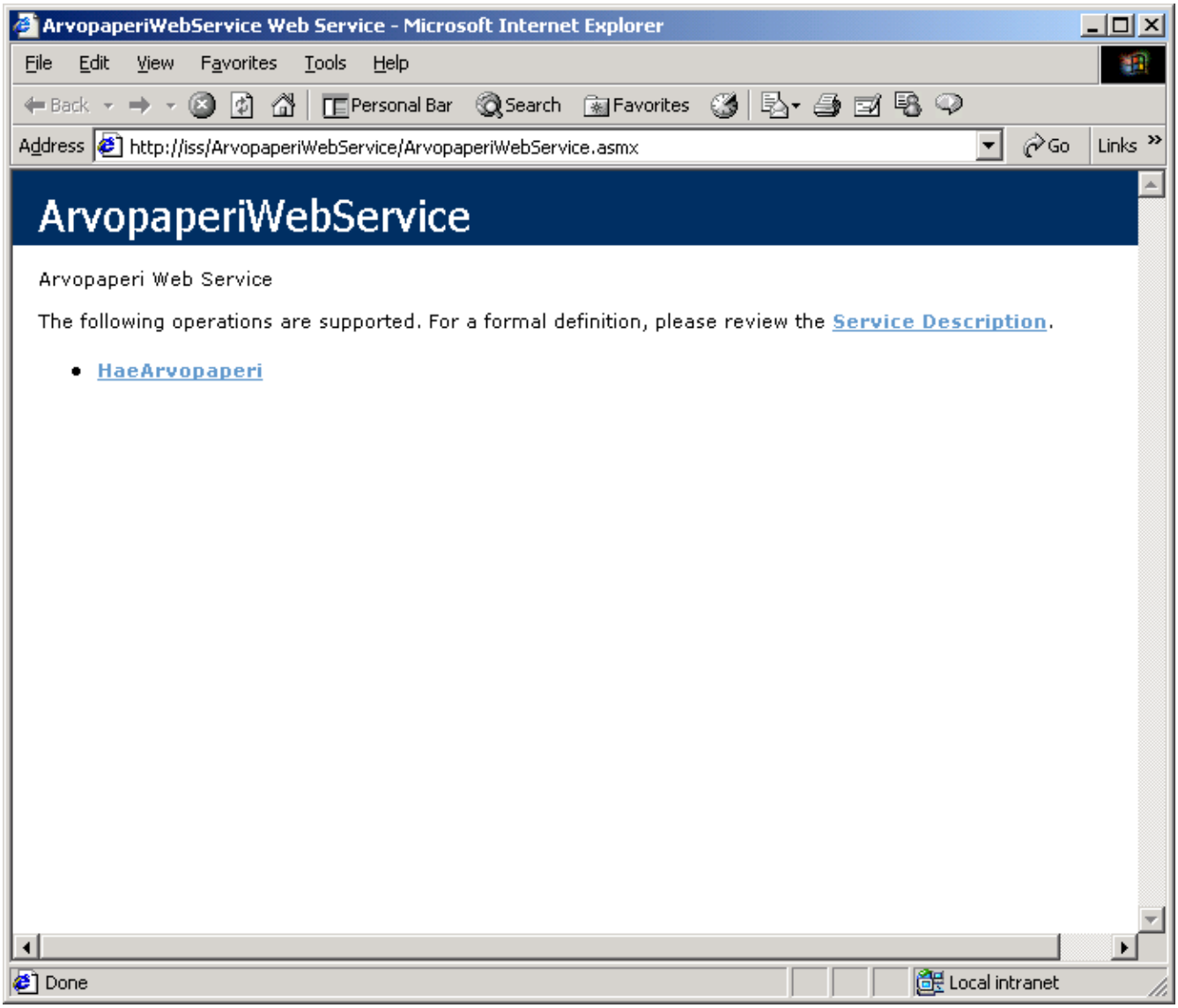
Jotta voisimme käyttää rakentamiamme luokkia sellaisenaan, muuttamatta niiden ohjelmakoodia, rakennamme eräänlaisen rajapinnan, jonka kautta luokkia ja niiden metodeja käytetään. Näin saamme rakennettua sovelluksesta myös tilattoman (kts. Kappale 3.2). Rakentamamme rajapinnan, eli itse Web Servicen, verkon yli kutsuttavan metodin ohjelmakoodi “esitelty seuraavassa”:

```
Public Function <WebMethod()> HaeArvopaperi(ByVal sTunnus As String) As _
cArvopaperi
        Dim AP As cArvopaperi
        AP.HaeTiedot(sTunnus)
        HaeArvopaperi = AP
End Function
```

Web Servicen ohjelmakoodi on liitteessä 8.6.

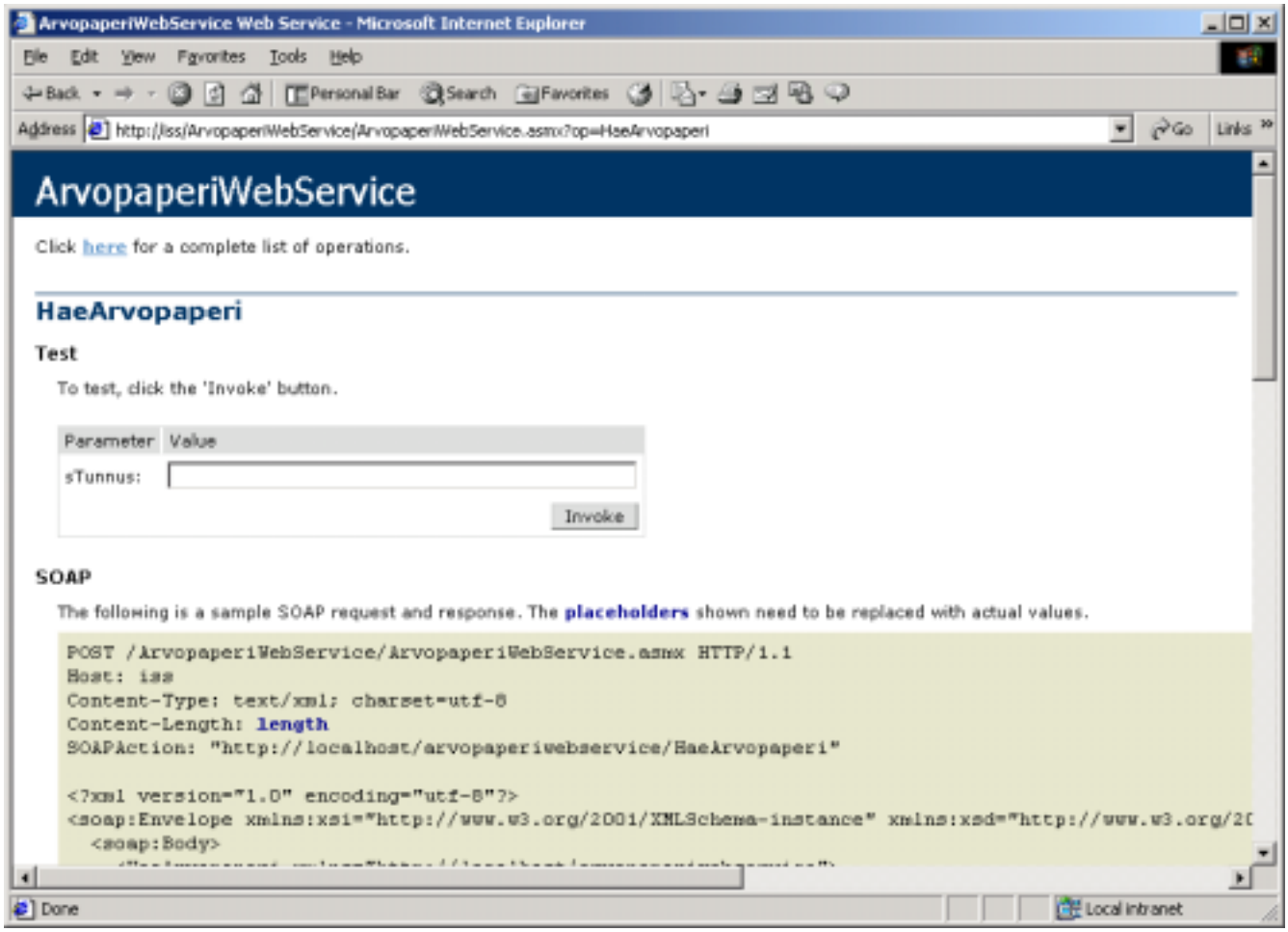
Rakentamassamme Web Servicessä on nyt yksi kutsuttava metodi, nimeltään HaeArvopaperi, jolle viedään arvopaperin tunnus parametrinä.

Tarkasteltaessa *.asmx tiedostoa www-selaimella, nähdään ASP.NET:n automaattisesti muodostama sivu [How01], jossa kerrotaan kutsuttavissa olevat Web Service:t (kts. kuva 10).



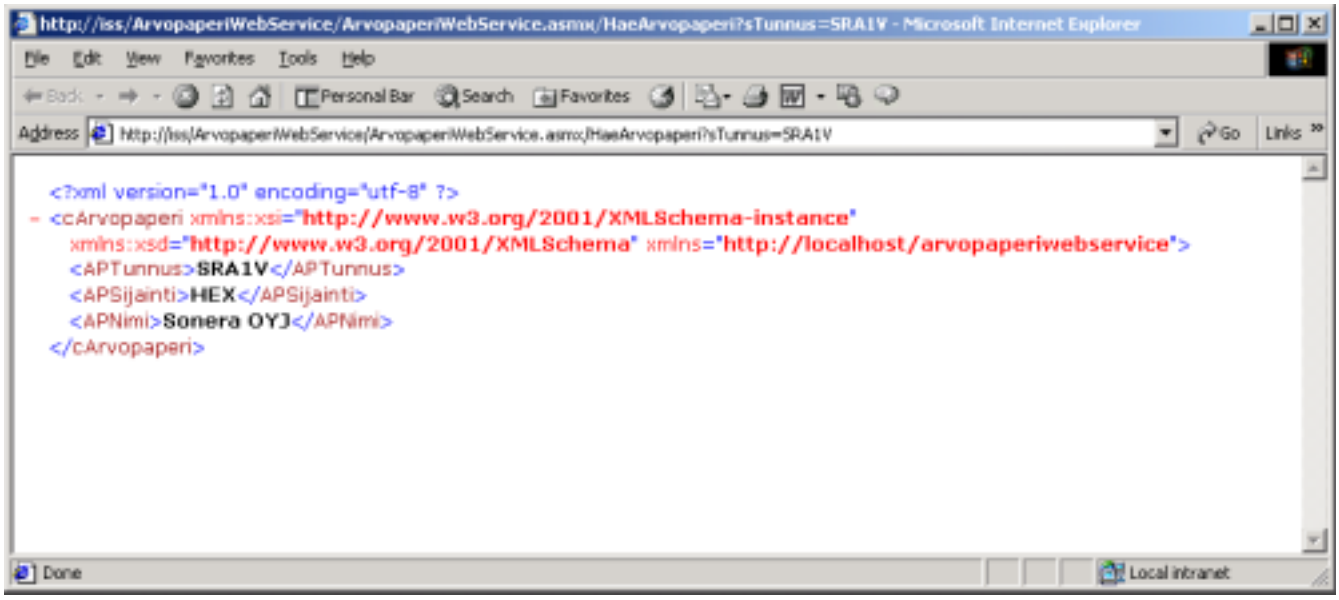
Kuva 10. Web Service:n .asmx-sivu www-selaimella

Seuraamalla tietyn Web Service:n linkkiä, nähdään sen parametrit, palautusarvot, sekä mahdolliset eri kutsutavat. Lisäksi Web Service:ä voidaan myös testata ko. sivun kautta (kts. kuva 11).



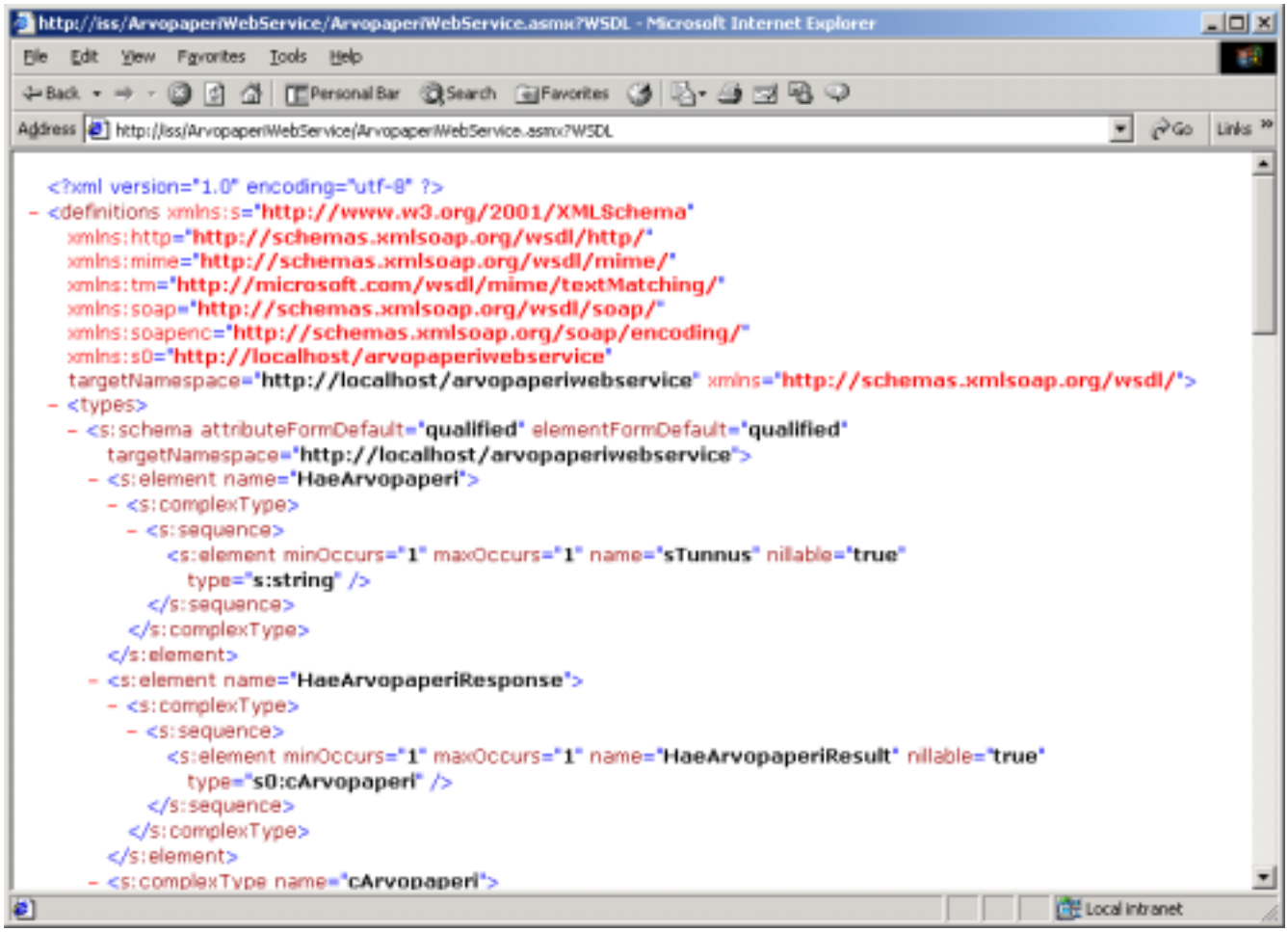
Kuva 11. Web Service:n testisivu

Testattaessa Web Service:ä www-selaimen kautta, saamme tuloksena XML-viestin, jossa on kutsumamme metodin palauttama arvo, tässä tapauksessa yhden arvopaperin perustiedot, kuva 12.



Kuva 12. Web metodin palauttama XML -viesti

Asmx –sivun kautta on mahdollista nähdä myös Web Servicen kuvaus SOAP / XML muodossa. Kuvassa 13 näemme Web Service:n palauttaman *HaeArvopaperi* metodin kuvauksen WSDL (Web Service Description Language) –muodossa.



Kuva 13. Web Servicen kuvaus.

Kuvan 13 Web Service Description Language (WSDL) -viesti on kokonaisuudessaan liitteessä 8.7.

Asmx-sivu, sekä selaimella näkyvät SOAP- ja XML-viestit ovat lähinnä Web Servce:n testaamista yms. varten. Käytännössä Web Service:n kutsut on upotettu ohjelmakoodiin ja Web Servicen palauttama viesti tulkitaan ohjelmallisesti.

Palataan vielä kappaleessa 4.2.4. rakentamaamme WinForms-esimerkkiin. Voimme liittää siihen tässä kappaleessa rakentamamme Web Servicen, jolloin Web Serviceä voidaan sovelluksesta käsin käyttää kuten paikallista metodia. .NET SDK (kts. kappale 4.4.1) sisältää työkalut, jolla Web Service:n WSDL-kuvauksesta voidaan luoda ns. proxy-luokka [Bal00]. Proxy-luokka sisältää kaikki Web Servicen kutsuttavat metodit ja luokkaa käytetään aivan kuin kaikki metodit olisivat paikallisia. Metodit eivät kuitenkaan sisällä varsinaista sovelluslogiikkaa, vaan ainoastaan kutsut Web Servicen verkko-osoitteeseen. Alla näemme osan Arvopaperi Web Servicen proxy-luokasta. Ohjelmakoodi on kokonaisuudessaan liitteessä 8.8.

```
RequestNamespace:="http://localhost/arvopaperiwebservice",     ResponseName-
space:="http://localhost/arvopaperiwebservice",
Use:=System.Web.Services.Description.SoapBindingUse.Literal,      Parameter-
Style:=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)>  _
        Public Function HaeArvopaperi(ByVal sTunnus As String) As _ cAr-
        vopaperi
            Dim results() As Object = Me.Invoke("HaeArvopaperi", New _  Ob-
            ject() {sTunnus})
            Return CType(results(0),cArvopaperi)
        End Function
```

Web Service:ä voidaan kutsua HTTP:n avulla kolmella eri tavalla. Mahdolliset tavat ovat HTTP GET, HTTP POST ja Simple Object Access Protocol (SOAP) [Pla01c].

HTTP GET:n avulla kutsuttaessa metodin kutsu näkyy suoraan URL-osoitteessa, esim.

http://localhost/ArvopaperiWebService/Service1.asmx/HaeArvopaperi?sTunnus=SRA1V

HTTP POST lähettää tarvittavat parametrit HTTP-sanoman "Content"-osuudessa, jotka ASP.NET osaa purkaa ja kutsua haluttua metodia.

SOAP-viesti lähetetään suoraan Web Service:n osoitteeseen HTTP POST-metodilla. SOAP-viesti sisältää kaikki vaaditut metodien kutsut ja parametrit (SOAP, kts. Kappale 4.3.2), jolloin niitä ei kirjoiteta esim. URL–osoitteeseen.

4.3.6 COM+

COM (Component Object Model) on Microsoftin vuonna 90-luvun puolivälissä kehittämä malli komponenttipohjaisten järjestelmien toteuttamiseen. Se mahdollistaa eri valmistajien komponenttien käyttämisen samassa sovelluksessa.

Distributed Component Object Model (DCOM) protokolla mahdollistaa COM-komponentin kutsumisen etäkoneelta.

Microsoft Transaction Server (MTS) on ajonaikainen ympäristö palvelinkoneelle, jossa ajetaan DCOM:in avulla kutsuttuja COM komponentteja [Pat00]. MTS hallinnoi mm. komponenttien tarvitsemia yhteyksiä tietokantaan, joka osaltaan mahdollistaa skaalautuvuuden useammalle käyttäjälle. MTS vastaa myös tapahtumankäsittelystä siten, että esim. virheen sattuessa koko tapahtumaketju puretaan, jolloin suoritus ei jää puolitiehen. MTS:n kaikkine ominaisuuksineen on tarkoitus mahdollistaa sovelluspalvelimen tehokas käyttö suurelle määrälle asiakkaita.

COM:n ja MTS:n yhteenliittämisen tuloksena syntyi COM+. COM+ on osa Windows 2000 -käyttöjärjestelmää ja myös osa .NET-arkkitehtuuria. Jo olemassaolevat COM- komponentit voidaan tuoda .NET-luokiksi ja vastaavasti mitä tahansa .NET-luokkaa voidaan käyttää kuten COM-komponenttia.

4.4 Kommunikointitavat

Eräs merkittävimmistä teknologioista .NET –arkkitehtuurissa on Extensible Markup Language (XML). XML:ään pohjautuu myös internet-sovellusten tiedonsiirtoa varten kehitetty Simple Object Access Protocol (SOAP).

4.4.1 Extensible Markup Language (XML)

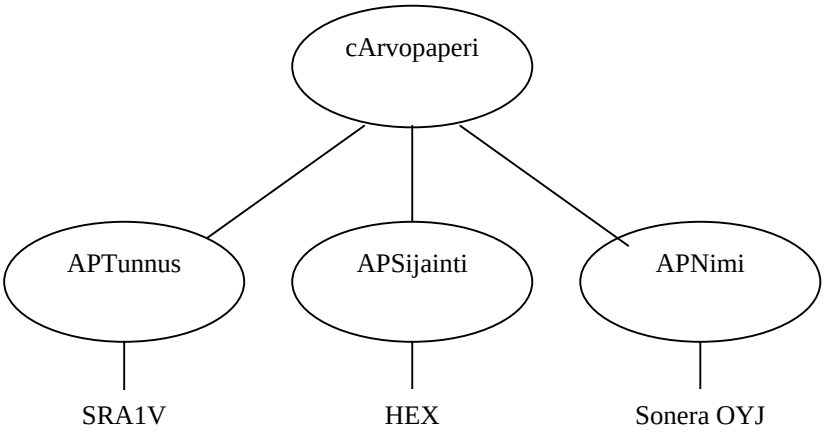
Extensible Markup Language (XML), on World Wide Web Consortiumin (W3C) alunperin lähinnä verkkokäyttöön kehittämä rakenteisen tiedon kuvauskieli, joka on muodostettu Standard Generalized Markup Language:n (SGML) (ISO 8879) pohjalta. Oikeastaan XML ei itsessään ole kuvauskieli, vaan se määrittelee syntaksin, jolla käyttäjä voi rakentaa omia kuvauskieliään [Tra00, s. 41]. XML:n ensimmäiset versiot (v1.0, 1998) olivat aitoja SGML:n osajoukkoja, sittemmin XML:ään on tullut ominaisuuksia, joita SGML:ssä ei ole, kuten nimiavaruudet (engl. namespaces) ja mallit (schemas) [Tra00, s. 42].

XML muistuttaa rakenteeltaan HTML:ää, mutta on HTML:ää tarkempi syntaksiltaan. XML ei myöskään ota kantaa ulkoasun ja esitystavan kuvaamiseen, joten se soveltuu HTML:ää paremmin pelkän tiedon siirtoon ja ohjelmien väliseen kommunikointiin.

Kun XML-syntaksia lähdettiin kehittämään, tavoitteena oli tehdä helppokäyttöinen, alusta- ja ohjelmistoriippumaton metakieli. XML-standardi onkin vain noin 30 sivun mittainen ja yksinkertaisia XML-dokumentteja on varsin helppo muodostaa. XML-dokumentti on tekstipohjainen ja koostuu elementeistä. Elementeillä on nimi, alku, loppu, sekä mahdollisesti sisältö ja attribuutteja. Esimerkkinä luvussa 4.2.6 rakentamamme Web Servicen palauttama viesti XML-muodossa.

```
  <?xml version="1.0" ?>
- <cArvopaperi xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/1999/XMLSchema" xmlns="http://tempuri.org/">
  <APTunnus>SRA1V</APTunnus>
  <APSijainti>HEX</APSijainti>
  <APNimi>Sonera OYJ</APNimi>
  </cArvopaperi>
```

XML-dokumentti voidaan aina esittää myös puu-muodossa. Kuvassa 14 on edellinen XML-dokumentti esitetty puu-rakenteena.



Kuva 14. xml-dokumentin rakenne.

XML-dokumenttien käsittelyssä käytetään usein ns. parsereita (engl. parser), joilla XML-viesti saadaan purettua. XML-parseri on mahdollista suhteellisen pienellä työllä rakentaa itse, mutta saatavana on myös runsaasti sekä kaupallisia että ilmaisia versioita. .NET:n mukana tulee yleisissä luokkakirjastoissa XML-dokumenttien käsittelyyn tarvittavat luokat ja metodit, jotka löytyvä *system.XML* –nimiavaruudesta.

XML-dokumenttien purkamiseen parsereilla on käytännössä kaksi eri käsittelytapaa. Document Object Model (DOM), sekä Simple API to XML (SAX). DOM käsittelee XML-dokumenttiä oliomaisesti ja on hyvä tilanteissa, joissa tarvitaan koko dokumenttia tai dokumentista luettavan tiedon tarve vaihtelee suorituksen aikana. SAX käsittelee XML-dokumenttia ns peräkkäishakuna ja on DOM:ia parempi tilanteissa, joissa luettava XML-dokumentti on suuri, eikä sitä ole tarvetta tai mahdollista lukea kokonaan.

4.4.2 Simple Object Access Protocol (SOAP)

Simple Object Access Protocol (SOAP) –viestiä käytetään välittämään palvelinkoneelle kutsu johonkin tarvittavaan metodiin parametreineen XML-muodossa [Ble00, s.95]. Metodin palautusarvo tai mahdollinen virheilmoitus palautetaan myös vastaavanlaisena SOAP –viestinä.

Simple Object Access Protocol (SOAP) -viestit välitetään yleensä HTTP-protokollan avulla. Hajautettaessa sovellus toimimaan internetissä, esim. Web Servicejä käyttäen, HTTP-porttia käytettäessä ratkaistaan osaltaan myös yritysten palomuurien aiheuttamat ongelmat.

SOAP-viesti koostuu kolmesta osasta. Kuoresta (envelope), otsikosta (header), sekä rungosta (body) [Box00a]. Kuori pitää sisällään koko SOAP-viestin. Otsikko määrittelee mitä viesti sisältää, kenen sitä kuuluu käsitellä, sekä onko se pakollinen vai ei. Runko sisältää varsinaiset ohjelmakutsut yms [Box00b].

Luvussa 4.2.6 rakentamamme Web Servicen palauttama SOAP-viesti voisi olla seuraavanlainen:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body>
   <HaeArvopaperiResponse xmlns="http://localhost/arvopaperiwebservice">
      <HaeArvopaperiResult>
        <APTunnus>SRA1V</APTunnus>
        <APSijainti>HEX</APSijainti>
        <APNimi>Sonera OYJ</APNimi>
      </HaeArvopaperiResult>
   </HaeArvopaperiResponse>
  </soap:Body>
</soap:Envelope>
```

Esim. CORBA:an (kts. kappale 3.4) tai Microsoftin COM+ -teknologiaan pohjautuvissa hajautetuissa sovelluksissa eri järjestelmän osat kommunikoivat usein tiiviisti keskenään (engl. tightly coupled). SOAP:n avulla luodut etäohjelmakutsut ovat ns. löyhästi sidottuja (engl. loosely coupled), jonka vuoksi SOAP soveltuu em. tekniikoita paremmin internetin yli suoritettaviin ohjelmakutsuihin [Joh00].

SOAP-kutsujen käyttäminen mahdollistaa myös eri alustoilla toimivien sovellusten kommunikoinnin keskenään, koska kutsuvan sovelluksen ei tarvitse tietää kutsuttavan ohjelman rakenteesta muuta kuin sijainti, metodin nimi, sekä vaadittavat parametrit,

4.4.3 Remoting

Remoting on .NET:n toinen hajautusteknologia Web Service:n lisäksi (kts. kappale 4.2.6). Remoting saattaa olla Web Serviceä käytännöllisempi esim. lähiverkossa toimivissa sovelluksissa, jolloin sovellusta ei tarvitse hajauttaa toimimaan internetin yli. Remoting-tiedonsiirto toteutetaan yleensä HTTP- tai TCP-protokollan (Transfer Control Protocol) avulla. Remoting ei tarvitse Microsoftin IIS (Internet Information Server) –palvelinta, toisin kuin Web Service:t.

Käytettäessä asiakaskoneen ja sovelluspalvelimen välisen yhteyden muodostamiseen Remoting-yhteystapaa, sovelluspalvelimella olevia luokkia voidaan käyttää kahdella eri tavalla. Voidaan käyttää asiakkaan aktivoimia- (engl. client-activated objects) tai palvelimen aktivoimia olioita (engl. server-activated objects). Palvelimen aktivoimat oliot voivat olla joko SingleCall tai Singleton –tyyppisiä. SingleCall tyyppinen olio luodaan kutsun ajaksi, eli se on ns. tilaton olio. Singleton-tyyppinen olio on koko ajan muistissa ja palvelee useita eri asiakkaita [Obe00]. Asiakkaan aktivoimien olioiden tapauksessa kyseessä on tilallinen olio, jota ikäänkuin lainataan (engl. leasing) asiakkaalle tietyksi ajaksi. Kun laina-aika menee umpeen, olio tuhotaan, ellei lainaa uusita.

Luokka määritellään Remote-luokaksi perimällä se *MarshalByRefObject*-luokasta [Obe00]. Jos Remote-luokalle viedään parametrinä olio, parametrinä vietävä luokka on määriteltävä serialisoituvaksi (engl. serialized) *serializable* –attribuutilla, esim.

```
Public Class <Serializable> cArvopaperi
```

Serialisoituva luokka voidaan näin lähettää parametrinä esim. SOAP-viestissä. Kun koko luokka määritellään serialisoituvaksi, lähetetään mukana myös yksityiset (enlg. private) muuttujat. Muuttujat, joita ei haluta lähettää olion mukana, voidaan määritellä *Not Serialized* attribuutilla jätettäväksi serialisoinnin ulkopuolelle, esim.

```
<Not Serialized> Private strAPTunnus as String
```

Ennen kuin asiakas pystyy käyttämään Remote-oliota, asiakkaan on rekisteröitävä kanava (engl. Channel) tiedonsiirtoon kutsumalla *ChannelService* –luokan *RegisterChannel* metodia. Samaa kanavaa käytetään usein sekä tiedon lähettämiseen että vastaanottamiseen. Kanavana voidaan käyttää joko TCP- tai HTTP-kanavaa. Kanavien luomiseen tarvittavat luokat löytyvät seuraavista nimiavaruuksista:

```
System.Runtime.Remoting.Channels.TCP
System.Runtime.Remoting.Channels.HTTP
```

Vielä ennen kuin Remote-oliota voidaan käyttää asiakaskoneesta käsin, on itse olio rekisteröitävä. Rekisteröintiin on kaksi tapaa, *RegisterWellKnownType*- tai *ConfigureRemoting* –metodi. *RegisterWellKnownType* –metodilla kaikki rekisteröintiin tarvittavat parametrit ja URI-osoitteet ovat ohjelmakoodissa. *ConfigureRemoting* –metodi tekee saman asian kuin *RegisterWellKnownType* –metodi, mutta parametrit yms. luetaan erillisestä tiedostosta. *ConfigureRemoting* on näin parempi, mikäli osoitteet ja parametrit saattavat vaihtua, koska muutokset tehdään tekstitiedostoon, eikä koko sovellusta tarvitse kääntää uudestaan.

Kun asiakas aktivoi Remote-olion, asiakaskoneelle luodaan ns. proxy-luokan ilmentymä, joka sisältää kutsutun luokan metodit ja muuttujat. Luodun proxy-luokan kautta sovellus hoitaa kutsut Remote-olioon.

Koska asiakaskoneesta käsin hoidetaan kanavien rekisteröintejä yms., asiakaskoneessa on myös oltava .NET CLR asennettuna.

4.5 Kehitysympäristö

Tällä hetkellä, .NET beta 1 vaiheessa, kehitysympäristön voi asentaa Microsoft Windows 2000, Windows NT 4.0, Windows ME ja Windows 98 -käyttöjärjestelmiin. Lisäksi .NET-sovelluksia pystyy ajamaan myös Windows 95 -ympäristöissä, jos koneelle on asennettu .NET CLR [Rel00a]. Poikkeuksena ASP.NET -sovellukset, jotka eivät vaadi lisäasennuksia asiakaskoneeseen, mutta palvelimena täytyy olla Microsoftin Internet Information Server (IIS) -palvelin.

4.5.1 .NET Framework Software Development Kit (SDK)

.NET Framework Software Development Kit (SDK) sisältää vaadittavat kääntäjät, luokkakirjastot ja sovellukset .NET arkkitehtuurin mukaisten ohjelmistojen kehitykseen ja suorittamiseen. .NET-sovelluksia on mahdollista rakentaa pelkän SDK:n avulla, mutta Microsoft:in Visual Studio .NET helpottaa varsinkin graafisten sovellusten kehittämistä huomattavasti. Visual Studio .NET:ä tarkastelemme seuraavassa kappaleessa.

4.5.2 Visual Studio .NET

Visual Studio on Microsoftin sovelluskehitin, sisältäen ympäristöt mm. Visual Basicille ja Visual C++:lle. Suurin muutos Microsoftin Visual Studio.NET:ssä verrattuna Visual Studion aiempiin versioihin on riippumattomuus ohjelmointikielestä. Visual Studio.NET:llä voi rakentaa sovelluksen periaatteessa millä tahansa ohjelmointikielellä, joka on integroitu .NET-kääntäjään, ja saada aikaan saman tuloksen. Tämä on vaatinut suuria muutoksia varsinkin Visual Studion kenties suosituimpaan ohjelmointikieleen, Visual Basic:iin. Visual Basicin uutta versiota kutsutaan nimellä Visual Basic.NET (lyhennetään VB.NET).

Microsoft on kehittänyt .NET:iin myös uuden ohjelmointikielen, C#:n, joka lausutaan C sharp. C# on turvallisempi versio C++ -kielestä ja sitä pidetään Microsoftin vastineena Java-kielelle. C#:n tekee C++:aa turvallisemmaksi mm. automaattinen muistin varaaminen ja vapauttaminen.

Luotaessa uusi projekti Visual Studio.NET:ssä, valitaan minkä tyyppistä projektia ja millä kielellä lähdetään rakentamaan. Periaatteessa lähes kaikki ohjelmointikielet voidaan tuoda .NET ympäristöön. Beta 1 -versiossa mukana ovat mm. C++, Visual Basic, Cobol, Perl ja Eiffel.

Siirtyminen Visual Basicista käyttämään VB.NET:ä ei välttämättä uusien projektien yhteydessä tule aiheuttamaan ongelmia, koska syntaksi pysyy samana, olkoonkin että VB.NET on täydellinen olio-kieli [Hol01a]. Sen sijaan ongelmia syntyy, jos yritetään siirtää VB6:lla tai vanhemmalla Visual Basicilla tehty sovellus, jonka koodi on kenties lisäksi huonosti jäsenneltyä, .NET ympäristöön. Vaikkakin VB6:n kääntämiseen VB.NET:ksi on tehty ”velhoja” (engl. wizard), jotka tekevät koodiin tarvittavia muutoksia ja ilmoittavat kohdista, jotka käyttäjän tulee tarkastaa, koodivirheitäkin tulee helposti jäämään. Mutta ehkä vielä suurempi ongelma syntyy, jos ollaan siirtämässä vanhaa kaksitasoarkkitehtuuriin tehtyä sovellusta monitasoarkkitehtuuriin. Koska VB6 ja sitä aiemmat Visual Basicin versiot eivät täysin ole tukeneet olio-ohjelmointia, on sovelluslogiikkakerroksen erottaminen käyttöliittymäkerroksesta sitäkin vaikeampaa.

Koodillisia eroavaisuuksia VB6:lla ja VB.NET:llä on mm. tietotyypeissä, taulukoiden käytössä, lomakkeiden luomisessa (WinForms), virheiden käsittelyssä jne. [MS00b]. VB6 sovelluksen voi rakentaa niin, että siirtyminen VB.NET:iin käy helposti, mutta vanhemmissa sovelluksissa näitä asioita ei ole voitu tietää (ohjeita mm. MS00c). Uutena VB.NET:ssä on jo mainittujen olio-ominaisuuksien lisäksi mm. säikeet.

4.6 .NET -sovellusten toimittaminen ja asentaminen

.NET-sovellus toimitetaan yhtenä tai useampana pakettina, joita kutsutaan assembly:iksi [Ric01a], vapaasti suomennettuna kokoonpano. Kaikki kokoonpanon sisältämät komponentit ovat oletuksena ainoastaan yhden, asennettavan ohjelman käytettävissä. Asennus onkin mahdollista suorittaa uuteen hakemistoon ns. XCOPY-toimintona, joten asennus ei vaikuta jo käytössä olevien ohjelmien toimivuuteen, koska mitään tiedostoja ei päivitetä [Ric01a]. Asennus ei myöskään tee merkintöjä käyttöjärjestelmän rekisteriin (engl. registry). Kokoelma sisältää mm. tiedon siihen kuuluvista tiedostoista, käyttöoikeusvaatimukset sekä versionumeronsa. Rakentamamme Windows Form -esimerkin tapauksessa asennukseen sisältyy ainoastaa exe-tiedosto, laajemmissa sovelluksissa mukana on usein myös esim. dll-tiedostoja.

Kokoelmaa, joka asennetaan ainoastaan sovelluksen omaan käyttöön kutsutaan yksityiseksi kokoelmaksi (engl. private assembly) ja kokoelmaa, joka jaetaan muiden sovellusten käyttöön kutsutaan jaetuksi kokoelmaksi (shared assembly) [Ric01b]. Kokoelmien asentaminen yksityiseksi vaatii paljon levytilaa, koska sovellukset eivät enää jaa tiedostoja.

Asennettaessa .NET-sovellusta, se on Microsoft Intermediate Language (MSIL) muodossa. MSIL-koodi voidaan kääntää natiivikoodiksi asennusvaiheessa, tai vasta suorituksen yhteydessä. WinForms ja muut .NET Framework -pohjaiset sovellukset ovat helppoja asentaa ja ylläpitää, koska asennettaessa ei enää tarvitse huolehtia komponenttien rekisteröinnistä yms. [Con00, s.22]. Seuraavassa kappaleessa käsitellään ohjelman kääntämistä MSIL välikieleltä natiivikoodiksi.

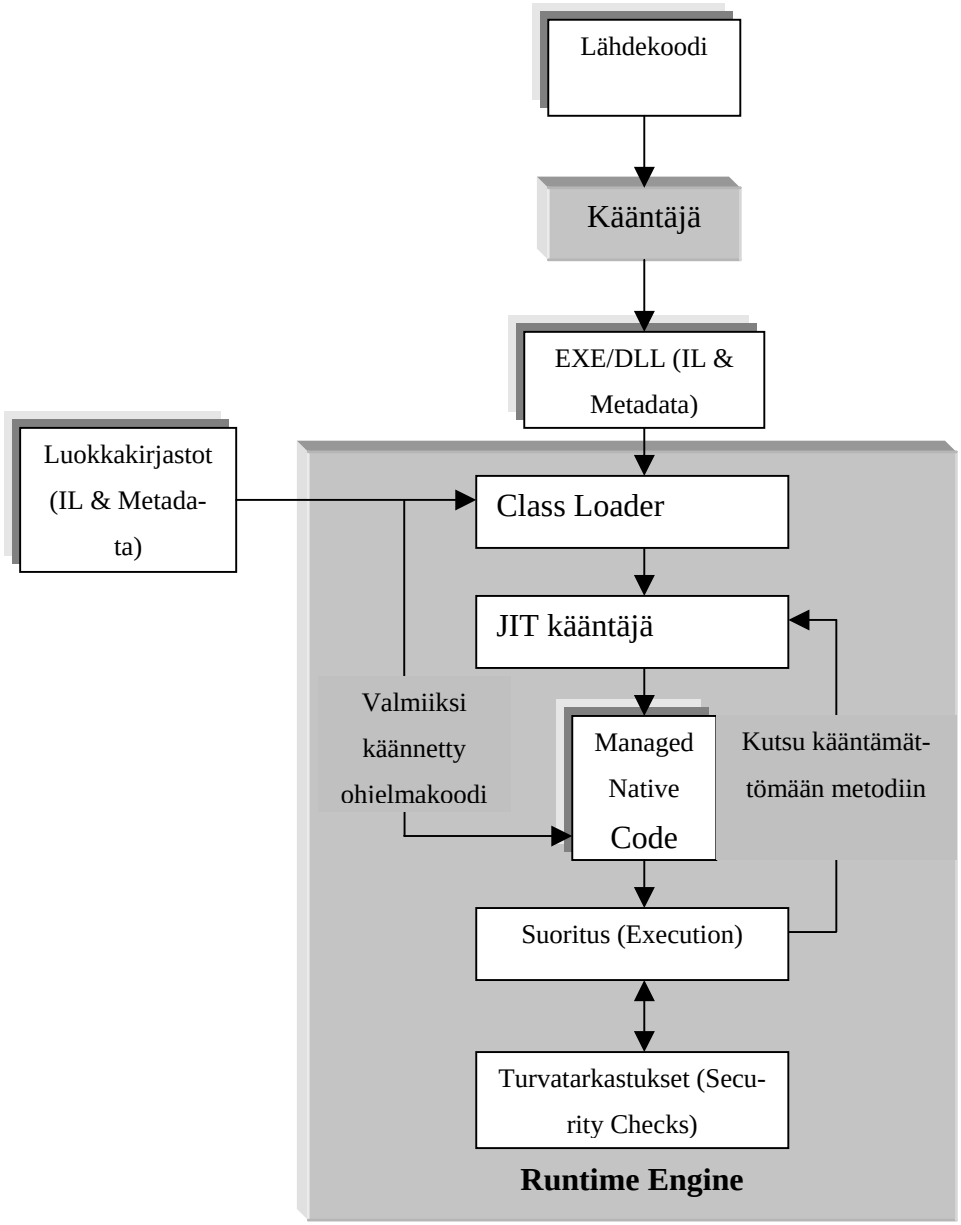
4.6.1 Ohjelman kääntäminen

.NET-sovellus ei siis ole vielä asennuksen yhteydessä ajettavassa muodossa, vaan koodi on ensin käännettävä natiivikoodiksi, jota prosessori ymmärtää. Kääntämiseen on kaksi tapaa. Voidaan joko käyttää Just In Time (JIT) -kääntäjiä, joilla sovellus käännetään juuri ennen suoritusta ja mahdollisesti ainoastaan tarvittava osa sovelluksesta, tai koko sovellus voidaan kääntää asennusvaiheessa, jolloin itse suoritus nopeutuu. JIT-kääntäjiä on kolme erilaista, EconoJIT, JIT, sekä OptJIT [MS00d], jokainen eri asioihin optimoituja. Esim. EconoJIT-kääntäjällä kääntäminen tapahtuu nopeasti, mutta lopputulosta ei ole optimoitu nopeuden suhteen. JIT-kääntäjä taas saattaa käyttää hieman enemmän aikaa itse kääntämiseen, mutta lopputuloksena on nopeampi ohjelma. Riippuu täysin kohteesta, mitä kääntäjää kannattaa käyttää [Con00, s.20].

Vaikka ohjelma suoritettaisiinkin käyttäen JIT-kääntäjiä, ei suoritettavaa osiota silti tarvitse välttämättä kääntää joka kerta uudestaan, vaan käännetyt osat tallennetaan levylle tulevaisuuden käyttöä varten [Con00, s.19]. Kääntäjien luvataan osaavan optimoida koodin tietylle prosessorityypille, erottaen esim. eri Pentium-prosessorit toisistaan [Ric00a].

4.6.2 Ohjelman suorittaminen

Jos ohjelma käännetään natiivikoodiksi jo asennusvaiheessa, ohjelman suorittaminen on suoraviivaista, periaatteessa sen hoitaa Runtime Engine. Jos taas käytetään JIT-kääntäjiä, tarvitsee Runtime Enginen ensin kääntää ainakin osa koodista. Kuvassa 15 on esimerkki ohjelman suorittamisesta, kun käytetään JIT-kääntäjää.



Kuva 15. Ohjelman suoritus .NET ympäristössä [kuva: Ric00a]

Kuvan 14 tapauksessa lähdekoodi käännetään ensin Microsoft Intermediate Languageksi (MSIL). Tuloksena on exe- tai dll-tiedosto, joka sisältää MSIL-koodin, sekä tietoja ohjelmasta (Metadata). Seuraavassa vaiheessa Class Loader huolehtii mm. koodin lähteen, esim. Web Servicen URI-osoitteen, ja julkaisijan tietojen ”lukemisesta” ja pystyy niiden perusteella antamaan suoritettavalle koodille oikeuksia ja rajoituksia [Ric00b]. Just In Time (JIT) -kääntäjä huolehtii MSIL-koodin kääntämisestä natiivikoodiksi, joka on valmis suoritettavaksi. Mikäli koodista käännetään ennen suoritusta ainoastaan suoritettava osa, kääntämätöntä MSIL-koodia kutsuttaessa se käännetään ajon aikana JIT-kääntäjässä.

4.7 Tietoturva

Komponenttipohjaisissa sovelluksissa on usein ilmennyt tietoturvaongelmia [Bro01]. Useiden eri valmistajien komponenttien käyttäminen samassa sovelluksessa saattaa jättää aukkoja tietoturvaan, joko sovelluskehittäjien huolimattomuudesta tai komponenttien valmistusvirheistä johtuen.

.NET-arkkitehtuurissa on tietoturvaa yritetty parantaa mm. Common Language Runtime:n sisältämän Code Access Security:n avulla, joka valvoo ohjelman suoritusta myöntäen oikeuksia ja asettaen rajoituksia mm. ohjelmakoodin alkuperän mukaan.

Tässä kappaleessa tarkastelemme, kuinka .NET-sovelluksissa on mahdollista hoitaa tietoturvaan liittyvät asiat Code Access Security:n ja muiden .NET:n ominaisuuksien avulla.

4.7.1 Framework-luokkien turvaominaisuudet

.NET-arkkitehtuuri sisältää runsaasti valmiita luokkia, joissa on sisäänrakennetut turvaominaisuudet tietoturvaa vaativien toimintojen rakentamiseen. Luokkia löytyy mm. tiedostojen- ja leikepöydän käsittelyyn jne., joiden toimintojen valvontaa on automatisoitu. Lisäksi .NET Framework sisältää luokkia ja metodeja tiedon salaamiseen ja suoritusoikeuksien määrittelyyn yms. *System.Security* –nimiavaruuden alta löytyy mm. seuraavat luokat:

```
System.Security.Cryptography
System.Security.Policy
System.Security.Permissions
```

System.Cryptography -luokka sisältää metodit mm. tiedon salakirjoittamiseen ja purkamiseen eri algoritmeilla. *System.Permissions* ja *System.Policy* –luokkien avulla voidaan määritellä käyttöoikeuksia yms., perustuen mm. ohjelmakoodin julkaisijaan, www-sivuun jolta ohjelmakoodi on peräisin, jne.

Käyttöoikeuksien avulla on mahdollista rajoittaa valmiiden .NET Framework -luokkien käyttöä, joissa on sisäänrakennetut tietoturvaominaisuudet. Esim. yllämainitun tiedoston käsittelyyn vaadittavan *System.IO.FileStream*- luokan ilmentymää luodessa tarkastetaan käyttäjäoikeudet ja annetaan oikeudet ainoastaan tiettyihin toimintoihin.

4.7.2 Ohjelmakoodin turvatarkastukset

.NET Code Access Security sisältää tarvittavat ominaisuudet vieraan ohjelmakoodin suoritusoikeuksien määrittelyyn. Code Access Security:n avulla voidaan määritellä ohjelmakoodille oikeuksia, riippuen sen alkuperästä, julkaisijasta jne. Esimerkiksi oikeudet kirjoittaa kiintolevylle voidaan antaa vain tietyn julkaisijan koodille [MS00e].

.NET Role Based Security on käyttäjätasoihin perustuva käyttöoikeuksien määrittelytapa. Role Based Security on verrattavissa Windows NT –käyttöjärjestelmän käyttäjätasoihin, jossa käyttäjät jaetaan tiettyihin ryhmiin, esim. Administrator, Guest jne., joille on valmiiksi määritelty tietty joukko oikeuksia, joita voidaan laajentaa. Microsoftin COM+ 1.0:ssa on samantyyppinen tarkastusmekanismi, mutta siinä käyttäjätasot sidottu Windows NT- tai Windows 2000 -käyttäjätasoihin, kun taas .NET ei vaadi NT:tä. Tosin myös .NET osaa haluttaessa hyödyntää käyttöjärjestelmän käyttäjätasoja [MS00e].

Käyttäjän tunnistaminen Role Based Security:ssä tapahtuu ns. Authentication-toiminnolla, jossa selvitetään esim. käyttäjätunnus ja salasana. Käyttäjän tunnistamiseksi .NET-sovelluksissa on monta tapaa. Em. käyttöjärjestelmän käyttäjätasojen hyödyntämisen lisäksi voidaan käyttää esim. Microsoftin Passport-tunnistautumista tai itse rakennettua tunnistautumisjärjestelmää. Varsinaista käyttäjäoikeuksien myöntämistä kutsutaan Authorization:ksi ja se tapahtuu yleensä Authentication-toiminnon jälkeen [MS01a].

Sekä Code Access Security:ä, että Role Based Security:ä hoitaa suorituksen aikana Common Language Runtime (CLR). CLR pystyy valvomaan .NET-ympäristössä ns. managed code:a (kts. kappale 4.2.1), mutta jos ajetaan vanhoja .NET-ympäristöön liitettyjä tai C++:lla tarkoituksella rakennettuja ns. unmanaged-sovelluksia, CLR ei pysty enää valvomaan suoritettavaa koodia kokonaisuudessaan [MS00e].

4.7.3 Tiedon salaaminen

.NET Framework sisältää valmiit toiminnot tiedon salaamiseen, satunnaislukujen muodostamiseen yms. *System.Security.Cryptography* –nimiavaruudessa. Framework sisältää tiedon salaamiseen sekä symmetrisiä yhdellä salausavaimella toimivia, että asymmetrisiä, ns. Public key -algoritmeja [MS00e], joissa käytetään kahta salausavainta. Asymmetrisessä algoritmissa käytettävistä avaimista toinen on julkinen (engl. public) ja toinen on ainoastaan yhden osapuolen tiedossa. Asymmetrisiä algoritmeja käyttäen voidaan myös digitaalisesti allekirjoittaa tiedostoja (engl. digital signature).

4.7.4 Web Service:n ja ASP.NET:n turvaominaisuudet

ASP.NET vaatii toimiakseen Microsoftin Internet Information Server:in (IIS) ja pystyy myös hyödyntämään siihen rakennettuja turvaominaisuuksia. IIS.n avulla käyttäjän tunnistukseen (engl. authentication) voidaan käyttää mm. Basic, Basic / SSL, Digest, Integrated Windows Authentication tai Client Sertificates tunnistuksia [Mor99]. Lisäksi voidaan käyttää mm. ns. cookie:ta tai kuten muissakin .NET-sovelluksissa itse rakennettuja, Web Service:n tapauksessa esim. SOAP-viestin avulla toimivia tunnistusmekanismeja [MS00e].

Microsoftin Passport -tunnistusmekanismi ei varsinaisesti sovellu Web Service:n kanssa toimivaksi, koska Web Service:ä kutsutaan yleensä sovelluksesta käsin, eikä suoraan esim. käyttäjän selaimelta, johon Passport on rakennnettu.

Mikäli kaikkien käyttäjien IP-tunnukset (Internet Protocol) on tiedossa, voidaan Web Service:n käyttäjiä kontrolloida palomuurin tai esim. Internet Protocol Security:n (IPSec) avulla. Usein kaikkien käyttäjien IP-tunnukset eivät kuitenkaan ole tiedossa, jolloin joudutaan käyttämään muita suojaustekniikoita.

Käyttäjän tunnistamisen jälkeen oikeuksia voidaan myöntää joko vertaamalla käyttäjätunnusta ja salasanaa NTFS-tiedostojärjestelmän oikeuksiin tai lukemalla esim. XML-tiedostosta käyttäjätunnusta vastaavat käyttöoikeudet yms.

Vaikka ASP.NET perustuukin pitkälti SOAP-protokollan hyödyntämiseen, SOAP-viestien salaamiseen liittyvät standardit ovat vasta kehitteillä. Kehitteillä on mm. XML-dokumenttien digitaaliseen allekirjoitukseen tarkoitettu standardi, XML Digital Signature specification (XMLDSIG) [MS01a], sekä vastaava SOAP standardi [Ris01]. Tulevaisuudessa voidaan odottaa standardeja myös XML- ja SOAP-viestien salaukseen.

SOAP-viestien lähettäminen täysin salaamattomina ja ilman digitaalisia allekirjoituksia saattaa myös mahdollistaa sovellusten väärinkäytön [Ris01]. Koska HTTP:n avulla välitetyt SOAP-viestit pääsevät yleensä helposti palomuureista läpi, niitä saattaa olla mahdollista myös käyttää väärin sovellusten kuormittamiseen tai tietomurtoihin.

4.7.5 MSIL koodin purkaminen

MSIL (Microsoft Intermediate Language) –koodi on mahdollista kääntää takaisin luettavaan muotoon, *debug*-käännöksellä käännetty versio jopa täysin identtiseksi alkuperäisen lähdekoodin kanssa [Con00, s. 26]. .NET SDK:n mukana tulee ohjelmat, joilla kääntäminen on mahdollista toteuttaa. Alla näemme esimerkin arvopaperi-sovelluksesta takaisin lähdekoodiksi käännetystä MSIL-koodista. Koodi on käännetty .NET SDK:n mukana tulleella ildasm.exe –ohjelmalla, /SOURCE –optiolla, jolloin konekielen lisäksi myös alkuperäinen lähdekoodi tulee näkyviin, mikäli ohjelma on käännetty *debug*-käännöksellä. Esimerkissä alkuperäinen lähdekoodi näkyy kommentoituna. Mikäli ohjelmasta on tehty *release*-käännös, varsinainen ohjelmakoodi ei tule näkyviin, mutta ohjelman rakenne on silti selvitettävissä. Arvopaperi-sovelluksen Button1_click –metodin MSIL-koodista käännetty koodi kokonaisuudessaan liitteessä 8.9.

```
//000089:          Dim Tulos As New System.Collections.ArrayList()
      IL_000d:  newobj     instance void
[mscorlib]System.Collections.ArrayList::.ctor()
      IL_0012:  stloc.3
//000090:
//000091:
//000092:          APt.LataaArvopaperi(TextBoxTunnus.Text)
      IL_0013:  ldloc.1
      IL_0014:  ldarg.0
      IL_0015:  callvirt instance class
[System.WinForms]System.WinForms.TextBox
WindowsApplication1.Form1::get_TextBoxTunnus()
      IL_001a:  callvirt   instance class System.String
[System.WinForms]System.WinForms.TextBoxBase::get_Text()
      IL_001f:  callvirt   instance void
WindowsApplication1.cArvopaperit::LataaArvopaperi(class System.String)
//000093:          Tulos = APt.Arvopaperit
      IL_0024:  ldloc.1
      IL_0025:  callvirt   instance class
```

.NET sovelluksille pystyy antamaan käännettäessä ns. OWNER-option, joka toimii ikäänkuin salasanana takaisin lähdekoodiksi kääntämistä vastaan. OWNER-optio toimii kuitenkin ainoastaan käännettäessä .NET SDK:n mukana tulleilla kääntäjillä, sen sijaan verkosta löytyy kääntäjiä, jotka eivät välitä OWNER-optiosta, vaan pystyvät siitä huolimatta muodostamaan lähdekoodin. Sama ongelma on myös Java-sovelluksissa, joissa Java:lla rakennettu .class-luokka on mahdollista helposti kääntää takaisin lähdekoodiksi.

Mahdollisuus sovelluksen kääntämiseen takaisin luettavaan muotoon muodostaa tietoturvariskejä. Vaikka asiakassovellus on monitasoarkkitehtuurissa yleensä vain käyttöliittymä, lähdekoodin avulla voi silti saada aikaan vahinkoa. Lisäksi tulevaisuudessa kaikki .NET-sovellukset, eli suurin osa Windows-sovelluksista, on mahdollista kääntää takaisin lähdekoodiksi, ellei MSIL-koodin salaukseen toimiteta menetelmää.

Useimmat sovellukset on lähes aina ollut mahdollista purkaa tavalla tai toisella, mutta mitä helpommin lähdekoodi on saatavissa, sitä suurempi on riski lähdekoodin väärinkäytölle.

4.8 Suorituskyky

.NET-sovellusten lopullista suorituskykyä on vielä hieman varhaista arvioida, koska .NET on vielä testausvaiheessa, mutta tässä kappaleessa käymme läpi suorituskykyyn mahdollisesti vaikuttavia asioita.

4.8.1 Suorituskykyä heikentäviä asioita

Ehkä suurin yksittäinen suorituskykyä heikentävä tekijä .NET:ssä on Common Language Runtime (CLR) (kts. kappale 4.2.1.). CLR parantaa sovellusten turvallisuutta sekä helpottaa sovelluskehitystä, mutta suorituskyvyn kustannuksella. Sen lisäksi, että sovellusten kääntäminen Microsoft Intermediate Language:sta (MSIL) suoritettavaan muotoon hidastaa niiden suoritusta, CLR:n toiminnot vaikuttavat suorituskykyyn myös sovelluksen kääntämisen jälkeen. Tosin monia CLR:n sisältämistä ominaisuuksista voidaan pitää välttämättöminä, olivat ne sitten osa CLR:ää tai itse sovellusta. Tällaisia ovat esim. roskienkeruu, käyttäjäoikeuksien yms. valvonta jne.

Koska Web Service:t ovat olennainen osa .NET-arkkitehtuuria, myös niiden suorituskyky tulee olemaan ratkaisevassa asemassa. Web Service:kin vaatii palvelinkoneella CLR:n, joka osittain heikentää suorituskykyä. Web Servicen tapauksessa on kuitenkin myös useita muita riskitekijöitä suorituskyvyn suhteen.

Web Service:n tiedonsiirtoon käytetään usein HTTP:tä (HyperText Transfer Protocol), jota ei varsinaisesti ole rakennettu nopeaa tiedonsiirtoa varten. Hajautettaessa sovellus toimimaan internetin yli, on aina riski että yhteydet toimivat hitaasti tai ei lainkaan.

Kommunikointi sovellusten ja Web Service:n välillä tapahtuu yleensä XML-pohjaisina SOAP-viesteinä, joiden käsittelyyn tarvitaan aina ns. parsereita, joilla viestit voidaan kasata ja purkaa. Parserien toiminta taas vie aina oman aikansa.

Koska Web Service:t lähettävät tietoa verkon yli, myös tiedon riittävä salaaminen on tärkeää. Mitä varmempaa salausmenetelmää käytetään, yleensä sitä enemmän resursseja tiedon salaaminen ja purkaminen vaatii.

4.8.2 Suorituskykyä parantavia asioita

Suorituskykyä parantavat asiat liittyvät suurelta osin edellisessä kappaleessa esitettyihin suorituskykyä heikentävien asioiden korjaamiseen. Yhtenä tärkeimpänä suorituskykyä parantavana asiana voidaan pitää MSIL:stä natiivikoodiksi käännetyn ohjelmakoodin säilyttämistä levyllä (cache) tulevaa käyttöä varten. CLR osaa haluttaessa myös kääntää vain suoritettavan osan ohjelmasta, joka myös osaltaan nopeuttaa suurten ohjelmien käsittelyä tai ohjelma voidaan kääntää natiivikoodiksi kokonaan jo asennusvaiheessa. Joka tapauksessa .NET ohjelmat eivät kaikissa tilanteissa toimi yhtä nopeasti kuin esim. pelkällä C++:lla tehdyt sovellukset, johtuen ajonaikaisesta CLR:stä.

Toisaalta monet CLR:n ominaisuudet saattavat toimiessaan parantaa suorituskykyä siinä mielessä, että ohjelmat ovat luotettavampia ja turvallisempia kuin ilman CLR:ää rakennetut sovellukset.

4.9 Yhteenveto .NET arkkitehtuurista

Microsoftin .NET–arkkitehtuuri tuo mukanaan paljon uudistuksia varsinkin Microsoftin alustoilla toimiviin järjestelmiin. .NET helpottaa sovellusten hajauttamista ja tuo uusia mahdollisuuksia monitasoarkkitehtuuristen järjestelmien rakentamiseen. .NET–sovellusten luvataan myös olevan entistä turvallisempia mm. CLR:n muistinhallinnan sekä sovellusten uuden asennustavan (yksityiset kokoelmat) myötä.

Näyttää siltä, että ainakin Microsoftin alustoille ja varsinkin Microsoftin kehitysvälineillä tehtävät sovellukset tulevat siirtymään .NET:iin, mutta on vaikea sanoa, tuleeko .NET kuinka suuressa määrin syrjäyttämään esim. Java:a kehitysvälineenä. .NET sisältää paljon samoja ominaisuuksia kuin J2EE ja sitä onkin syytetty osittain ideoiden kopioimisesta Java:sta.

.NET-sovellusten yleistymiseen tulee myös vaikuttamaan CLR-ympäristön jakelu ja asentaminen. CLR toimitetaan mm. uusimman Microsoft Internet Explorer:in mukana (versiosta 6.0 alkaen) sekä lisäksi ainakin uudessa WindowsXP:ssä. Joka tapauksessa CLR on asennettava koneelle, jolla ajetaan .NET-sovelluksia. Vaikka Microsoft onkin kertonut mahdollisuudesta toteuttaa CLR myös muille kuin Windows-alustoille, asian toteutuksesta ei ole mitään näyttöä. Todennäköistä onkin, että .NET on ainakin toistaiseksi teknologia, joka tuo Microsoftin kehitysvälineitä käyttäville vastaavat työkalut kuin esim. Java puolella on jo olemassa, mutta on valitettavan tiukasti sidottu Microsoftin omiin käyttöympäristöihin.

5 Järjestelmäarkkitehtuurin uudistamisesta

Tässä kappaleessa tarkastelemme olemassaolevien sovellusten siirtämistä monitasoiseen .NET–arkkitehtuuriin. Tarkastelemme lähinnä tilannetta, jossa vanhat sovellukset ovat Microsoftin kehitysvälineillä rakennettuja kaksitasoarkkitehtuurisovelluksia.

5.1 Valmistautuminen arkkitehtuurin uudistamiseen

Sen lisäksi, että tässä kappaleessa tarkastelemme tapoja, joiden avulla on mahdollista valmistaua olemassaolevan sovelluksen siirtämiseen .NET-ympäristöön, tarkastelemme myös miten uudet sovellukset tulisi rakentaa .NET:ä vanhemmilla tekniikoilla, jotta tulevaisuudessa mahdollinen siirtyminen uuteen arkkitehtuuriin olisi mahdollista. Koska .NET on tätä kirjoitettaessa vielä beta-vaiheessa, uusien projektien aloittaminen suoraan .NET:ä käyttäen olisi riskialtista.

Jos vanha sovellus toteutettu 3-tasomallin mukaisesti, siirtyminen .NET:iin on lähinnä ohjelmakoodin muokkausta vastaamaan .NET:n uudistettuja ohjelmointikieliä (esim. VB.NET, Managed C++, C#). .NET:n mukana toimitetaan myös ns. velhoja (engl. wizard), jotka osaavat automaattisesti päivittää ohjelmakoodia .NET-ympäristöön, mutta nekään eivät pysty täysin luotettavasti muuntamaan koko ohjelmakoodia.

Jos vanhat sovellukset rakennettu asiakas/palvelin –arkkitehtuuriin, uudistamisessa on enemmän työtä. Asiakas/palvelin –arkkitehtuuriin rakennettu sovellus voi sekin olla rakennettu hyvin tai huonosti silmälläpitäen monitasoarkkitehtuuriin siirtymistä.

Gartner Group -tutkimuslaitos on listannut ohjeita, joita Microsoftin sovelluskehitysvälineitä käyttävien olisi hyvä noudattaa, jotta uuteen arkkitehtuuriin siirtyminen olisi mahdollisimman helppoa. Gartner Group listaa mm. seuraavat asiat [Gar00a]:

1. Jatka COM teknologioiden käyttämistä
2. Kehitä palveluihin pohjautuvaa rakennetta (engl. service-oriented)
3. Erota sovelluslogiikka käyttöliittymästä

4. Käytä mahdollisuuksien mukaan Microsoft Message Queue Server (MSMQ) –viestijonoja, jotka muistuttavat .NET:n tukemaa löyhästi sidottua (engl. Loosely coupled) viestintätapaa.

5. Käytä mahdollisuuksien mukaan SOAP–viestejä COM-komponenttien kutsumisessa.

COM-komponenttien käyttö helpottaa monessakin mielessä .NET-arkkitehtuuriin siirtymistä. Koska Microsoftin edellinen suosittu hajautusteknologia pohjautuu suurelta osin COM-komponentteihin, niiden käyttö on osaksi senkin takia suoraan mahdollista .NET-sovelluksista. Lisäksi kaikkia .NET luokkia voidaan käyttää COM-komponenttien tavoin. Vaikka COM-komponentit on alunperin rakennettu toimimaan tilallisen yhteyden kautta, ne voidaan rakentaa toimimaan myös tilattomien yhteyksien avulla, joka on yleisin .NET-arkkitehtuurissa käytetty yhteystapa.

Palveluihin pohjautuva rakenne tarkoittaa juuri edellämainittujen tilattomien yhteyksien avulla toimivia komponentteja tai vastaavia. Siinä asiakas antaa esim. sovelluspalvelimelle suorituspyynnön, johon sovelluspalvelin suorituksen jälkeen vastaa, mutta yhteyttä ei jätetä auki. .NET-arkkitehtuurissa esim. Web Service:t pohjautuvat juuri palvelutyyppisiin toimintoihin.

Koska monitasoarkkitehtuurissa käyttöliittymä toimii omalla työasemallaan, sen on oltava erillään sovelluslogiikasta. Vaikka sovellus rakennettaisinkin asiakas/palvelin-arkkitehtuuriin, käyttöliittymä ja sovelluslogiikka on silti mahdollista rakentaa erilleen, esim. sijoittamalla sovelluslogiikka omiin luokkiinsa, vaikka se toimisikin saman sovelluksen sisällä. Sovelluslogiikan eriyttämistä käyttöliittymästä tarkastelemme vielä kappaleessa 5.2.

Eräs Microsoftin teknologia tilattomien yhteyksien hallintaan on Microsoft Message Queue Server (MSMQ) –viestijonot.

Vaikka SOAP-protokolla on Microsoftin teknologioissa laajemmin käytössä vasta .NET-arkkitehtuurissa, sen käyttö on mahdollista myös vanhemmissa sovelluksissa. SOAP-viestien käsittelyyn on mahdollista itse rakentaa tarvittavat metodit, periaatteessa riittää että ohjelma osaa lähettää ja vastaanottaa HTTP-sanomia. SOAP-viestien käsittelyyn on tarjolla myös valmiiksi rakennettuja komponentteja. Microsoftin SOAP Toolkit helpottaa vanhojen sovellusten siirtämistä .NET-arkkitehtuuriin [Hol00], mutta edellyttää Microsoft Internet Explorer version 5.5 tai uudemman asennusta.

Tärkeä monitasoarkkitehtuuriin siirtymistä helpottava tekijä on hyvin jäsennelty ohjelman rakenne. Koska yksi suurimpia muutoksia siirryttäessä kaksitasoarkkitehtuurista monitasoarkkitehtuuriin on käyttöliittymän ja sovelluslogiikan eriyttäminen, hyvin jäsennelty tai selkeän oliomallin mukaisesti rakennettu sovellus helpottaa monitasoarkkitehtuuriin siirtymistä huomattavasti.

5.2 Sovelluslogiikan eriyttäminen käyttöliittymästä

Jos asiakas/palvelin –arkkitehtuurissa on jo valmiiksi käyttöliittymän ohjelmakoodi eroteltu sovelluslogiikasta, monitasoarkkitehtuuriin siirtymisen ei tulisi aiheuttaa suuria ongelmia. Usein asiakas/palvelin –arkkitehtuurissa on kuitenkin sovelluslogiikkaa rakennettu suoraan käyttöliittymän sekaan, jolloin niiden erotteleminen on työläämpää. Tavoitteena on tehdä täysin erillinen sovelluslogiikkakerros, johon voidaan tarpeen mukaan liittää jopa useita erilaisia käyttöliittymiä.

Microsoftin kehitysvälineillä varsinkin Visual Basic –ohjelmointikielellä rakennetut sovellukset saattavat usein sisältää paljonkin toiminnallisuutta käyttöliittymän seassa, koska Visual Basic ei ole ollut tarkka ohjelman rakenteen suhteen. Tilanne saattaa olla parempi olio-ohjelmointikielillä (esim. C++) tehdyissä sovelluksissa, joissa ohjelmakoodi on selkeämmin jäsenneltyä.

Täysin käyttöliittymästä eriytettyyn sovelluslogiikkaan voidaan rakentaa esim. www- käyttöliittymä, Windows- käyttöliittymä tai siitä voidaan pienen rajapinnan avulla tehdä Web Service. Ideana kuitenkin on, että sovelluslogiikkaan ei jouduta tekemään muutoksia riippuen siitä, millaista käyttöliittymää käytetään. Esim. Web Service:n tapauksessa ei siis lisätä ohjelmakoodiin <WebService> määreitä tai vastaavia, vaan ne rakennetaan erilliseksi rajapinnaksi, jonka kautta sovelluslogiikkaa kutsutaan.

Siirryttäessä .NET-arkkitehtuuriin voidaan järjestelmä siirtää aluksi .NET-ympäristöön vain osittain, esimerkiksi sovelluslogiikka voidaan siirtää .NET-ympäristöön ja käyttöliittyminä voidaan käyttää nykyisiä käyttöliittymiä. Tai sovelluslogiikka voi toimia nykyisissä järjestelmissä ja käyttöliittymät rakennetaan .NET-ympäristöön, jolloin kaikilla asiakaskoneilla vaaditaan .NET Common Language Runtime -ympäristö. Sovelluslogiikan ja käyttöliittymän on osattava kommukoida keskenään, jolloin myös vanhempiin järjestelmän osiin on rakennettava rajapinta, joka osaa kommunikoida .NET-sovellusten kanssa. Käyttöliittymän ja sovelluslogiikan välistä kommmunikointia tarkastelemme kappaleessa 5.3.

5.3 Käyttöliittymä- ja sovelluskerroksen välinen kommunikointi

Monitasoarkkitehtuurissa käyttöliittymän ja sovelluslogiikan välinen kommunikointi tapahtuu yleensä verkon yli, toisin kuin kaksitasoarkkitehtuurissa, jossa käyttöliittymä ja sovelluslogiikka sijaitsevat samassa sovelluksessa. .NET tarjoaa käyttöliittymä- ja sovelluslogiikkakerroksen välisen kommunikoinnin toteuttamiseen valmiita luokkia. Käyttöliittymätyypin ja käyttötarkoituksen perusteella voidaan yhteystapa toteuttaa joko valmiilla luokilla tai itse rakennetuilla tekniikoilla.

.NET tarjoaa valmiita luokkia mm. SOAP (kts. kappale 4.4.2.) viestien käsittelyyn (sisältäen mm. viestien muodostamisen, lähettämisen, vastaanottamisen ja purkamisen), Remoting-yhteyden (kts. kappale 4.4.3.) muodostamiseen, tiedon lähetykseen ja vastaanottamiseen Remoting-kanavaa pitkin esim. HTTP tai TCP protokollien avulla.

SOAP-prokollaa hyödyntävät ratkaisut sopivat sovelluksiin, jotka hajautetaan toimimaan lähiverkkojen ulkopuolelle, esim. internetiin. Remoting-yhteystapa sopii sovelluksiin, joissa järjestelmä toimii lähiverkon sisällä. Remoting mahdollistaa valmiita SOAP-komponentteja laajemman siirretävän tiedon ja yhteystavan kontrolloinnin.

Mikäli sovellukset siirretään .NET-ympäristöön vain osittain, valmiiden .NET-luokkien täysipainoinen hyödyntäminen ei ole mahdollista. Vanhoihin sovelluksiin tiedonsiirtorajapinta on rakennettava erikseen. Tällöin voidaan käyttää apuna mm. kappaleessa 5.1. mainittua SOAP Toolkit:ä, jonka avulla voidaan olla yhteydessä esim. Web Service –palveluihin.

Riippumatta siitä, mikä yhteystapa käyttöliittymä- ja sovelluskerroksen välille valitaan, kaikki tiedonsiirtoon vaadittavat toiminnot tulisi rakentaa omaksi rajapinnakseen, joka on erillään muusta sovelluksesta. Tällöin rajapinta toimii eräänlaisena edustakerroksena sovellukseen, jolloin yhteystavan mahdollinen vaihtaminen tai useiden eri yhteystapojen käyttäminen samassa sovelluksessa on helpompaa.

6 Yhteenveto

Monitasoarkkitehtuuri tarjoaa useita etuja kaksitasoarkkitehtuuriin nähden, varsinkin suurten, jopa tuhansien käyttäjien järjestelmissä. Sen lisäksi, että monitasoarkkitehtuuri mahdollistaa suuren määrän yhtäaikaisia käyttäjiä, se helpottaa sovellusten asennusta ja hallintaa. Kun asiakas/palvelin –arkkitehtuurissa yhtäaikaisten käyttäjien määrä on usein rajattu, monitasoarkkitehtuurissa vastaavaa ylärajaa käyttäjien määrälle ei muodostu.

Vaikka monitasoarkkitehtuuri tarjoaa monissa tilanteissa etuja kaksitasoarkkitehtuuriin nähden, tässä tutkielmassa tarkasteltiin myös onko kaksitasoarkkitehtuuriin rakennettuja järjestelmiä kannattavaa siirtää monitasoarkkitehtuuriin. Siitä huolimatta, että monitasoarkkitehtuuri-järjestelmien valmistaminen on uusien kehitysvälineiden myötä helpottunut, saattaa toisissa tilanteissa olla hyödyllisempää pysyä asiakas/palvelin –arkkitehtuurissa tai jopa toteuttaa uusia asiakas/palvelin –arkkitehtuuriin pohjautuvia järjestelmiä. Monitasoarkkitehtuuri-järjestelmien suunnittelu ja toteuttaminen vaatii usein kaksitasoarkkitehtuuria enemmän teknistä osaamista. Varsinkin pienille käyttäjämäärille rakennetuissa järjestelmissä monitasoarkkitehtuurin tarjoamia ominaisuuksia ei pystytä aina täysin hyödyntämään.

Microsoft .NET tarjoaa työvälineet hajautettujen sovellusten rakentamiseksi monitasoarkkitehtuuriin. .NET tarjoaa valmiit komponentit sovellusten väliseen kommunikointiin, verkkopalvelujen julkaisemiseen sekä www- tai windows-käyttöliittymien rakentamiseen

.NET tuo Microsoftin kehitysvälineitä käyttäville paljolti samoja ominaisuuksia kuin esimerkiksi Java-puolella on ollut käytössä jo jonkin aikaa. .NET perustuu pitkälti XML-viestien käyttämiseen sovellusten välisessä kommunikoinnissa, joka osaltaan mahdollistaa .NET sovellusten kutsumisen muistakin kuin Microsoftin alustoilta. Itse .NET-sovellukset kuitenkin toimivat ainakin toistaiseksi ainoastaan Windows-ympäristössä. .NET tulee luultavasti olemaan paljon käytetty ratkaisu Windows-ympäristöön rakennetuissa sovelluksissa, mutta Javan alustariippumattomuus tuo Javalle vahvan kilpailuedun teknologioita valittaessa.

7 Lähteet

And01, Ron Anderson, MICROSOFT LOOKS TO SNAG DEVELOPERS, DOLLARS IN ITS .NET, Network Computing, 5.3.2001, Vol. 12 Issue 5, sivut 87 – 90

Bal00, Keith Ballinger, Jonathan Hawkins, Pranish Kumar, SOAP in the Microsoft .NET Framework and Visual Studio.NET, Microsoft Corporation, Marraskuu 2000

Ble00, Blexrud, Botniker, Crossland, Esposito, Hales, Hankinson, Honnaya, Huckaby, Kirstich, Lhotka, Loesgen, Mohr, Robinson, Rofail, Sherrell, Short, Wahlin, Lee, Professional Windows DNA, Wrox Press, 2000

Box00a, Don Box, A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages, MSDN Magazine, Maaliskuu 2000

Box00b, Don Box, DevelopMentor, David Ehnebuske, IBM, Gopal Kakivaya, Microsoft, Andrew Layman, Microsoft, Noah Mendelsohn, Lotus Development Corp., Henrik Frystyk Nielsen, Microsoft, Satish Thatte, Microsoft, Dave Winer, UserLand Software, Inc., Simple Object Access Protocol (SOAP) 1.1, W3C Note, 08.05.2000

Bro01, Keith Brown, Security in .NET: Enforce Code Access Rights with the Common Language Runtime, MSDN Magazine, Helmikuu 2001

Con00, Conard, Dengler, Francis, Glynn, Harvey, Hollis, Ramachandran, Schenken, Short, Ullman, Introducing .NET, Wrox Press, 2000

Dar97a, Darleen Sadoski, Two Tier Software Architectures, Carnegien Mellon Software Engineering Institute, 10.1.1997

Dar97b, Darleen Sadoski, Client/Server Software Architectures--An Overview, Carnegien Mellon Software Engineering Institute, 10.1.1997 (2.8.1997)

Dar97c, Darleen Sadoski, Santiago Comella-Dorda, Three Tier Software Architectures, Carnegien Mellon Software Engineering Institute, 10.1.1997 (16.2.2000).

Far00, Jim Farley, Microsoft .NET vs. J2EE: How Do They Stack Up?, http://www.oreilly.com/, 8.11.2000

Gar00a, Yefim Natis, Daryl Plummer, David Smith, Microsoft COM: No Longer a Strategic Choice, Gartner Group, 25. 8. 2000

Gra99, Max P. Grasso, Bharat Gogia, Hoa Nguyen, Application servers unmasked, ADT Mag, Lokakuu 1999

How01, Rob Howard, Web Services with ASP.NET, Microsoft Corporation, 22.2.2001

Hol00, Billy Hollins, 10 Ways to Prepare for VB.NET, Visual Basic Programmer's Journal, Joulukuu 2000 vol. 10 No. 14, sivut 62 – 65.

Hol01a, Billy Hollins, 10 More Ways to Prepare for VB.NET, Visual Basic Programmer's Journal, Maaliskuu 2001 vol. 11 No. 3, sivut 42 – 43, 46 - 47.

Joh00, Stuart J. Johnston, The Future of COM+ — Microsoft's .NET Revealed, XML magazine 2000

Lef01, Alain Lefebvre, The True Nature of Web Services, TrendMarkers, Toukokuu 2001

Man01, Marko Mannila, Bill Gates kirkasti jätin Internet-strategiaa, ITviikko, Nro 13/2001, sivu 12.

Mor99, Jim Morey, IIS 4.0 and 5.0 Authentication Methods Chart, Microsoft Corporation 19.7.1999

MS00b, The Transition from Visual Basic 6.0 to Visual Basic.NET, Microsoft Corporation

MS00c, Preparing Your Visual Basic 6.0 Applications for the Upgrade to Visual Basic.NET, Microsoft Corporation, lokakuu 2000

MS00d, .NET Framework Common Language Runtime Architecture Version 1.9 Final, 8.6.2000

MS00e, .NET Framework Developer's Guide, Microsoft Corporation 2000

MS01a, About .NET Security, Microsoft Corporation 2001

Obe00, Piet Obermeyer, Jonathan Hawkins, Microsoft .NET Remoting: A Technical Overview, Microsoft Corporation, Marraskuu 2000

Pat00, Ted Pattison, COM+ Overview for Microsoft Visual Basic Programmers, Microsoft Corporation, Helmikuu 2000

Pla01c, David S. Platt, Web Services: Building Reusable Web Components with SOAP and ASP.NET, MSDN Magazine, Helmikuu 2001

Ric00a, Jeffrey Richter, Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web, MSDN Magazine, Syyskuu 2000

Ric00b, Jeffrey Richter, Part 2: Microsoft .NET Framework Delivers the Platform for an Integrated, Service-Oriented Web, MSDN Magazine, Lokakuu 2000

Ric01a, Jeffrey Richter, .NET Framework: Building, Packaging, Deploying, and Administering Applications and Types, MSDN Magazine, Helmikuu 2001

Ric01b, Jeffrey Richter, .NET Framework: Building, Packaging, Deploying, and Administering Applications and Types—Part 2, MSDN Magazine, Maaliskuu 2001

Ris01, Rissanen Olli-Pekka, Rotsten Petri, Microsoft .NETin jäljillä, Tietokone, 4/2001, sivut 74 – 75.

Sch96, George Schussel, Client/Server: Past, Present and Future, 1996

Tra00, Brian E. Travis, XML and SOAP Programming for BizTalk Servers, Microsoft Press 2000

8 Liitteet

8.1 cArvopaperi

```
'cArvopaperi-luokka, toimii asiakas/palvelin –arkkitehtuurissa
'asiakassovelluksessa tai monitasoarkkitehtuurissa
'sovelluslogiikkakerroksella.
'Visual Studio .NET:n tekemä ohjelmakoodi valkealla pohjalla,
'itse kirjoitettu harmaalla.

Imports System.Collections

Public Class cArvopaperi
    Inherits System.Object

    Private strAPTunnus As String
    Private strCompany As String
    Private strLocation As String

    Public Sub New()

    End Sub

    Property APTunnus() As String
        Get
            APTunnus = strAPTunnus
        End Get
        Set
            APTunnus = Value
        End Set
    End Property

    Property APSijainti() As String
        Get
            APSijainti = strLocation
        End Get
        Set
            APSijainti = Value
        End Set
    End Property

    Property APNimi() As String
        Get
            APNimi = strCompany
        End Get
        Set
            APNimi = Value
        End Set
    End Property

    Public Sub HaeTiedot(ByVal strTunnus As String)
        Dim Tietokantahaku As New cDatabase()
        Dim aTulos As New ArrayList()
```

```
        aTulos = Tietokantahaku.SuoritaSQL("SELECT * FROM ARVOPAPERIT WHERE
TUNNUS ='" & strTunnus & "'")
        strAPTunnus = aTulos.Item(0).ToString
        strCompany = aTulos.Item(1).ToString
        strLocation = aTulos.Item(2).ToString
    End Sub
End Class
```

8.2 cArvopaperit

```
'cArvopaperit-luokka pitää kirjaa yksittäisistä cArvopaperi-olioista.
'Toimii asiakas/palvelin –arkkitehtuurissa asiakassovelluksessa tai
'monitasoarkkitehtuurissa 'sovelluslogiikkakerroksella.
'Visual Studio .NET:n tekemä ohjelmakoodi valkealla pohjalla,
'itse kirjoitettu harmaalla.

Public Class cArvopaperit

    Dim colArvopaperit As New System.Collections.ArrayList()

    'Ladataan arvopaperi tunnuksen perusteella
    Public Sub LataaArvopaperi(ByVal APTunnus As String)

        Dim I As Integer
        Dim objArvopaperi As cArvopaperi

        colArvopaperit.Clear()
        objArvopaperi = New cArvopaperi()
        objArvopaperi.HaeTiedot(APTunnus)
        colArvopaperit.Add(objArvopaperi)

    End Sub

    Public ReadOnly Property Arvopaperit() As System.Collections.ArrayList
        Get
            Arvopaperit = colArvopaperit
        End Get
    End Property

End Class
```

8.3 cDatabase

```
'cDatabase luokan avulla suoritetaan tietokantahaut.
'Luokkaa voidaan käyttää suoraan asiakassovelluksesta
'kaksitasoarkkitehtuurissa tai monitasoarkkitehtuurissa se voi olla
'sovelluslogiikan käytettävissä.
'Visual Studio .NET:n tekemä ohjelmakoodi valkealla pohjalla,
'itse kirjoitettu harmaalla.

Imports System.data.SqlClient

Public Class cDatabase
    Inherits System.Object
    Private connection As New SQLConnection()

    Public Sub New()

    End Sub

    Public Function SuoritaSQL(ByVal SQL As String) As ArrayList
        Dim myCommand As SQLCommand
        Dim ConnStr As String
        Dim sqlConn As SqlClient.SqlConnection
        Dim I As Integer
        ConnStr = "server=localhost;uid=sa;pwd=;database=DB1"
        sqlConn = New SqlConnection(ConnStr)
        myCommand = New SqlCommand(SQL, sqlConn)
        myCommand.Connection.Open()
        Dim myReader As SqlDataReader = myCommand.ExecuteReader()
        SuoritaSQL = New ArrayList()
        'Poimitaan kyselyn tulokset
        while myReader.read()
            SuoritaSQL.Add(myreader.Item(i).ToString)
        end while
        Finally
                    myReader.Close()
             sqlConn.Close()
        End try
    End Function

End Class
```

8.4 MainForm.vb

```
'Arvopaperisovelluksen MainForm ohjelmakoodi asiakas/palvelin –
'arkkitehtuurin asiakassovelluksessa
'Visual Studio .NET:n tekemä ohjelmakoodi valkealla pohjalla,
'itse kirjoitettu harmaalla.

Public Class MainForm
    Inherits System.Windows.Forms.Form

#Region " Windows Form Designer generated code "

    Public Sub New()
        MyBase.New()

        'This call is required by the Windows Form Designer.
        InitializeComponent()

        'Add any initialization after the InitializeComponent() call

    End Sub

    'Form overrides dispose to clean up the component list.
    Protected Overloads Overrides Sub Dispose(ByVal disposing As Boolean)
        If disposing Then
            If Not (components Is Nothing) Then
                components.Dispose()
            End If
        End If
        MyBase.Dispose(disposing)
    End Sub
    Friend WithEvents Button1 As System.Windows.Forms.Button
    Friend WithEvents Label1 As System.Windows.Forms.Label
    Friend WithEvents Label2 As System.Windows.Forms.Label
    Friend WithEvents TextBox1 As System.Windows.Forms.TextBox

    'Required by the Windows Form Designer
    Private components As System.ComponentModel.Container

    'NOTE: The following procedure is required by the Windows Form Designer
    'It can be modified using the Windows Form Designer.
    'Do not modify it using the code editor.
    <System.Diagnostics.DebuggerStepThrough()> Private Sub
InitializeComponent()
        Me.Label1 = New System.Windows.Forms.Label()
        Me.Label2 = New System.Windows.Forms.Label()
        Me.Button1 = New System.Windows.Forms.Button()
        Me.TextBox1 = New System.Windows.Forms.TextBox()
        Me.SuspendLayout()
        '
        'Label1
        '
        Me.Label1.Location = New System.Drawing.Point(16, 32)
        Me.Label1.Name = "Label1"
        Me.Label1.Size = New System.Drawing.Size(120, 24)
```

```
        Me.Label1.TabIndex = 1
        Me.Label1.Text = "Arvopaperin tunnus:"
        '
        'Label2
        '
        Me.Label2.Location = New System.Drawing.Point(24, 72)
        Me.Label2.Name = "Label2"
        Me.Label2.Size = New System.Drawing.Size(248, 176)
        Me.Label2.TabIndex = 2
        '
        'Button1
        '
        Me.Button1.Location = New System.Drawing.Point(240, 24)
        Me.Button1.Name = "Button1"
        Me.Button1.Size = New System.Drawing.Size(40, 32)
        Me.Button1.TabIndex = 0
        Me.Button1.Text = "Hae"
        '
        'TextBox1
        '
        Me.TextBox1.Location = New System.Drawing.Point(152, 32)
        Me.TextBox1.Name = "TextBox1"
        Me.TextBox1.Size = New System.Drawing.Size(72, 20)
        Me.TextBox1.TabIndex = 3
        Me.TextBox1.Text = ""
        '
        'MainForm
        '
        Me.AutoScaleBaseSize = New System.Drawing.Size(5, 13)
        Me.ClientSize = New System.Drawing.Size(292, 273)
        Me.Controls.AddRange(New System.Windows.Forms.Control()
{Me.TextBox1, Me.Label2, Me.Label1, Me.Button1})
        Me.Name = "MainForm"
        Me.StartPosition =
System.Windows.Forms.FormStartPosition.CenterScreen
        Me.Text = "Arvopaperi"
        Me.ResumeLayout(False)

    End Sub

#End Region

    Private Sub Button1_Click(ByVal sender As System.Object, ByVal e As
System.EventArgs) Handles Button1.Click
        Dim APt As New cArvopaperit()
        Dim A As New cArvopaperi()
        Dim Info As String
        Dim Tulos As New System.Collections.ArrayList()


        APt.LataaArvopaperi(TextBox1.Text)
        Tulos = APt.Arvopaperit
        Info = ""
        For Each A In Tulos
            Info = "Arvopaperi " & Chr(13) & Chr(10)
            Info = Info & Chr(13) & Chr(10)
```

```
            Info = Info & "Tunnus: " & A.APTunnus
            Info = Info & Chr(13) & Chr(10)
            Info = Info & "Nimi: " & A.APNimi
            Info = Info & Chr(13) & Chr(10)
            Info = Info & "Kauppapaikka: " & A.APSijainti
            Info = Info & Chr(13) & Chr(10)
        Next


        Label2.Text = Info
    End Sub

End Class
```

8.5 WebApplication.aspx

WebApplication.aspx koodi on kokonaisuudessaan Visual Studio .NET:n generoimaa ohjelmoijan rakentaman (drag'n drop) lomakkeen pohjalta

```
<%@ Page Language="vb" AutoEventWireup="false" Codebehind="WebForm1.vb"
Inherits="WebApplication1.WebForm1"%>
<HTML>
          <HEAD>
                    <meta content="Microsoft Visual Studio.NET 7.0"
name="GENERATOR">
                    <meta content="Visual Basic 7.0"
name="CODE_LANGUAGE">
          </HEAD>
          <body ms_positioning="GridLayout">
           <TABLE height="545" cellSpacing="0" cellPadding="0" width="213"
border="0" ms_2d_layout="TRUE">
            <TR vAlign="top">
             <TD width="213" height="545">
              <form id="WebForm1" method="post" runat="server">
               <TABLE height="196" cellSpacing="0" cellPadding="0"
width="533" border="0" ms_2d_layout="TRUE">
                <TR vAlign="top">
                 <TD width="68" height="16">
                 </TD>
                 <TD width="74">
                 </TD>
                 <TD width="2">
                 </TD>
                 <TD width="1">
                 </TD>
                 <TD width="87">
                 </TD>
                 <TD width="46">
                 </TD>
                 <TD width="1">
                 </TD>
                 <TD width="1">
                 </TD>
                 <TD width="140">
                 </TD>
                 <TD width="113">
                 </TD>
                </TR>
                <TR vAlign="top">
                 <TD colSpan="5" height="2">
                 </TD>
                 <TD colSpan="4" rowSpan="2">
                  <asp:textbox id="TextBoxTunnus" runat="server"
Height="23px" Width="158px"></asp:textbox>
                 </TD>
                 <TD rowSpan="2">
```

```
                    <asp:button id="BHae" runat="server" Height="24"
Width="112" Text="Hae tiedot"></asp:button>
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD height="72">
                   </TD>
                   <TD colSpan="4">
                    <asp:label id="Label1" runat="server" Height="19"
Width="135">Arvopaperin tunnus</asp:label>
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD colSpan="2" height="33">
                   </TD>
                   <TD colSpan="8">
                    <asp:label id="LabelTiedot" runat="server" Height="24px"
Width="291px"></asp:label>
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD colSpan="2" height="2">
                   </TD>
                   <TD colSpan="4" rowSpan="2">
                    <asp:Label id="Label2" runat="server" Height="19px"
Width="122px">Tunnus:</asp:Label>
                   </TD>
                   <TD colSpan="4">
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD colSpan="2" height="25">
                   </TD>
                   <TD>
                   </TD>
                   <TD colSpan="2">
                    <asp:Label id="LabelTunnus" runat="server" Height="19px"
Width="133px"></asp:Label>
                   </TD>
                   <TD rowSpan="4">
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD colSpan="8" height="1">
                   </TD>
                   <TD rowSpan="2">
                    <asp:label id="LabelNimi" runat="server" Height="19px"
Width="135px"></asp:label>
                   </TD>
                  </TR>
                  <TR vAlign="top">
                   <TD colSpan="3" height="25">
                   </TD>
                   <TD colSpan="3">
                    <asp:Label id="Label4" runat="server" Height="19px"
Width="116px">Nimi:</asp:Label>
```

```
                    </TD>
                    <TD colSpan="2">
                    </TD>
                   </TR>
                   <TR vAlign="top">
                    <TD colSpan="4" height="20">
                    </TD>
                    <TD colSpan="2">
                     <asp:label id="Label6" runat="server" Height="19px"
Width="122px">Kauppapaikka:</asp:label>
                    </TD>
                    <TD colSpan="3">
                     <asp:label id="LabelKauppapaikka" runat="server"
Height="19px" Width="136px"></asp:label>
                    </TD>
                   </TR>
                    
                  </TABLE>
                 </form>
                </TD>
               </TR>
              </TABLE>
             </body>
</HTML>
```

8.6 ArvopaperiWebService.vb

```
'Toimii www-palvelimella ASP.NET:n alaisuudessa. Web Service:n kautta
'on mahdollista kutsua sovelluslogiikkaa mm. Windows- tai www-
'sovelluksesta.
'Visual Studio .NET:n tekemä ohjelmakoodi valkealla pohjalla,
'itse kirjoitettu harmaalla.



Imports System.ComponentModel
Imports System.Configuration
Imports System.Web.Services
Imports System.Diagnostics
Imports System.Data

<WebService(Namespace:="http://localhost/arvopaperiwebservice", _
 Description:="Arvopaperi Web Service")> _
Public Class ArvopaperiWebService
    Inherits System.Web.Services.WebService

#Region " Web Services Designer Generated Code "

    'Required by the WebServices Designer
    Private components As System.ComponentModel.Container

    Private Sub InitializeComponent()
        'CODEGEN: This procedure is required by the WebServices Designer
        'Do not modify it using the code editor.
        components = New System.ComponentModel.Container()
    End Sub

    Overrides Sub Dispose()
        'CODEGEN: This procedure is required by the WebServices Designer
        'Do not modify it using the code editor.
    End Sub

#End Region


    ' WEB SERVICE EXAMPLE
    ' The HelloWorld() example service returns the string Hello World
    ' To build, uncomment the following lines then save and build the
project
    ' To test, right-click the Web Service's .asmx file and select View in
Browser
    '
    ' Public Function <WebMethod> HelloWorld() As String
    '      HelloWorld = "Hello World"
    ' End Function


    Public Sub New()
        MyBase.New()
```

```
        'CODEGEN: This procedure is required by the WebServices Designer
        'Do not modify it using the code editor.
        InitializeComponent()

        'Add your own initialization code after the InitializeComponent
call
    End Sub

    <WebMethod()> Public Function HaeArvopaperi(ByVal sTunnus As String) As
cArvopaperi
        Dim AP As New cArvopaperi()

        AP.HaeTiedot(sTunnus)

        HaeArvopaperi = AP

    End Function

End Class
```

8.7 ArvopaperiWebService.wsdl

ASP.NET:n generoima Web Service:n kuvaustiedosto.

```
<?xml version="1.0" encoding="utf-8"?>
<definitions xmlns:s="http://www.w3.org/2001/XMLSchema"
xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:s0="http://localhost/arvopaperiwebservice"
targetNamespace="http://localhost/arvopaperiwebservice"
xmlns="http://schemas.xmlsoap.org/wsdl/">
  <types>
    <s:schema attributeFormDefault="qualified"
elementFormDefault="qualified"
targetNamespace="http://localhost/arvopaperiwebservice">
      <s:element name="HaeArvopaperi">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1" name="sTunnus"
nillable="true" type="s:string" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:element name="HaeArvopaperiResponse">
        <s:complexType>
          <s:sequence>
            <s:element minOccurs="1" maxOccurs="1"
name="HaeArvopaperiResult" nillable="true" type="s0:cArvopaperi" />
          </s:sequence>
        </s:complexType>
      </s:element>
      <s:complexType name="cArvopaperi">
        <s:sequence>
          <s:element minOccurs="1" maxOccurs="1" name="APTunnus"
nillable="true" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="APSijainti"
nillable="true" type="s:string" />
          <s:element minOccurs="1" maxOccurs="1" name="APNimi"
nillable="true" type="s:string" />
        </s:sequence>
      </s:complexType>
      <s:element name="cArvopaperi" nillable="true" type="s0:cArvopaperi"
/>
    </s:schema>
  </types>
  <message name="HaeArvopaperiSoapIn">
    <part name="parameters" element="s0:HaeArvopaperi" />
  </message>
  <message name="HaeArvopaperiSoapOut">
    <part name="parameters" element="s0:HaeArvopaperiResponse" />
```

```
  </message>
  <message name="HaeArvopaperiHttpGetIn">
    <part name="sTunnus" type="s:string" />
  </message>
  <message name="HaeArvopaperiHttpGetOut">
    <part name="Body" element="s0:cArvopaperi" />
  </message>
  <message name="HaeArvopaperiHttpPostIn">
    <part name="sTunnus" type="s:string" />
  </message>
  <message name="HaeArvopaperiHttpPostOut">
    <part name="Body" element="s0:cArvopaperi" />
  </message>
  <portType name="ArvopaperiWebServiceSoap">
    <operation name="HaeArvopaperi">
      <input message="s0:HaeArvopaperiSoapIn" />
      <output message="s0:HaeArvopaperiSoapOut" />
    </operation>
  </portType>
  <portType name="ArvopaperiWebServiceHttpGet">
    <operation name="HaeArvopaperi">
      <input message="s0:HaeArvopaperiHttpGetIn" />
      <output message="s0:HaeArvopaperiHttpGetOut" />
    </operation>
  </portType>
  <portType name="ArvopaperiWebServiceHttpPost">
    <operation name="HaeArvopaperi">
      <input message="s0:HaeArvopaperiHttpPostIn" />
      <output message="s0:HaeArvopaperiHttpPostOut" />
    </operation>
  </portType>
  <binding name="ArvopaperiWebServiceSoap"
type="s0:ArvopaperiWebServiceSoap">
    <soap:binding transport="http://schemas.xmlsoap.org/soap/http"
style="document" />
    <operation name="HaeArvopaperi">
      <soap:operation
soapAction="http://localhost/arvopaperiwebservice/HaeArvopaperi"
style="document" />
      <input>
        <soap:body use="literal" />
      </input>
      <output>
        <soap:body use="literal" />
      </output>
    </operation>
  </binding>
  <binding name="ArvopaperiWebServiceHttpGet"
type="s0:ArvopaperiWebServiceHttpGet">
    <http:binding verb="GET" />
    <operation name="HaeArvopaperi">
      <http:operation location="/HaeArvopaperi" />
      <input>
        <http:urlEncoded />
      </input>
      <output>
```

```
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>
  <binding name="ArvopaperiWebServiceHttpPost"
type="s0:ArvopaperiWebServiceHttpPost">
    <http:binding verb="POST" />
    <operation name="HaeArvopaperi">
      <http:operation location="/HaeArvopaperi" />
      <input>
        <mime:content type="application/x-www-form-urlencoded" />
      </input>
      <output>
        <mime:mimeXml part="Body" />
      </output>
    </operation>
  </binding>
  <service name="ArvopaperiWebService">
    <documentation>Arvopaperi Web Service</documentation>
    <port name="ArvopaperiWebServiceSoap"
binding="s0:ArvopaperiWebServiceSoap">
      <soap:address
location="http://iss/arvopaperiwebservice/ArvopaperiWebService.asmx" />
    </port>
    <port name="ArvopaperiWebServiceHttpGet"
binding="s0:ArvopaperiWebServiceHttpGet">
      <http:address
location="http://iss/arvopaperiwebservice/ArvopaperiWebService.asmx" />
    </port>
    <port name="ArvopaperiWebServiceHttpPost"
binding="s0:ArvopaperiWebServiceHttpPost">
      <http:address
location="http://iss/arvopaperiwebservice/ArvopaperiWebService.asmx" />
    </port>
  </service>
</definitions>
```

8.8 ArvopaperiWebService.vb

```
'ArvopaperiWebServicestä Visual Studio .NET:n generoima proxy-luokka.
'Tämän luokan kautta asiakassovelluksesta hoidetaan kutsut Web Service:en.
'Ohjelmoijan ei periaatteessa tarvitse tietää/välittää, että kutsut
'hoidetaan verkon yli.
'------------------------------------------------------------------------
----
' <autogenerated>
'     This code was generated by a tool.
'     Runtime Version: 1.0.2914.16
'
'     Changes to this file may cause incorrect behavior and will be lost if
'     the code is regenerated.
' </autogenerated>
'------------------------------------------------------------------------
----

Option Strict Off
Option Explicit On

Imports System
Imports System.Diagnostics
Imports System.Web.Services
Imports System.Web.Services.Protocols
Imports System.Xml.Serialization

Namespace iss


<System.Web.Services.WebServiceBindingAttribute(Name:="ArvopaperiWebService
Soap", [Namespace]:="http://localhost/arvopaperiwebservice")>  _
    Public Class ArvopaperiWebService
        Inherits System.Web.Services.Protocols.SoapHttpClientProtocol

        <System.Diagnostics.DebuggerStepThroughAttribute()>  _
        Public Sub New()
            MyBase.New
            Me.Url =
"http://iss/arvopaperiwebservice/ArvopaperiWebService.asmx"
        End Sub

        <System.Diagnostics.DebuggerStepThroughAttribute(),  _

System.Web.Services.Protocols.SoapDocumentMethodAttribute("http://localhost
/arvopaperiwebservice/HaeArvopaperi",
RequestNamespace:="http://localhost/arvopaperiwebservice",
ResponseNamespace:="http://localhost/arvopaperiwebservice",
Use:=System.Web.Services.Description.SoapBindingUse.Literal,
ParameterStyle:=System.Web.Services.Protocols.SoapParameterStyle.Wrapped)>
_
        Public Function HaeArvopaperi(ByVal sTunnus As String) As
cArvopaperi
```

```
            Dim results() As Object = Me.Invoke("HaeArvopaperi", New
Object() {sTunnus})
            Return CType(results(0),cArvopaperi)
        End Function

        <System.Diagnostics.DebuggerStepThroughAttribute()>  _
        Public Function BeginHaeArvopaperi(ByVal sTunnus As String, ByVal
callback As System.AsyncCallback, ByVal asyncState As Object) As
System.IAsyncResult
            Return Me.BeginInvoke("HaeArvopaperi", New Object() {sTunnus},
callback, asyncState)
        End Function

        <System.Diagnostics.DebuggerStepThroughAttribute()>  _
        Public Function EndHaeArvopaperi(ByVal asyncResult As
System.IAsyncResult) As cArvopaperi
            Dim results() As Object = Me.EndInvoke(asyncResult)
            Return CType(results(0),cArvopaperi)
        End Function
    End Class

    Public Class cArvopaperi

        Public APTunnus As String

        Public APSijainti As String

        Public APNimi As String
    End Class
End Namespace
```

8.9 ILDASM.EXE:llä käännetty MSIL-koodi

```
   //Arvopaperi-sovelluksen .exe-tiedostosta (debug-käännös) ILDASM.exe:llä
   //käännetty Button1_Click –metodin lähdekoodi.
.method private instance void  Button1_Click(object sender,
                                             class
[mscorlib]System.EventArgs e) cil managed
{
  // Code size       232 (0xe8)
  .maxstack  3
  .locals init ([0] class WindowsApplication1.cArvopaperi A,
           [1] class WindowsApplication1.cArvopaperit APt,
           [2] string Info,
           [3] class [mscorlib]System.Collections.ArrayList Tulos,
           [4] class [mscorlib]System.Collections.IEnumerator _Vb_t_ref_0)
  .language '{3A12D0B8-C26C-11D0-B442-00A0244A1DD2}', '{994B45C4-E6E9-11D2-
903F-00C04FA302A1}', '{00000000-0000-0000-0000-000000000000}'
// Source File 'D:\Gradu\projektit\WindowsApplication1\MainForm.vb'
//000088:     Private Sub Button1_Click(ByVal sender As System.Object,
ByVal e As System.EventArgs) Handles Button1.Click
  IL_0000:  nop
//000089:         Dim APt As New cArvopaperit()
  IL_0001:  newobj     instance void
WindowsApplication1.cArvopaperit::.ctor()
  IL_0006:  stloc.1
//000090:         Dim A As New cArvopaperi()
  IL_0007:  newobj     instance void
WindowsApplication1.cArvopaperi::.ctor()
  IL_000c:  stloc.0
//000091:         Dim Info As String
//000092:         Dim Tulos As New System.Collections.ArrayList()
  IL_000d:  newobj     instance void
[mscorlib]System.Collections.ArrayList::.ctor()
  IL_0012:  stloc.3
//000093:
//000094:
//000095:         APt.LataaArvopaperi(TextBox1.Text)
  IL_0013:  ldloc.1
  IL_0014:  ldarg.0
  IL_0015:  callvirt   instance class
[System.Windows.Forms]System.Windows.Forms.TextBox
WindowsApplication1.MainForm::get_TextBox1()
  IL_001a:  callvirt   instance string
[System.Windows.Forms]System.Windows.Forms.TextBoxBase::get_Text()
  IL_001f:  callvirt   instance void
WindowsApplication1.cArvopaperit::LataaArvopaperi(string)
  IL_0024:  nop
//000096:         Tulos = APt.Arvopaperit
  IL_0025:  ldloc.1
  IL_0026:  callvirt   instance class
[mscorlib]System.Collections.ArrayList
WindowsApplication1.cArvopaperit::get_Arvopaperit()
  IL_002b:  stloc.3
//000097:         Info = ""
  IL_002c:  ldstr      ""
```

```
  IL_0031:  stloc.2
//000098:         For Each A In Tulos
  IL_0032:  ldloc.3
  IL_0033:  callvirt   instance class
[mscorlib]System.Collections.IEnumerator
[mscorlib]System.Collections.ArrayList::GetEnumerator()
  IL_0038:  stloc.s    _Vb_t_ref_0
  IL_003a:  br         IL_00cd
  IL_003f:  ldloc.s    _Vb_t_ref_0
  IL_0041:  callvirt   instance object
[mscorlib]System.Collections.IEnumerator::get_Current()
  IL_0046:  castclass  WindowsApplication1.cArvopaperi
  IL_004b:  stloc.0
//000099:             Info = "Arvopaperi " & Chr(13) & Chr(10)
  IL_004c:  ldstr      "Arvopaperi \r\n"
  IL_0051:  stloc.2
//000100:             Info = Info & Chr(13) & Chr(10)
  IL_0052:  ldloc.2
  IL_0053:  ldstr      "\r"
  IL_0058:  ldstr      "\n"
  IL_005d:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_0062:  stloc.2
//000101:             Info = Info & "Tunnus: " & A.APTunnus
  IL_0063:  ldloc.2
  IL_0064:  ldstr      "Tunnus: "
  IL_0069:  ldloc.0
  IL_006a:  callvirt   instance string
WindowsApplication1.cArvopaperi::get_APTunnus()
  IL_006f:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_0074:  stloc.2
//000102:             Info = Info & Chr(13) & Chr(10)
  IL_0075:  ldloc.2
  IL_0076:  ldstr      "\r"
  IL_007b:  ldstr      "\n"
  IL_0080:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_0085:  stloc.2
//000103:             Info = Info & "Nimi: " & A.APNimi
  IL_0086:  ldloc.2
  IL_0087:  ldstr      "Nimi: "
  IL_008c:  ldloc.0
  IL_008d:  callvirt   instance string
WindowsApplication1.cArvopaperi::get_APNimi()
  IL_0092:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_0097:  stloc.2
//000104:             Info = Info & Chr(13) & Chr(10)
  IL_0098:  ldloc.2
  IL_0099:  ldstr      "\r"
  IL_009e:  ldstr      "\n"
```

```
  IL_00a3:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_00a8:  stloc.2
//000105:             Info = Info & "Kauppapaikka: " & A.APSijainti
  IL_00a9:  ldloc.2
  IL_00aa:  ldstr      "Kauppapaikka: "
  IL_00af:  ldloc.0
  IL_00b0:  callvirt   instance string
WindowsApplication1.cArvopaperi::get_APSijainti()
  IL_00b5:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_00ba:  stloc.2
//000106:             Info = Info & Chr(13) & Chr(10)
  IL_00bb:  ldloc.2
  IL_00bc:  ldstr      "\r"
  IL_00c1:  ldstr      "\n"
  IL_00c6:  call       string [mscorlib]System.String::Concat(string,
                                                              string,
                                                              string)
  IL_00cb:  stloc.2
//000107:         Next
  IL_00cc:  nop
  IL_00cd:  ldloc.s    _Vb_t_ref_0
  IL_00cf:  callvirt   instance bool
[mscorlib]System.Collections.IEnumerator::MoveNext()
  IL_00d4:  brtrue     IL_003f
//000108:
//000109:
//000110:         Label2.Text = Info
  IL_00d9:  ldarg.0
  IL_00da:  callvirt   instance class
[System.Windows.Forms]System.Windows.Forms.Label
WindowsApplication1.MainForm::get_Label2()
  IL_00df:  ldloc.2
  IL_00e0:  callvirt   instance void
[System.Windows.Forms]System.Windows.Forms.Control::set_Text(string)
  IL_00e5:  nop
//000111:     End Sub
  IL_00e6:  nop
  IL_00e7:  ret
} // end of method MainForm::Button1_Click
```