Integrering med .NET

Examensarbete 20p

Jonas Forsberg

2003-01-22

Abstract

Integration of different systems is a common problem for most companies. This thesis investigates the possibility to use Microsoft's new platform .NET, as a platform for integration. The following technologies are investigated and in many cases implemented: SOAP, secure web services, communication with Java, communication with COM, transactions, .NET Remoting, UDDI and loosely coupled systems, and finally distribution of .NET applications and .NET Framework. The results show that .NET in most cases handles and support these technologies very well. .NET can therefore be considered well suited as a platform for integration.

Sammanfattning

Integrering av olika system är ett vanligt problem hos de flesta företag. Denna rapport undersöker möjligheten att använda Microsofts nya utvecklingsplattform .NET, som en integreringsplattform. I separata kapitel undersöks och i många fall implementeras följande teknologier: SOAP, säkra webbtjänster, kommunikation med Java, kommunikation med COM, transaktioner, .NET Remoting, UDDI och löst kopplade system, samt distribution av .NET applikationer och .NET Framework. Resultaten visar att .NET i de flesta fall hanterar och stödjer dessa teknologier på ett mycket bra sätt. .NET kan därför anses vara väl lämpad som integreringsplattform.

Innehåll

1 Inledning

Detta examensarbete är den sista delen av mina studier till civilingenjör inom teknisk datavetenskap, på Umeå Universitet. Examensarbetet utförs på uppdrag av DataVis AB i Örnsköldsvik.

1.1 Bakgrund

Många företag har gjort stora investeringar för att ha ett IT-stöd i sin organisation. Problemet är ofta att dessa lösningar har byggts med för hårt kopplade system. Detta upptäcks oftast inte förrän ny funktionalitet skall införas. För att erhålla den nya funktionaliteten måste därför mycket av utvecklingskostanden läggas på att modifiera eller helt ersätta befintlig logik och detta anses onödigt.

Med Microsofts nya utvecklingsplattform *.NET*, skall det vara möjligt att minimera tiden för modifiering av befintlig funktionalitet och istället koncentrera sig på att införa den nya. .NET ska med andra ord kunna användas som en integreringsplattform, för att sammanlänka olika system.

1.2 Målsättning

För att kunna använda .NET som en integreringsplattform, måste följande punkter undersökas och klargöras i rapporten:

- SOAP protokollet
- Hur skapas säkra webbtjänster?
- Kommunikation med Java
- Kommunikation med COM
- Transaktioner i .NET
- .NET Remoting
- Kan UDDI användas för att skapa löst kopplade system?
- Distribution av .NET applikationer och .NET Framework

Punkterna skall prioriteras i fallande ordning och enklare implementationer skall tas fram som demonstrationsexempel i de fall där det visar sig nödvändigt för att förklara eller bekräfta teorin.

1.3 Upplägg

Rapporten är strukturerad efter punkterna ovan, där varje punkt utgör ett kapitel. För bästa förståelse bör rapporten läsas i kapitlens nummerordning. Varje kapitel avslutas med en sektion där jag delger mina personliga slutsatser av det som har presenterats i kapitlet. Kapitel 10 innehåller till sist en diskussion, där de olika slutsatserna sammanfattas och knyts samman. I Bilaga A finns en ordlista där svåra ord och förkortningar förklaras.

2 SOAP

Simple Object Access Protocol (SOAP) är ett textbaserat protokoll tänkt att användas för informationsutbyte i en decentraliserad och distribuerad miljö. Användningsområdena är stora och sträcker sig från system som använder *Message passing* till *RPC (Remote Procedure Call)* system. SOAP är baserat på *XML (eXtensible Markup Language)* och består av följande tre delar:

1. Ett *Envelope* (kuvert) som beskriver innehållet i meddelandet, vem som ska behandla det och om det är frivilligt eller obligatoriskt att behandla det.
2. Kodningsregler för att uttrycka datatyper som definierats i olika applikationer.
3. Regler för att representera RPC förfrågningar och svar.

SOAP har potentialen att användas i kombination med ett flertal olika protokoll och innehåller därför protokollbindningar som beskriver hur det kan användas tillsammans med dem. Denna rapport kommer att ta upp två kombinationer: SOAP över *HyperText Transfer Protocol (HTTP)*, som är vanligast och används av dagens webbtjänster, samt SOAP över *HTTP Extension Framework (HTTP-EF)*. Bindningar för SOAP över *Simple Mail Transfer Protocol (SMTP)* är för närvarande under utveckling.

World Wide Web Consortium (W3C) mottog SOAP 1.1 specifikationen den 8 maj 2000, men utvecklig av denna pågår ständigt och det senaste förslaget, SOAP 1.2, 26 juni 2002, väntar på att W3C ska anta det som rekommendation.

Det brukar sägas att SOAP är ett ”lättviktsprotokoll” och det styrks av två av huvudmålen för designen av SOAP: enkelhet samt utbyggbarhet. Ett flertal av funktionerna hos traditionella Message passing system och RPC system saknas hos SOAP, t.ex.: distribuerad *Garbage Collection (GC)*, objektreferenser samt transaktions-hantering. SOAP är ändå en mycket kvalificerad medtävlare till dessa system, vilket kommer att visas i följande sektioner. [1]

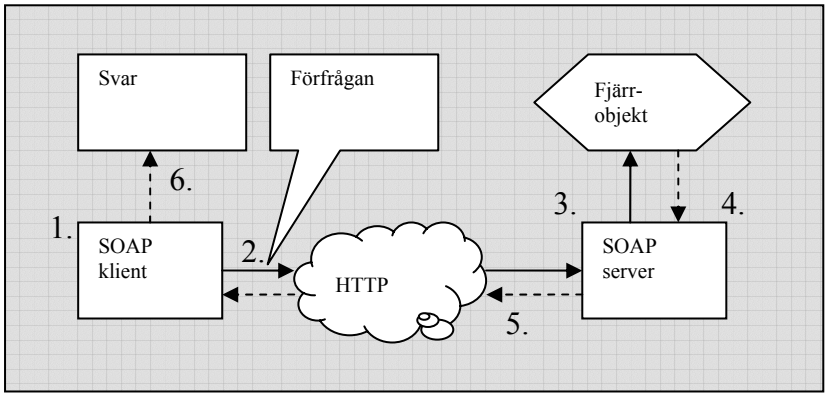
2.1 Bakgrund

Det finns i huvudsak två modeller för kommunikation mellan distribuerade system: Förfrågan (request) och svar (response) system som RPC samt Message passing system. Message passing system är asynkrona och tillåter att meddelanden skickas när som helst, även när mottagaren är offline. RPC system är däremot synkrona och kräver normalt att applikationen väntar på svaret innan den fortsätter. Designen av SOAP gör att det kan användas till båda dessa system, även om RPC är klart vanligast.

Låt oss först titta närmare på några problem som finns med existerande RPC system. De två vanligaste standarderna är Microsofts *Distributed Component Object Model (DCOM)* samt *Internet Inter-Orb Protocol (IIOP)*, som används i implementationer av *Common Object Request Broker Architecture (CORBA)*. Båda dessa standarder fungerar bra enskilt, men inte alls tillsammans. Detta gör att en klient inte kan göra ett godtyckligt anrop till en server utan att först ta reda på vilken standard den använder. DCOM fungerar dessutom bäst när alla datorer i systemet kör Windows (det finns DCOM implementationer för andra plattformar), medan IIOP har minst problem då alla

datorer i systemet använder en CORBA implementation från samma tillverkare. Detta kan vara acceptabelt i ett mindre intranät, men då kommunikationen ska gå över Internet är det omöjligt att förutsätta en specifik plattform. Eftersom både DCOM och IIOP skickar data i binärt format blir det dessutom ofta problem att ta sig igenom brandväggar. Det är här SOAP kommer in i bilden.

SOAP är helt uppbyggt kring redan existerande och väl beprövad teknologi: HTTP används som transportkanal (även om andra transportkanaler som sagt är möjliga), medan data och funktionalitet representeras av XML. Alla plattformar som kan tolka HTTP, samt har en XML parser som kan tyda SOAP blir plötsligt SOAP servrar och det är i stort sett alla plattformar. Användningen av HTTP (port 80) och XML (textbaserat), gör dessutom att SOAP, till skillnad från CORBA och DCOM, utan problem tar sig igenom de flesta brandväggar.



Figur 1: Förfrågan och svar med SOAP över HTTP

Figur 1 visar hur vägen för en förfrågan och ett svar med SOAP över HTTP ser ut. Klienten och servern kan i detta fall vara helt skilda plattformar. Applikationen som agerar SOAP klient skapar först ett SOAP meddelande innehållande information för att anropa ett objekt på en fjärrmaskin (1). Detta meddelande kapslas in i en HTTP förfrågan som skickas över en vanlig HTTP uppkoppling (2). En lyssnande applikation, ofta en webbserver som letar efter SOAP meddelanden inuti de anländande HTTP paketen, använder informationen i meddelandet för att anropa rätt objekt och metod, med eventuella parametrar (3). Objektet utför den begärda funktionen och returnerar resultatet till SOAP servern, som paketerar det i ett SOAP meddelande (4). Detta meddelande kapslas i sin tur in i ett HTTP svar, som skickas tillbaka till klienten på samma HTTP uppkoppling som tidigare (5). SOAP klienten väntar på ett HTTP svar och när det anländer plockas SOAP meddelandet, som innehåller svaret på metod-anropet, ut för vidare behandling (6). Figur 2 och Figur 3 visar mer detaljerade exempel på förfrågan- och svarsmeddelanden.

Framtida versioner av CORBA och DCOM (se .NET Remoting i kapitel 7), kommer att stödja SOAP. Java gör det redan, se kapitel 4. Här öppnar sig stora möjligheter till integration mellan olika plattformar och språk, men eftersom SOAP är ett relativt nytt protokoll finns det tyvärr en del interoperabilitetsproblem mellan olika SOAP implementationer, se 2.7. [2]

2.2 SOAP över HTTP

Det vanligaste kommunikationsprotokollet för SOAP är som sagt HTTP. RPC med SOAP passar väl in på HTTP:s förfrågan och svarsmodell, som vi såg i Figur 1. I följande sektioner ska vi titta närmare på detta.

2.2.1 HTTP förfrågan

SOAP förfrågningar skickas vanligtvis med HTTP POST och i huvudfältet på meddelandet skall attributet *SOAPAction* finnas för att tala om syftet med förfrågan. Värdet är en *URI* som identifierar syftet. Om SOAPAction innehåller en tom sträng antas URI:n vara samma som HTTP förfrågans URI och om värdet saknas är inte syftet med SOAP meddelandet tillgängligt. Brandväggar kan använda attributet SOAPAction för att effektivt filtrera HTTP förfrågningar som innehåller SOAP meddelanden. I Figur 2 visas ett exempel på ett SOAP meddelande som är inbäddat i en HTTP POST förfrågan. Meddelandet är ett metodanrop (RPC), som går till ASP.NET webbtjänsten *myWebService.asmx* och dess publika metod: *int GetPris(string strVarunamn, int intAntal)*. Under figuren förklaras varje rad i meddelandet. [1]

```
(001)  POST /myWebService/myWebService.asmx HTTP/1.1
(002)  Host: localhost
(003)  Content-Type: text/xml; charset=utf-8
(004)  Content-Length: 227
(005)  SOAPAction: "http://localhost/GetPris"
(006)
(007)  <?xml version="1.0" encoding="utf-8"?>
(008)  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
(009)                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(010)                 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
(011)    <soap:Body>
(012)      <GetPris xmlns="http://localhost">
(013)        <strVarunamn>Fotboll</strVarunamn>
(014)        <intAntal>4</intAntal>
(015)      </GetPris>
(016)    </soap:Body>
(017)  </soap:Envelope>
```

Figur 2: SOAP förfrågan via HTTP

På rad (001) anropas ASP.NET webbtjänsten myWebService.asmx via HTTP POST version 1.1. Rad (002) anger hostadressen som i detta fall är *localhost*. Ett krav är att HTTP sätter Content-Type till *text/xml* när SOAP meddelanden skickas. Detta görs på rad (003), där även teckenuppsättningen sätts till *utf-8*. Rad (004) anger längden i bytes för HTTP meddelandets payload, som finns på rad (007)-(017). Rad (005) innehåller attributet SOAPAction, som beskrevs ovan. I detta fall innehåller det namnrymden ”http://localhost/”, samt namnet på metoden som ska anropas: GetPris. Rad (006) är tom och den markerar att huvudfältet är slut och att det som följer är payload.

Rad (007) talar om att ett XML dokument följer och att det använder XML version 1.0 och utf-8 teckenkodning. På rad (008) börjar själva SOAP meddelandet med elementet *Envelope*. Envelope måste alltid vara rotelement och kan innehålla attributet *encodingStyle*, se 2.6.3. I detta fall innehåller det definitionen av tre namnrymder som används för att undvika namnkrockar mellan XML identifierare, SOAP identifierare samt applikationsspecifika identifierare. Namnrymder specificerar även med hjälp av

XSD (XML Schema Definition) upp strukturen för SOAP dokument. XSD schemat under den obligatoriska namnrymden *http://schemas.xmlsoap.org/soap/envelope/* bestämmer t.ex. att ett korrekt *Envelope* måste innehålla ett underelement *Body*, medan underelementet *Header* är valfritt, se 2.4. Det bestämmer även versionen på SOAP meddelandet. SOAP 1.1 associeras med denna namnrymd och om den saknas eller är felaktig ska mottagaren generera ett felmeddelande av typen *VersionMismatch*, se 2.5.

På rad (011) återfinns det obligatoriska elementet *Body*, som innehåller SOAP meddelandets payload, i detta fall metodnamn och parametrar. På rad (012) återfinns ett element med samma namn som metoden som ska anropas. Detta element definierar även en default namnrymd, ”http://localhost”, som metoden tillhör (samma som SOAPAction). De följande två elementen, *strVarunamn* och *intAntal*, har samma namn som metodens parametrar och innehåller värdet av dessa. I detta fall anropas GetPris med värdena ”Fotboll” och 4. Tanken är att denna enkla metod ska beräkna kostnaden för fyra fotbollar, inklusive mängdrabatt. Exemplet i Figur 2 ovan visar hur enkelt det är att göra ett RPC anrop till en webbtjänst med hjälp av SOAP över HTTP. Svaret på metodanropet visas i Figur 3.

2.2.2 HTTP svar

SOAP över HTTP följer semantiken för HTTP:s statuskoder. En statuskod 2xx indikerar t.ex. att klientens förfrågan inklusive SOAP delen togs emot och behandlades på ett lyckat sätt. Om behandlingen av ett SOAP meddelande misslyckas ska servern svara med ett HTTP 500 ”Internal Server Error” meddelande. Detta ska i sin tur inkludera ett SOAP meddelande innehållande elementet *Fault*, som beskriver felet, se 2.5. I Figur 3 nedan visas ett SOAP meddelande som är inbäddat i ett HTTP svar. Meddelandet innehåller returvärdet från metodanropet som gjordes i Figur 2. Under figuren förklaras varje rad som skiljer sig från Figur 2. [1]

```
(001)  HTTP/1.1 200 OK
(002)  Content-Type: text/xml; charset=utf-8
(003)  Content-Length: 215
(004)
(005)  <?xml version="1.0" encoding="utf-8"?>
(006)  <soap:Envelope xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/"
(007)                 xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
(008)                 xmlns:xsd="http://www.w3.org/2001/XMLSchema">
(009)    <soap:Body>
(010)      <GetPrisResponse xmlns="http://localhost">
(011)        <GetPrisResult>1400</GetPrisResult>
(012)      </GetPrisResponse>
(013)    </soap:Body>
(014)  </soap:Envelope>
```

Figur 3: SOAP svar via HTTP

Rad (001) beskriver att meddelandet är OK, dvs., att förfrågan behandlades och returnerades på ett lyckat sätt. På rad (010) finns ett element med den anropade metodens namn inklusive ordet ”Response”, som har lagts till sist i namnet. Denna benämning är inte ett krav, men däremot en rekommendation från W3C. Rad (011) innehåller ett element med metodens returvärde, som i detta fall är 1400. Element med returvärden brukar benämnas med metodens namn och ordet ”Result” sist i namnet. Fyra fotbollar kostar alltså 1400 kronor.

2.3 SOAP över HTTP-EF

Det går att begränsa vilka meddelanden som kommer igenom en brandvägg eller proxy genom att endast släppa igenom M-POST istället för POST. M-POST är en ny HTTP metod som definieras i *HTTP Extension Framework (HTTP-EF)*. Metoden används när obligatorisk information ska finnas i huvudfältet. I Figur 4 har attributet *SOAPAction* markerats som obligatoriskt med hjälp av prefixet *NNNN* och attributet *Man*. [1]

```
M-POST /order HTTP/1.1
Man: "http://schemas.xmlsoap.org/soap/envelope/"; ns=NNNN
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn
NNNN-SOAPAction: "www.fakeorders/order"
.
.
```

Figur 4: Obligatoriskt attribut med hjälp av M-POST i HTTP-EF

2.4 Elementet Header

Elementet *Header* ger möjligheten att utöka funktionaliteten hos ett SOAP meddelande. Några vanliga funktioner som kan implementeras i *Header* är identifikation, se 3.6, transaktioner, betalning, mm. Det finns ännu ingen accepterad standard för detta, utan sändare och mottagare måste komma överens. Elementet *Header* är valfritt, men ska om det används placeras direkt före elementet *Body*. [1]

2.4.1 Attributet actor

Ett SOAP meddelande färdas från sändare till en slutgiltig destination. På vägen kan det passera ett antal mellanliggande SOAP noder, som både kan ta emot och sända SOAP meddelanden. Hela originalmeddelandet behöver inte vara ämnat för slutdestinationen, utan med hjälp av en URI i attributet *actor* kan mottagaren av ett visst underelement i *Header* specificeras, se Figur 5 på nästa sida. De mellanliggande SOAP noderna kan sedan använda denna information för att plocka ut de element som tillhör dem. De skickar sedan vidare det modifierade meddelandet till nästa destination. Om attributet *actor* sätts till URI:n *http://schemas.xmlsoap.org/soap/actor/next* är det aktuella elementet ämnat för den första SOAP nod som tar emot meddelandet. Om attributet *actor* utelämnas antas det att mottagaren är den slutgiltiga destinationen. [1]

2.4.2 Attributet mustUnderstand

Attributet *mustUnderstand* kan användas för att specificera om det ska vara frivilligt eller obligatoriskt för mottagaren att behandla ett visst element i *Header*. Om attributet *mustUnderstand* är satt till 1 måste mottagaren både förstå och behandla elementet korrekt, annars ska ett felmeddelande returneras, se 2.5. Figur 5 på nästa sida visar ett exempel där *Header* innehåller ett underelement ”Transaction”, som måste förstås och behandlas av den SOAP nod som anges av attributet *actor*. [1]

```
<soap:Header>
  <t:Transaction xmlns:t="some-URI"
                 soap:mustUnderstand="1"
                 soap:actor="http://www.node.com/soapnode">
    5
  </t:Transaction>
</soap:Header>
```

Figur 5: Exempel på användning av attributen actor och mustUnderstand

2.5 Elementet Fault

Elementet *Fault* används för att transportera fel- och statusinformation i ett SOAP meddelande. *Fault* måste vara underelement till *Body* och har själv fyra underelement:

faultcode: Används för att identifiera ett fel via en algoritm i mjukvaran hos klienten. SOAP definierar en liten mängd felkoder som täcker in de grundläggande SOAP felen. *VersionMismatch* tyder t.ex. på en felaktig namnrymd för elementet *Envelope*, medan *MustUnderstand* betyder att ett element i *Header*, med attributet *mustUnderstand* satt till 1, inte kunde behandlas, se Figur 6. *faultcode* måste alltid finnas i elementet *Fault*.

faultstring: Detta element presenterar felet på ett för människor lättbegripligt sätt. *faultstring* måste alltid finnas i elementet *Fault*.

faultactor: Används för att visa vilken SOAP nod på meddelandevägen som genererade felet. Detta element måste finnas i elementet *Fault* om noden som genererade felet är en mellanliggande nod och inte slutdestinationen.

detail: Innehåller applikationsspecifik information om felet. Detta element måste endast finnas i elementet *Fault* om innehållet i elementet *Body* inte kunde behandlas korrekt. Detta kan användas vid felsökning eftersom saknaden av elementet *detail* tyder på att felet inte finns i *Body*. [1]

Det är upp till klienterna hur de behandlar elementet *Fault*. När t.ex. en .NET klient tar emot ett *Fault* kapslas det in i klassen *SoapException*, som sedan kan fångas i programkoden. Denna klass har ett antal attribut som kan avläsas för att erhålla information motsvarande den i elementen ovan. [3]

```
HTTP/1.1 500 Internal Server Error
Content-Type: text/xml; charset="utf-8"
Content-Length: nnnn

<soap:Envelope
  xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
   <soap:Body>
       <soap:Fault>
           <faultcode>soap:MustUnderstand</faultcode>
           <faultstring>SOAP Must Understand Error</faultstring>
       </soap:Fault>
   </soap:Body>
</soap:Envelope>
```

Figur 6: Exempel på felmeddelande i SOAP, som genereras då ett element i Header med attributet mustUnderstand satt till 1 inte kunde förstås eller behandlas

2.6 SOAP Encoding

Ett SOAP meddelande kan syntaktiskt kodas på olika sätt. W3C har definierat kodningsregler dels för hur elementet *Body* och dess underelement ska formateras, men även hur parametrarna till en metod ska kodas. Dessa kodningsregler är dock valfria och kan bytas ut efter behag, även om det är rekommenderat att följa W3C:s syntaxregler för största interoperabilitet, se 2.7. W3C:s syntax är väl överensstämmande med den som hittas hos vanliga programmeringsspråk och databaser. Till exempel är en datatyp antingen en enkel typ eller en sammansatt typ.

Web Service Description Language (WSDL), som används för att beskriva gränssnittet för en webbtjänst, definierar två sätt att formatera elementet *Body*: *RPC Encoding* (som är W3C:s specifikation) samt *Document Encoding*. Dessutom definieras två sätt att formatera parametrarna: *Literal* samt *Encoded* (som är W3C:s specifikation). Dessa olika möjligheter diskuteras nedan. Dessutom ges exempel på datatyper som följer *Encoded* syntaxen. [1][4]

2.6.1 RPC Encoding

RPC Encoding formaterar elementet *Body* enligt reglerna från sektion 7 i SOAP specifikationen [1]. Alla parametrar ska t.ex. omges av ett element med samma namn som metoden som ska anropas. Parametrarna i sig skall även de namnges efter sina motsvarigheter i metodhuvudet, se Figur 7. Funktionshuvudet för metoden som anropas i figuren ser ut enligt följande: *Rpc(Address address, bool useZipPlus4)*, där *Address* är en *struct* innehållande tre strängar.

I .NET anges RPC Encoding genom att sätta attributet *SoapRpcMethod* framför den metod det gäller, alternativt sätts attributet *SoapRpcService* framför webbtjänstklassen, vilket påverkar alla metoder. [4]

```
<?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:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:tns="http://www.contoso.com"
xmlns:tnsTypes="http://www.contoso.com/encodedTypes"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns:soap="http://schemas.xmlsoap.org/soap/envelope/">
  <soap:Body soap:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <tns:Rpc>
      <address href="#1" />
      <useZipPlus4>false</useZipPlus4>
    </tns:Rpc>
    <tnsTypes:Address id="1">
      <Street id="2">Kovägen 3</Street>
      <City id="3">Umeå</City>
      <Zip id="4">907 34</Zip>
    </tnsTypes:Address>
  </soap:Body>
</soap:Envelope>
```

Figur 7: RPC formaterat SOAP meddelande med Encoded parametrar

2.6.2 Document Encoding

Vid Document Encoding formateras elementet *Body* enligt beskrivningen i ett XSD schema, som definieras i WSDL dokumentet för den webbtjänst metodanropet gäller. WSDL dokumentet innehåller scheman för både SOAP förfrågan och SOAP svar till metoden. I Figur 8 visas ett exempel på ett SOAP meddelande som är formaterat enligt Document Encoding. Metoden *DocumentWrappedLiteral* tar samma parametrar som den i Figur 7. Notera dock hur olika formateringen av parametrarna är i jämförelse med Figur 7. Detta beror på att i Figur 8 används Literal parametrar.

Document Encoding är default i .NET, men kan ändå specificeras tillsammans med metoden för kodning av parametrar (*Encoded* eller *Literal*), med hjälp av attributen *SoapDocumentMethod* samt *SoapDocumentService*. [4]

```
<?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>
    <DocumentWrappedLiteral xmlns="http://www.contoso.com">
      <MyAddress>
        <Street>Kovägen 3</Street>
        <City>Umeå</City>
        <Zip>907 34</Zip>
      </MyAddress>
      <useZipPlus4>false</useZipPlus4>
    </DocumentWrappedLiteral>
  </soap:Body>
</soap:Envelope>
```

Figur 8: Document formaterat SOAP meddelande med Literal parametrar

2.6.3 Encoded parametrar

Encoded parametrar formateras enligt reglerna i SOAP specifikationen [1], sektion 5. I SOAP meddelandet syns detta på det globala attributet *encodingStyle*, se Figur 7, som sätts att peka på XSD schemat under *http://schemas.xmlsoap.org/soap/encoding/*. Detta attribut kan peka på andra scheman för en alternativ formatering, men av interoperabilitetsskäl är det rekommenderat att använda schemat ovan, se 2.7.

I .NET specificeras användningen av Encoded parametrar genom att sätta egenskapen *Use* för SoapDocumentMethod/Service till *SoapBindingUse.Encoded* eller genom att använda attributet SoapRpcMethod/Service. [4]

2.6.4 Literal parametrar

Literal parametrar kan endast användas tillsammans med Document Encoding, se Figur 8. Här formateras parametrarna enligt ett XSD schema som återfinns i WSDL dokumentet, på samma sätt som för Document Encoding.

Literal parametrar är default i .NET, men kan ändå specificeras genom att sätta egenskapen *Use* för SoapDocumentMethod/Service till *SoapBindingUse.Literal*. [4]

2.6.5 Enkla datatyper

Vid användning av Encoded parametrar enligt rekommendationen ovan är samtliga enkla datatyper som definieras under ”*XML Schema Part 2: Datatypes*” [5] tillgängliga. I Figur 9 visas ett exempel på några enkla datatyper. Först definieras de i ett schema och sedan följer motsvarande XML data. [1]

```
<element name="age" type="int"/>
<element name="height" type="float"/>
<element name="displacement" type="negativeInteger"/>
<element name="color">
  <simpleType base="xsd:string">
    <enumeration value="Green"/>
    <enumeration value="Blue"/>
  </simpleType>
</element>

<age>45</age>
<height>5.9</height>
<displacement>-450</displacement>
<color>Blue</color>
```

Figur 9: Definition och användande av några enkla datatyper

2.6.6 Sammansatta datatyper

Två av de vanligast förekommande sammansatta datatyperna i programmeringsspråk: *struct* och *array*, finns representerade i SOAP. Figur 10 visar definitionen av en struct liknande den som används i Figur 7, medan Figur 11 visar definitionen och användandet av en heltalsarray med plats för tre element. Arrayer definieras under *http://schemas.xmlsoap.org/soap/encoding/* och denna namnrymd måste användas som i exemplet nedan. SOAP stödjer även användandet av flerdimensionella arrayer, glesa arrayer samt överföring av delar av arrayer. [1][4]

```
<s:complexType name="Address" xmlns:s="http://www.w3.org/2001/XMLSchema">
   <s:sequence>
      <s:element minOccurs="1" maxOccurs="1" name="Street type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="City" type="s:string" />
      <s:element minOccurs="1" maxOccurs="1" name="Zip" type="s:string" />
   </s:sequence>
</s:complexType>
```

Figur 10: Definition av en struct

```
<NumberArray soapenc:arrayType="xsd:int[3]"
      xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
      xmlns:xsd="http://www.w3.org/2001/XMLSchema">
   <number>14</number>
   <number>6</number>
   <number>22</number>
</NumberArray>

<element name="NumberArray" type="soapenc:Array"
   xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"/>
```

Figur 11: Definition och användande av en heltalsarray med tre element

2.7 SOAP interoperabilitet

I teorin ska SOAP tillåta problemfri kommunikation mellan alla plattformar som har en SOAP implementation. I praktiken finns det dock ett antal små skillnader mellan olika implementationer som gör att vissa problem kan uppstå. Nedan ges några allmänna tips för ASP.NET webbtjänster, som ökar chanserna för interoperabilitet med SOAP implementationer på andra plattformar. Problemen beror mycket på att SOAP är ett relativt nytt protokoll, men det pågår ständigt testning och utveckling av de olika implementationerna och i framtiden kommer några av tipsen och förmaningarna nedan inte längre vara aktuella. I kapitel 4 visas implementationer som testar SOAP interoperabilitet mellan .NET och Java i praktiken.

Använd RPC Encoding och Encoded parametrar: Alla implementationer stödjer ännu inte Document Encoding och därför är det säkrast att använda RPC.

Erbjud WSDL dokument: Detta gör det lättare för andra att använda webbtjänsten på ett korrekt sätt. Många implementationer, t.ex. Apache SOAP och .NET, använder WSDL filen för att skapa en proxyklass, se nedan.

Använd XSD 2001: Den senaste rekommendationen för XSD scheman finns under *http://www.w3.org/2001/XMLSchema* och denna stöds av de flesta SOAP implementationerna.

Undvik HTTP/GET och HTTP/POST bindningar i WSDL dokumenten: Alla implementationer stödjer inte detta och klarar inte av att skapa proxyklassen när de stöter på dem i WSDL dokumentet. I .NET skapas WSDL dokumentet automatiskt, men genom att sätta in raderna i Figur 12 i filen *web.config* under sektionen <*configuration*><*system.web*>, skapas inte bindningarna till GET och POST. Filen *web.config* används för konfigurering av ASP.NET projekt.

```
<webServices>
    <protocols>
        <remove name="HttpPost" />
        <remove name="HttpGet" />
    </protocols>
</webServices>
```

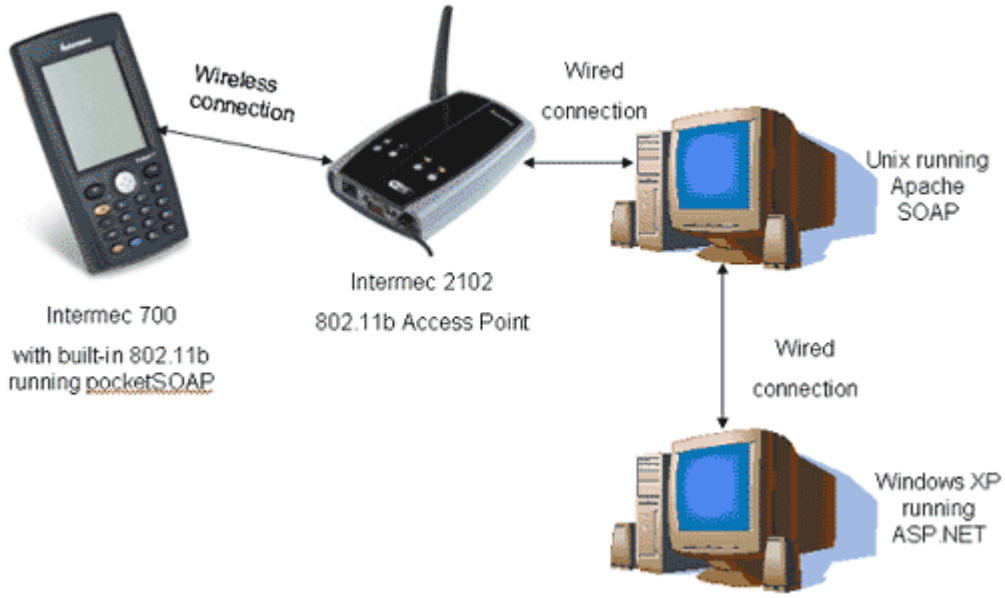
Figur 12: Denna kod i filen web.config anger att bindningar till GET och POST inte ska automatgenereras i WSDL dokumentet

Använd proxy generatorer: Genom att använda program som automatgenererar proxyklasser givet en WSDL fil minskas både risken för fel och utvecklingstiden. ASP.NET erbjuder *wsdl.exe* samt ”*Add Web Reference*” från *Visual Studio.NET (VS.NET)*.

Övrigt: Alla implementationer är inte överens om hur elementet *Header* och dess attribut *mustUnderstand* och *actor* ska behandlas. Många har dessutom problem med glesa arrayer. Dessa problem är på väg att försvinna, men ännu finns det anledning att tänka på dem vid designen av webbtjänster.

Figur 13 visar en intressant demonstration av fungerande SOAP interoperabilitet mellan olika plattformar. Exemplet är från [6] och visar en handdator, Intermec 700, som via

SOAP implementationen *pocketSOAP* kommunicerar med en Unix maskin som kör Apache SOAP. Denna kommunicerar i sin tur med en Windows XP maskin som kör ASP.NET. [6]



Figur 13: Demonstration av SOAP interoperabilitet

2.8 Slutsats

I mitt tycke är SOAP utan tvekan ett mycket intressant och kompetent protokoll för kommunikation mellan olika plattformar. Dess enkla uppbyggnad kring existerande teknologier gör att det är mycket flexibelt. Det kan användas i allt från RPC- till Message passing system. Eftersom SOAP bygger på XML är det dessutom väldigt utbyggbart. Med HTTP som kommunikationskanal undviks problemen med brandväggar som är vanliga hos existerande RPC system.

SOAP är inte kopplat till någon specifik plattform och det är relativt enkelt att implementera på nya plattformar, mycket tack vare att det är så enkelt uppbyggt och inte har så många inbyggda funktioner, som t.ex. distribuerad Garbage Collection (GC) och transaktionshantering. Detta är även en nackdel, eftersom utvecklaren själv måste implementera sådan extra funktionalitet. Med t.ex. CORBA och DCOM kommer mycket extra funktionalitet på köpet, men i stället är de mycket svårare att implementera på nya plattformar, dessutom lider de av problemen som nämndes under 2.1. Eftersom SOAP är textbaserat är det lätt att läsa och förstå, t.ex. vid debuggning. Prestandamässigt är det dock något långsammare och mer bandbreddsslukande än binära protokoll.

Alltfler SOAP implementationer kommer ständigt ut på marknaden och även existerande RPC system som CORBA börjar stödja SOAP. Möjligheterna till integration mellan olika system har helt klart ökat med SOAP och dess webbtjänst-applikationer. Visst finns det fortfarande interoperabilitetsproblem som måste lösas innan allt fungerar perfekt, men jag tror definitivt SOAP är här för att stanna.

3 Säkra webbtjänster

Behovet av säkerhet hos webbtjänster är minst lika stort som för andra webbaserade tekniker. SOAP specifikationen nämner dock ingenting om hur en webbtjänst kan säkras, men som tur är finns det andra tekniker. SOAP är dessutom utbyggbart och detta utnyttjas i *Web Services Security (WS-Security)*, se 3.6. Nedan visas några allmänna säkerhetskrav för meddelanden som skickas i ett distribuerat system:

Konfidentiell kommunikation: Meddelanden skall inte kunna läsas av en tredje part som avlyssnar kommunikationskanalen. Detta kan undvikas med hjälp av *Secure Socket Layer (SSL)*, se 3.4. Många meddelanden lagras temporärt på servern och en person som får tillgång till denna kan läsa dem om de inte är krypterade. Detta löses genom att kryptera känsliga delar av meddelandet i stället för hela kanalen som hos SSL, se 3.6.

Identifikation: För att kunna begränsa vilka som har rättighet att använda vissa tjänster krävs det att tjänsten kan verifiera att användaren är den som den påstår sig vara. Detta kan ske med hjälp av lösenord eller mer sofistikerade metoder som *digitala signaturer*, se 3.2 och *digitala certifikat*, se 3.3.

Integritet: Meddelandet får inte modifieras mellan sändare och mottagare. Detta kan kontrolleras med t.ex. digitala signaturer, se 3.2.

Duplicerade meddelanden: Om ett meddelande t.ex. innehåller en förfrågan om att ta ut pengar på ett konto vill inte sändaren att någon annan ska kunna fånga upp meddelandet och sända det igen (replay-attack). Detta kan undvikas genom att tidstämpla eller numrera meddelandet. Det är sedan mottagarens ansvar att ignorera meddelanden som kommer för sent eller de som den redan har behandlat.

Icke förnekande: I vissa fall är det bra att kunna bevisa att ett meddelande skapats av en viss person. Även detta kan lösas med digitala signaturer, se 3.2.

Alla dessa krav gäller även för webbtjänster, även om alla kanske inte krävs samtidigt av samma webbtjänst. WS-Security erbjuder tekniker som uppfyller samtliga av ovan nämnda krav, se 3.6.

En webbtjänst är ofta sammankopplad i en kedja med andra webbtjänster, där alla i kedjan är beroende av varandra. Alla kedjor har en svagaste länk och hackers är inte sena att utnyttja detta. Tyvärr är det bara möjligt att kontrollera säkerheten för sina egna webbtjänster, men om alla i kedjan gör det kan ett förtroende byggas upp.

Denial of Service (DOS)-attacker går ut på att en server som innehåller en tjänst bombarderas med förfrågningar i en sådan hastighet att den inte hinner med. Detta leder till att legitima användare inte kan använda tjänsten. Webbtjänster som drabbas av DOS-attacker propagerar förfrågningar vidare till andra webbtjänster i kedjan och detta leder till en dominoeffekt, där en attack kan slå ut ett flertal webbtjänster. DOS-attacker är generellt svåra att skydda sig emot. En strategi är att blockera IP-adresserna som attacken kommer ifrån och en annan är att sända upptagetsignaler till alla tills attacken är över. Dessa strategier gäller webbtjänster såväl som andra Internettjänster. [7][8]

3.1 Kryptering

Kryptering är en teknologi som gör att information kan förvrängas så att den blir oläsbar och sedan dekrypteras så att ursprungsinformationen kommer tillbaka. Det finns två typer av kryptering och båda använder nycklar till krypteringsalgoritmen:

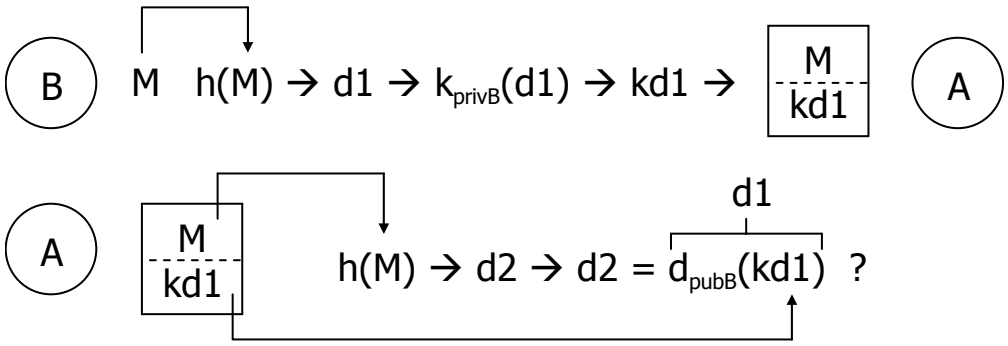
Symmetrisk kryptering: Samma nyckel används till både kryptering och dekryptering av meddelandet. Nyckeln hålls hemlig och måste därför spridas på ett säkert sätt via en krypterad kanal eller t.ex. via en diskett.

Asymmetrisk kryptering: Använder två olika nycklar, en *privat* och en *publik*. Ett meddelande som krypteras med den publika nyckeln kan endast dekrypteras med den privata nyckeln och vice versa. Den privata nyckeln hålls hemlig medan den publika är tillgänglig för alla.

Symmetrisk kryptering är snabbare än asymmetrisk kryptering och passar bättre till att kryptera stora informationsmängder, som elementet *Body* i ett SOAP meddelande. Asymmetrisk kryptering passar däremot bättre för identifikation via digitala signaturer och för att säkerställa integritet. [8][9]

3.2 Digitala signaturer

En digital signatur är kopplad till en viss informationsmängd och genom användningen av asymmetrisk kryptering kan användaren som signerade informationen identifieras och dessutom kan det kontrolleras att informationen inte har förändrats, dvs., att dess integritet kan säkerställas. En *digest* är ett värde som erhålls då en informationsmängd körs genom en säker hashfunktion. Kravet är att olika informationsmängder måste ge olika digest och givet en digest ska det vara omöjligt att få fram informationsmängden. Nedan visas ett exempel på digital signering då Bob skickar ett meddelande till Alice:



Bob (B) beräknar först en digest (d1) för meddelandet (M) och krypterar denna med sin privata nyckel (k_{privB}). Bob skickar sedan meddelandet tillsammans med den krypterade digest:en (kd1) till Alice (A). Hon dekrypterar sedan digest:en (kd1) med Bobs publika nyckel (d_{pubB}). Alice måste ha fått tag på Bobs publika nyckel på ett säkert sätt, t.ex. via ett *digitalt certifikat*, se 3.3. Hon använder sedan samma hashfunktion (h) som Bob på meddelandet. Om värdet hon får (d2) är samma som den dekrypterade digest:en (d1) vet hon att meddelandet är oförändrat och att det bara kan vara Bob som krypterade digest:en, eftersom hans publika nyckel kunde dekryptera den. Bobs krypterade digest (kd1) är det som kallas *digital signatur*.

Notera att digital signering inte döljer meddelandet. Det är vanligt att meddelandet signeras som ovan. Klartextmeddelandet krypteras sedan med symetrisk kryptering. Den symmetriska nyckeln, som ofta är en tillfälligt skapad *sessionsnyckel*, krypteras med mottagarens publika nyckel och allt bakas samman till ett meddelande, som mottagaren sedan kan dekryptera genom att först dekryptera sessionsnyckeln med sin privata nyckel. Sessionsnyckeln används sedan för att dekryptera klartextmeddelandet, som hashas och jämförs med den digitala signaturen som ovan. På detta sätt uppnås identifikation, icke förnekande, konfidentiell kommunikation och integritet. Detta är precis vad som sker i WS-Security exemplet under sektion 3.6.1. [8][9]

3.3 Digitala certifikat

Vid användning av digitala signaturer uppstår problemet att mottagaren måste få tag på sändarens publika nyckel på ett säkert sätt. I exemplet under 3.2 måste Alice vara säker på att hon har fått Bobs nyckel och att det inte är en bedragare som utger sig för att vara Bob. Samma problem uppstår för SSL då webbservrar måste veta att de talar med den server de tror sig tala med. Lösningen på detta är digitala certifikat.

Ett digitalt certifikat är ett sorts digitalt pass som innehåller en publik nyckel som är bunden till en viss person, företag eller enhet. Certifikatet är digitalt signerat av en *Certification Authority (CA)*, så att innehållet inte kan modifieras. CA:n är normalt en myndighet eller en tjänst hos ett tredjepartsföretag och dess publika nyckel erhålls på ett säkert sätt, ofta via installation av webbläsare. Då en CA signerar ett certifikat betyder det att den har kontrollerat ägarens identitet och att den kan bekräfta detta. Förtroendet för CA:n är avgörande för att digitala certifikat ska fungera.

Den mest använda standarden för digitala certifikat är *X.509*. Ett sådant certifikat innehåller information för att unikt identifiera ägaren, som namn, e-postadress, osv. Det innehåller även ägarens publika nyckel, CA:ns namn, en giltighetsperiod för certifikatet, ett serienummer, samt en digital signatur av certifikatets innehåll som skapats med CA:ns privata nyckel. Digitala certifikat kan publiceras på t.ex. webbsidor eller som en del i ett SOAP meddelande, se 3.6.1. [9][10]

3.4 Secure Socket Layer (SSL)

Secure Socket Layer (SSL) är det säkerhetsprotokoll som idag är vanligast för att skapa säkra kommunikationskanaler över Internet. SSL finns implementerat i alla stora webbservrar och webbläsare och är lätt att använda genom att skriva *https* i stället för *http*. SSL har antagits som Internetstandard under namnet *Transport Layer Security (TLS),* men det namn som vanligen används är SSL.

SSL använder digitala certifikat och digital signering för att skapa en säker, konfidentiell kommunikationskanal mellan två parter. Data som skickas via SSL kan inte modifieras utan att de båda parterna får reda på detta. SSL stödjer ett flertal krypteringsalgoritmer och identifikationssätt och dessa förhandlas fram mellan de båda parterna i början av SSL uppkopplingen.

SSL kan användas för att skapa säkra webbtjänster, men det har ett antal nackdelar som gör att alternativa tekniker ibland passar bättre. SSL är relativt långsamt och framförallt

förhandlingsprocessen tar tid. Kryptering av hela kanalen är heller inte effektivt om endast en del av SOAP meddelandet behöver skyddas. Det finns dock hårdvaru-acceleratorer som snabbar upp SSL.

För webbtjänster är *delkryptering* av meddelanden extra viktigt. Ett vanligt scenario är att en kund beställer en vara hos ett företag via en webbtjänst. I meddelandet inkluderas information om köpet till företaget, samt information till kundens bank om hur företaget ska betalas (kreditkortsnummer, osv.). Företaget kontaktar kundens bank via en annan webbtjänst och skickar vidare meddelandet. Företaget skall inte kunna se bankens information och vice versa. Det som efterfrågas är *end-to-end* säkerhet trots flera noder, samt *delkryptering*. SSL klarar inte detta, men lyckligtvis finns det andra tekniker som gör det, se 3.6. [10][11]

3.5 HTTP identifikation

Ett enkelt sätt att implementera säkra webbtjänster är att använda de identifikations-mekanismer som redan finns färdiga för HTTP, tillsammans med SSL. Microsofts webbserver *Internet Information Server (IIS)* och Windows 2000/XP server arbetar tillsammans för att erbjuda stöd för detta. Nedan beskrivs några identifikationstekniker för HTTP, samt deras för- och nackdelar:

Basic: Används för oskyddad identifikation av klienten. Användarnamn och lösenord skickas som base64-krypterad text, vilket är lätt att dekryptera. Stöds av de flesta webbläsare. IIS godkänner access till webbtjänsten om inloggningsinformationen matchar ett godkänt användarkonto.

Basic över SSL: Samma som Basic, men inloggningsinformationen är skyddad med hjälp av SSL.

Digest: Använder hashfunktioner för att överföra inloggningsinformationen på ett säkert sätt. Digest identifikation stöds dock inte av så många utvecklingsverktyg för webbtjänster. IIS godkänner access till webbtjänsten om inloggningsinformationen matchar ett godkänt användarkonto.

Integrerad Windows identifikation: Säker identifikationsmetod som använder Kerberos. Är dock endast användbar i intranät. Alla webbtjänstklienter stödjer inte denna metod.

Samtliga av ovan nämnda tekniker kräver att klienten har ett användarkonto på maskinen där webbtjänsten ligger. Detta fungerar oftast bra i intranät, men om webbtjänsten har ett stort antal användare eller om den ska vara tillgänglig över Internet är det inte alltid praktiskt användbart. I dessa fall är det vanligt att användarinformation lagras i en databas. Inloggningsinformationen kontrolleras då mot databasen innan access tillåts. För att detta ska vara möjligt måste denna information skickas med andra metoder än ovan.

Ett praktiskt ställe att skicka med extra information i ett SOAP meddelande är i elementet *Header*. Problemet är då att användarnamn och lösenord skickas i klartext. Detta kan dock lösas med SSL. Det har hittills inte funnits något standardiserat sätt att

skicka inloggningsinformation i elementet *Header*. I nästa sektion beskrivs dock en specifikation för detta, tillsammans med andra tekniker för säkra webbtjänster.

Det är även viktigt att tänka på att SOAP inte är bundet till HTTP. Det kan t.ex. även användas över SMTP. Genom att inte basera säkerheten på underliggande protokoll binder man inte applikationen till ett speciellt kommunikationsprotokoll och detta kan vara värdefullt i framtiden. [3][9][12]

3.6 Web Services Security (WS-Security)

Web Services Security (WS-Security) är en ny säkerhetsspecifikation för webbtjänster, som publicerades av IBM, Microsoft och VeriSign, den 5 april 2002. Det är ännu inte en standard, men chanserna är stora att det i framtiden kommer att bli det. WS-Security ingår som en del i ett större ramverk som kallas *Global XML Web Services Architecture (GXA)*. Där ingår även WS-Routing, WS-Coordination, WS-Inspection, WS-Referal samt WS-Transaction. WS-Security föreslår en standardiserad mängd SOAP extensions (tillägg) i elementet *Header*, som kan användas för att bygga säkra webbtjänster. WS-Security utgör även grunden för en mängd framtida säkerhetsspecifikationer, se 3.6.2.

WS-Security erbjuder en mekanism för att överföra säkerhetssymboler, t.ex. användarnamn och lösenord samt digitala certifikat. Det krävs ingen speciell typ av säkerhetssymbol, utan denna mekanism är tänkt att vara utbyggbar till att stödja en mängd olika format. Det finns även specificerat hur kryptering av binära säkerhetssymboler som Kerberos biljetter och X.509 certifikat skall gå till. Med hjälp av dessa och i vissa fall digital signering, kan identiteten för ett meddelande kontrolleras.

Digital signering erbjuds genom användningen av säkerhetssymboler och XML signering [13] och detta gör att WS-Security kan säkerställa integriteten för ett meddelande, dvs., att meddelandet inte har modifierats sedan det skickades. Det finns stöd för multipla signaturer av valda delar av meddelandet och signering kan ske av flera olika aktörer. Detta är mycket användbart i exemplet om delkryptering, som beskrevs under 3.4. Kunden kan i detta fall signera sin del av meddelandet medan företaget signerar sin del, innan meddelandet skickas vidare till banken. Företagets del adderas alltså till meddelandet och består t.ex. av företagets bankkontonummer dit kundens pengar skall gå. Banken kan sedan kontrollera identiteten för både kundens del och företagets del, samt de olika delarnas integritet.

Konfidentiell kommunikation garanteras genom användningen av säkerhetssymboler och XML kryptering [14]. Det finns stöd för kryptering av delar av SOAP meddelandet, vilket gör att olika aktörer kan kryptera sin del av meddelandet. Ett exempel på användningen av detta beskrevs under 3.4.

Det finns många typer av fel som kan uppstå då säkerhetsinformation behandlas av en mottagare. Säkerhetssymbolen kanske inte stöds eller så kan den inte identifieras. Signaturen kan vara felaktig, dekrypteringen kan misslyckas, osv. Samtliga fel måste returneras till sändaren via elementet *Fault*, se 2.5. Det finns ett antal specifika felkoder för WS-Security, se [17], som kan användas för att specificera felet.

Digitala signaturer är i sig inte tillräckligt för att bestämma identiteten på sändaren av ett meddelande. Någon kan spela av ett meddelande och återsända det vid ett senare tillfälle (replay-attack). WS-Security rekommenderar därför användningen av TTL-värden (Time-To-Live), nounce-värden eller sekvensnummer, för att garantera att ett SOAP meddelande är unikt. Ett exempel på detta visas under 3.6.3. [15][16][17]

3.6.1 WS-Security exempel

Figur 14 innehåller ett exempel på ett SOAP meddelande som illustrerar användningen av säkerhetssymboler (i detta fall ett X.509 certifikat), digital signering samt kryptering. Allt detta sker enligt WS-Security specifikationen [17]. Under figuren förklaras de olika delarna av meddelandet.

```
(001) <?xml version="1.0" encoding="utf-8"?>
(002) <S:Envelope xmlns:S="http://www.w3.org/2001/12/soap-envelope"
            xmlns:ds="http://www.w3.org/2000/09/xmldsig#"
            xmlns:wsse="http://schemas.xmlsoap.org/ws/2002/04/secext"
            xmlns:xenc="http://www.w3.org/2001/04/xmlenc#">
(003)   <S:Header>
(004)      <m:path xmlns:m="http://schemas.xmlsoap.org/rp/">
(005)         <m:action>http://fabrikam123.com/getQuote</m:action>
(006)         <m:to>http://fabrikam123.com/stocks</m:to>
(007)         <m:from>mailto:johnsmith@fabrikam123.com</m:from>
(008)         <m:id>uuid:84b9f5d0-33fb-4a81-b02b-5b760641c1d6</m:id>
(009)      </m:path>
(010)      <wsse:Security>
(011)         <wsse:BinarySecurityToken
                     ValueType="wsse:X509v3"
                     Id="X509Token"
                     EncodingType="wsse:Base64Binary">
(012)         MIIEZzCCA9CgAwIBAgIQEmtJZc0rqrKh5i...
(013)         </wsse:BinarySecurityToken>
(014)         <xenc:EncryptedKey>
(015)             <xenc:EncryptionMethod Algorithm=
                        "http://www.w3.org/2001/04/xmlenc#rsa-1_5"/>
(016)             <ds:KeyInfo>
(017)               <ds:KeyName>CN=Hiroshi Maruyama, C=JP</ds:KeyName>
(018)             </ds:KeyInfo>
(019)             <xenc:CipherData>
(020)                <xenc:CipherValue>d2FpbmdvbGRfE0lm4byV0...
(021)                </xenc:CipherValue>
(022)             </xenc:CipherData>
(023)             <xenc:ReferenceList>
(024)                 <xenc:DataReference URI="#enc1"/>
(025)             </xenc:ReferenceList>
(026)         </xenc:EncryptedKey>
(027)         <ds:Signature>
(028)            <ds:SignedInfo>
(029)               <ds:CanonicalizationMethod
                    Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
(030)               <ds:SignatureMethod
                  Algorithm="http://www.w3.org/2000/09/xmldsig#rsa-sha1"/>
(031)               <ds:Reference>
(032)                  <ds:Transforms>
(033)                     <ds:Transform
                            Algorithm="http://schemas.xmlsoap.org/
                              2001/10/security#RoutingSignatureTransform"/>
(034)                     <ds:Transform
                     Algorithm="http://www.w3.org/2001/10/xml-exc-c14n#"/>
(035)                  </ds:Transforms>
(036)                  <ds:DigestMethod
                    Algorithm="http://www.w3.org/2000/09/xmldsig#sha1"/>
(037)                  <ds:DigestValue>LyLsF094hPi4wPU...
```

```
(038)                    </ds:DigestValue>
(039)                 </ds:Reference>
(040)              </ds:SignedInfo>
(041)              <ds:SignatureValue>
(042)                       Hp1ZkmFZ/2kQLXDJbchm5gK...
(043)              </ds:SignatureValue>
(044)              <ds:KeyInfo>
(045)                  <wsse:SecurityTokenReference>
(046)                      <wsse:Reference URI="#X509Token"/>
(047)                  </wsse:SecurityTokenReference>
(048)              </ds:KeyInfo>
(049)           </ds:Signature>
(050)        </wsse:Security>
(051)     </S:Header>
(052)     <S:Body>
(053)        <xenc:EncryptedData
                    Type="http://www.w3.org/2001/04/xmlenc#Element"
                    Id="enc1">
(054)           <xenc:EncryptionMethod
                Algorithm="http://www.w3.org/2001/04/xmlenc#3des-cbc"/>
(055)           <xenc:CipherData>
(056)              <xenc:CipherValue>d2FpbmdvbGRfE0lm4byV0...
(057)              </xenc:CipherValue>
(058)           </xenc:CipherData>
(059)        </xenc:EncryptedData>
(060)     </S:Body>
(061)  </S:Envelope>
```

Figur 14: SOAP meddelande med X.509 certifikat, digital signering och kryptering

Rad (003)-(051) innehåller elementet *Header*.

Rad (004)-(009) specificerar meddelandets routinginformation enligt *WS-Routing* [18].

Rad (010)-(050) innehåller elementet *Security*. Här finns den säkerhetsrelaterade informationen för meddelandet.

Rad (011)-(013) specificerar en säkerhetssymbol för meddelandet. I detta fall är det en signerad binär säkerhetssymbol i form av ett X.509 certifikat, som är krypterad med base64. Rad (012) innehåller det krypterade certifikatet i förkortad version, för att göra exemplet mer läsbart.

Rad (014)-(026) specificerar nyckeln som används för att kryptera delar av meddelandet. Det är ofta av effektivitetsskäl en symmetrisk nyckel som tillfälligt har skapats för användning under en tidsbegränsad session och därför är den krypterad. Rad (015) visar vilken algoritm som användes för att kryptera nyckeln (asymmetrisk RSA-kryptering) och rad (016)-(018) specificerar namnet på mottagarens publika nyckel, som användes vid krypteringen. Mottagaren kan nu använda sin privata nyckel för att dekryptera den symmetriska nyckeln. Rad (019)-(022) visar den krypterade nyckeln. Rad (023)-(025) identifierar de block i meddelandet som har krypterats med den symmetriska nyckeln. I detta fall är det endast elementet *Body* som har krypterats, vilket URI:n ”#enc1” på rad (024) visar. Rad (053) innehåller det motsvarande id-numret.

Rad (027)-(049) specificerar den digitala signaturen, som i detta fall baseras på X.509 certifikatet från rad (011)-(013). Signaturen skapas med sändarens privata nyckel, som alltså hör samman med den publika nyckeln i certifikatet. Rad (029) indikerar vilken ”Canonicalization” algoritm som ska användas. En sådan algoritm gör att två logiskt

lika XML dokument ger samma *digest,* om de körs genom samma hashfunktion. Skillnader i indentering, ordning av attribut, osv., tas alltså bort av denna algoritm. Rad (030) specificerar algoritmen för signaturen. I detta fall är det asymmetrisk RSA-kryptering över hashfunktionen *Secure Hash Algorithm 1 (SHA1)*.

Rad (031)-(039) identifierar de delar av meddelandet som ska signeras. Rad (033) definierar en ”RoutingSignatureTransform”. Denna talar om att det inte bara är elementet *Body* som ska signeras, utan även routinginformationen på rad (004)-(009). Detta gör att även vägen för ett SOAP meddelande kan verifieras. Rad (034) specificerar den ”Canonicalization” algoritm som ska användas på de delar av meddelandet som valdes på rad (033), innan de ska köras genom hashfunktionen SHA1. Rad (037) innehåller en förkortad version av digest:en, som hashfunktionen producerar.

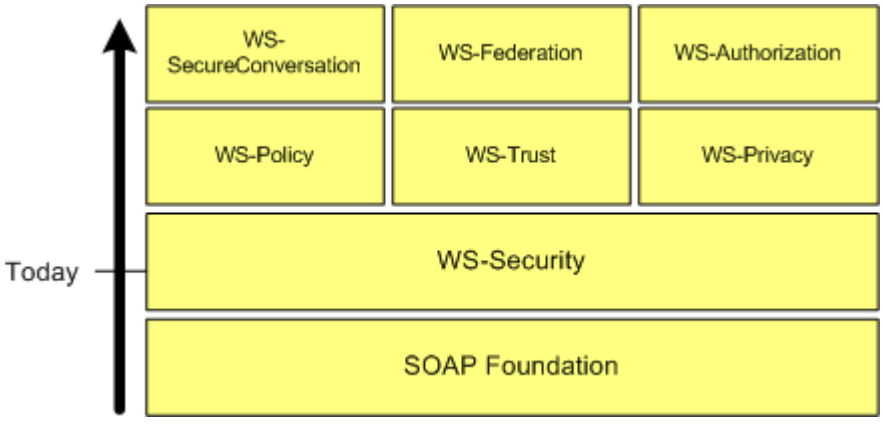
Om någon modifierar informationen om hashfunktionen eller algoritmerna, skulle detta förstöra signeringen. Detta undviks genom att allt inom elementet *SignedInfo*, rad (028)-(040), signeras enligt algoritmen på rad (030). Först skapas en digest med SHA1 och sedan krypteras denna med RSA för att skapa signaturvärdet som återfinns på rad (042).

Rad (044)-(048) visar vilken nyckel som användes för att skapa signaturen. I detta fall använde sändaren sin privata nyckel som hör samman med den publika nyckeln i X.509 certifikatet som finns på rad (011)-(013). URI:n ”#X509Token” pekar på detta certifikat, vars publika nyckel alltså kan dekryptera signaturvärdet på rad (042), vilket sedan kan användas för att verifiera skaparens identitet och meddelandets integritet.

SOAP meddelandets *Body* finns på raderna (052)-(060). Rad (053)-(059) representerar det krypterade datat i form av XML kryptering. Rad (053) indikerar att elementvärdet har ersatts. Här identifieras även det krypterade blocket som ”enc1”, vilket används av den symmetriska nyckeln på rad (024), för att peka ut vad som ska dekrypteras. Rad (054) specificerar krypteringsalgoritmen, i detta fall symmetrisk *Triple-DES (Data Encryption Standard)*. Rad (055)-(058) innehåller den krypterade texten. [8][9][17]

3.6.2 Framtida specifikationer

WS-Security utgör grunden för en mängd specifikationer som för närvarande är under utveckling. I Figur 15 visas en bild över detta. [15]

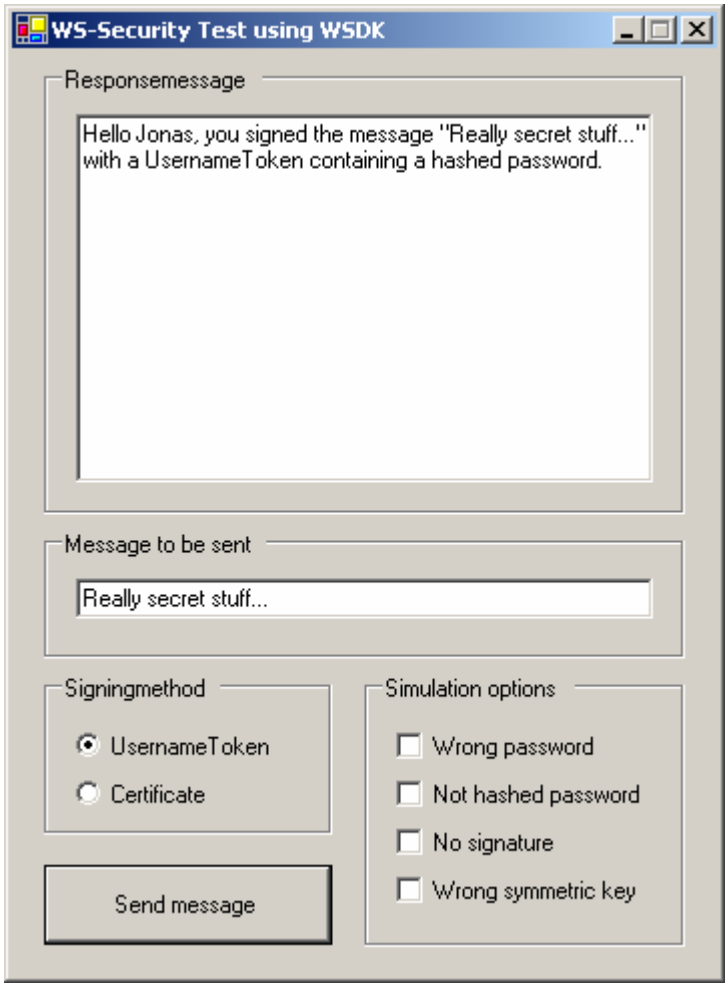


Figur 15: Framtida specifikationer som bygger på WS-Security

3.6.3 Implementation

För att bekräfta att WS-Security fungerar i praktiken och inte bara i teorin utvecklades en applikation. Till denna användes Microsofts nyligen släppta (2002-08-26) .NET bibliotek *Web Services Development Kit (WSDK),* som stödjer WS-Security. Det var en ”Technology Preview” av WSDK och därför var dokumentationen på vissa håll felaktig. Källkod och HTML dokumentation till applikationen finns under ”Dokument” på webbsidan för detta examensarbete, se [37]. Läs *Readme.txt* inuti projektkatalogen för installationsanvisningar.

I Figur 16 visas klientdelen av applikationen. I textrutan ”Message to be sent” skriver användaren in den meddelandetext som skall skickas till webbtjänsten som utgör serverdelen av applikationen. SOAP meddelandets *Body* signeras med antigen ett *X.509 certifikat* eller en *UsernameToken* (som består av det hårdkodade användarnamnet ”Jonas” och ett hash:at lösenord), beroende på vad användaren väljer. Elementet *Body* krypteras därefter för konfidentiell kommunikation. Detta sker via symmetrisk kryptering (Triple-DES) och en delad 128-bitars nyckel, som det antas att webbtjänsten känner till.



Figur 16: Klientdelen av applikationen som testar WS-Security med WSDK

Webbtjänsten har en enda metod: ”Hello”, vars uppgift är att verifiera att elementet *Body* signerades och att en godkänd identifikationsmetod användes (en UsernameToken med hash:at lösenord eller ett X.509 certifikat). Om inga fel uppstod returneras ett meddelande innehållande meddelandetexten, samt användarnamnet eller certifikat-namnet, annars returneras ett undantag som beskriver felet. Resultatet av anropet visas i textrutan ”Responsemessage” hos klienten.

WSDK sköter automatiskt både verifiering av TTL-värden och signaturen samt dekryptering av meddelandet, innan metoden ”Hello” anropas. WSDK kastar ett undantag om användaren inte kan identifieras, om TTL-värdet har gått ut, om signaturen är felaktig eller om dekrypteringen misslyckas. I dessa fall anropas aldrig ”Hello”. Denna automatiska kontroll är möjlig eftersom en klass som implementerar gränssnittet *IPasswordProvider,* samt en klass som implementerar gränssnittet *IDecryptionKeyProvider,* pekas ut i filen *web.config*. WSDK anropar dessa klasser under verifieringen.

Klienten erbjuder även användaren möjligheten att skicka olika typer av felaktiga meddelanden. Detta sker genom att kryssa i olika alternativ under ”Simulation options”. Det går t.ex. att skicka fel lösenord, meddelanden med fel lösenordstyp, meddelanden utan signatur, samt kryptera meddelanden med fel nyckel. Felmeddelandet som returneras kan sedan studeras i textrutan ”Responsemessage”.

I ett mer ”verkligt” scenario skulle lösenordet i klassen *PasswordProvider* hämtas på ett annat, säkrare sätt, t.ex. från en databas. Den symmetriska nyckeln skulle kunna vara en sessionsnyckel, som nyskapas och skickas med i varje anrop som i exemplet i Figur 14. Om en symmetrisk nyckel används under för lång tid, finns risken att någon knäcker den.

Replay-attacker undviks om servern kontrollerar det unika nounce-värdet som finns i en UsernameToken mot en tabell av gamla nounce-värden och endast accepterar meddelanden med nya värden. Created-värdet i en UsernameToken kan användas för att avgöra när ett nounce-värde är tillräckligt gammalt för att tas bort från tabellen. I applikationen används Created-värdet för att ignorera 30 sekunder gamla meddelanden och därigenom minska risken för replay-attacker. Detta är dock överflödigt eftersom TTL-värdet är satt till 30 sekunder av klienten, vilket gör att WSDK automatiskt kommer att returnera ett fel om meddelandet anländer för sent. Kontrollen av Created-värdet togs ändå med om klienten skulle glömma att skicka med ett TTL-värde eller om TTL-värdet är för stort. Om ett certifikat används måste TTL-värden användas, alternativt bör ett nounce-värde eller sekvensnummer inkluderas i elementet *Body*, som sedan kan kontrolleras av servern.

Applikationen visar att det är fullt möjligt att på ett relativt enkelt sätt implementera WS-Security i .NET, med hjälp av WSDK. Samtliga allmänna säkerhetskrav som nämndes i början av kapitlet kunde uppfyllas och signering kunde ske med både en UsernameToken och ett X.509 certifikat. [19]

3.7 Andra teknologier

SSL, HTTP identifikation och WS-Security är inte de enda tekniker som kan användas för att skapa säkra webbtjänster. Microsoft har som sagt släppt WSDK (2002-08-26) och erbjuder implementation av WS-Security i ASP.NET webbtjänster, men det är inte alla utvecklingsplattformar som har hunnit så långt. Nedan beskrivs kortfattat några alternativa teknologier [20]:

XML signering: Denna teknologi specificerar XML syntax och regler för att skapa och representera digitala signaturer. Det finns stöd för delsignering av ett XML dokument. WS-Security använder XML signering, men det kan även användas fristående.

XML kryptering: Definierar en process för att kryptera data och representera resultatet med XML. Det som krypteras kan t.ex. vara ett helt XML dokument, ett XML element eller innehållet i ett XML element. Liksom XML signering används det av WS-Security, men kan även fungera fristående.

XML Key Management Specification (X-KMS): Protokoll som används för att distribuera och registrera publika nycklar med hjälp av XML dokument. Kan användas tillsammans med XML signering och XML kryptering för att utbyta och verifiera de nycklar som krävs av dessa teknologier. X-KMS bygger på SOAP och WSDL.

Security Assertion Markup Language (SAML): XML baserad säkerhetsstandard för utbyte av identifikationsinformation.

3.8 Slutsats

Jag anser att detta kapitel har visat att det är fullt möjligt att skapa säkra webbtjänster på ett relativt enkelt sätt. När SSL och HTTP identifikation inte passar eller räcker till, finns det andra teknologier som tar vid. Med WS-Security kan samtliga allmänna säkerhetskrav som nämndes i början av kapitlet uppfyllas, vilket även bevisades i praktiken under 3.6.3. Stödet för delkryptering och delsignering av meddelanden är mycket användbart i många webbtjänstscenarier med fler än två parter. Något som är värt att notera är att WS-Security utökar SOAP meddelandets storlek väldigt mycket, se Figur 14. Detta gör att mer bandbredd upptas och det är alltid negativt för prestandan.

Det finns även andra alternativa tekniker som kan användas för att skapa säkra webbtjänster, se 3.7. Problemet är att det inte finns en standard som är accepterad av alla. WS-Security har dock god potential att bli den standard som krävs för att framtidens webbtjänster ska erbjuda både säkerhet och interoperabilitet. Det faktum att WS-Security bygger på och använder existerande teknologier som XML signering, XML kryptering, X.509 certifikat, SOAP och WSDL, samt att stora och betydelsefulla företag som Microsoft, IBM och VeriSign står bakom, ger tyngd åt teknologin. Det ingår dessutom i ett större ramverk (GXA) och nya framtida specifikationer bygger på WS-Security, se 3.6.2.

Om det blir WS-Security eller någon annan teknologi som blir allmänt accepterad säkerhetsstandard återstår att se, men jag tror att det är ett måste med en standardisering innan webbtjänster kan få sitt stora genombrott.

4 Kommunikation med Java

Detta kapitel kommer att utreda möjligheterna för kommunikation mellan .NET och Java. Integreringen kommer att ske med webbtjänster över SOAP, se kapitel 2. Webbtjänster är mycket lämpliga till denna typ av integrering, eftersom de är språk- och plattformsoberoende.

Ett antal olika tänkbara scenarion har implementerats för att testa kommunikationsmöjligheterna i praktiken. Under 4.1 testas först överföring av olika datatyper och under 4.2 implementeras sedan kedjor av webbtjänster, där olika webbtjänster anropar varandra. Den utvecklingsmiljö som har använts är *JBuilder 6 Enterprise* med *Web Services Kit for Java*. Detta val baserades på att DataVis sedan tidigare använder JBuilder, men andra utvecklingsmiljöer som *IBM WebSphere*, *SunONE Studio* och *Oracle9i JDeveloper* är också möjliga kandidater.

Webbtjänsterna som är skapade i JBuilder använder en SOAP server som baseras på *Apache Axis*. SOAP servern körs lokalt på en *Tomcat 4.0* webbserver, som startas upp tillsammans med den aktuella webbtjänsten. I längden är det dock bättre att lägga upp webbtjänsterna på en applikationsserver som *Borland Enterprise Server* eller *IBM WebSphere Application Server*. De tester som utförs i detta kapitel är dock oberoende av var SOAP servern körs.

Källkoden till samtliga applikationer finns att hämta under ”Dokument” på webbsidan för detta examensarbete, se [37]. För att öka chanserna till lyckad kommunikation har jag följt interoperabilitetstipsen under 2.7.

4.1 Test av datatyper

Det är självklart mycket viktigt att överföring av grundläggande datatyper som *int*, *float*, *string*, *boolean* och *arrayer* fungerar smärtfritt. Nedan följer implementationer som testar detta. Överföring av klasser är även möjligt, men det är relativt ovanligt i webbtjänstsammanhang och har därför inte prioriterats.

4.1.1 .NET klient → Java webbtjänst

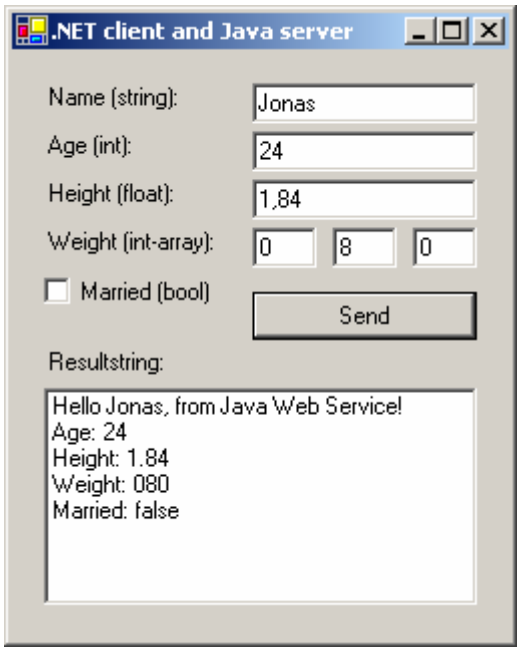
Denna applikation består av en .NET klient i form av ett Windows formulär som skickar ett antal olika parametrar till en Java webbtjänst, som i sin tur returnerar dem i en sträng. Figur 17 på nästa sida visar klientdelen av applikationen.

Resultatsträngen visar att parametrarna överfördes på ett korrekt sätt. Weight-fälten läggs i en int-array för att testa överföring av arrayer.

4.1.2 Java klient → .NET webbtjänst

Denna applikation visas i Figur 18 på nästa sida och gör exakt samma sak som den i Figur 17, förutom att klienten är skriven i Java och webbtjänsten i .NET.

Resultatsträngen visar även här att kommunikationen lyckades.



Figur 17: Parameteröverföring mellan .NET och Java

Figur 18: Parameteröverföring mellan Java och .NET

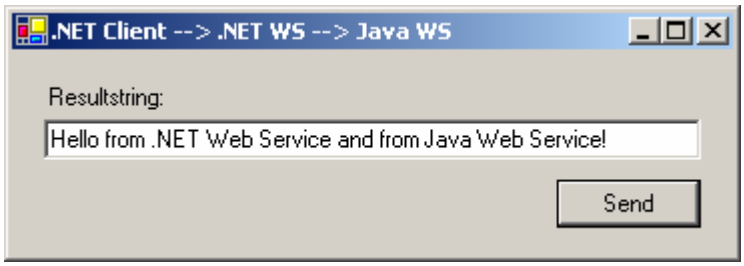
4.2 Kedjor av webbtjänster

Vid användningen av webbtjänster uppstår ofta situationen att olika webbtjänster anropar varandra i en kedja. Ett exempel på detta har tagits upp tidigare i kapitlet om säkra webbtjänster, se 3. Där diskuterades ett scenario där en klient (en kund) anropar en webbtjänst (ett företag) för att beställa en vara. Kreditkortsinformation skickas med i meddelandet och företagets webbtjänst skickar vidare denna tillsammans med egen information till en annan webbtjänst (en bank). I följande sektioner implementeras två olika scenarier med klienter och webbtjänster skrivna i .NET och Java.

4.2.1 .NET klient →.NET webbtjänst →Java webbtjänst

Ett möjligt scenario där en .NET klient anropar en .NET webbtjänst, som i sin tur anropar en Java webbtjänst är följande: Klienten anropar .NET webbtjänsten för att hämta information av något slag. Denna hämtar i sin tur information från ett antal olika källor. En av källorna är ett gammalt stordatorsystem som saknar stöd för .NET. Det finns dock en Java implementation på systemet och denna kan användas för att öppna upp och exponera systemets funktioner via en Java webbtjänst.

Figur 19 visar ett enkelt exempel på en applikation med kedjor av webbtjänster. .NET klienten anropar metoden *string Hello()* hos .NET webbtjänsten. I den metoden anropas metoden *string Hello(string mess)* hos Java webbtjänsten. Parametern *mess* innehåller texten ”Hello from .NET Web Service”. Java webbtjänsten returnerar denna, men lägger först till texten ” and from Java Web Service!” sist i strängen. .NET webbtjänsten returnerar sedan denna sträng till klienten, som visar den för användaren. Figur 19 visar att kommunikationen lyckades.

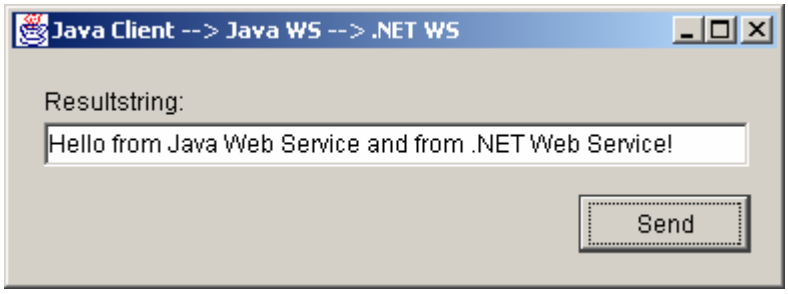


Figur 19: En .NET klient anropar en .NET webbtjänst som anropar en Java webbtjänst

4.2.2 Java klient →Java webbtjänst →.NET webbtjänst

Ett möjligt scenario där en Java klient anropar en Java webbtjänst, som i sin tur anropar en .NET webbtjänst är följande: En klient i form av en handdator saknar stöd för .NET, men har däremot stöd för Java. Denna klient skickar information till ett Unix system som exponerar en Java webbtjänst, som i sin tur kommunicerar med en .NET webbtjänst på en Windows maskin. Detta är exakt det scenario som visades under 2.7, Figur 13.

Figur 20 visar en applikation som fungerar på samma sätt som den i Figur 19, med skillnaden att klienten och webbtjänsterna är skriva i motsatt språk. Resultatsträngen visar att anropen lyckades.



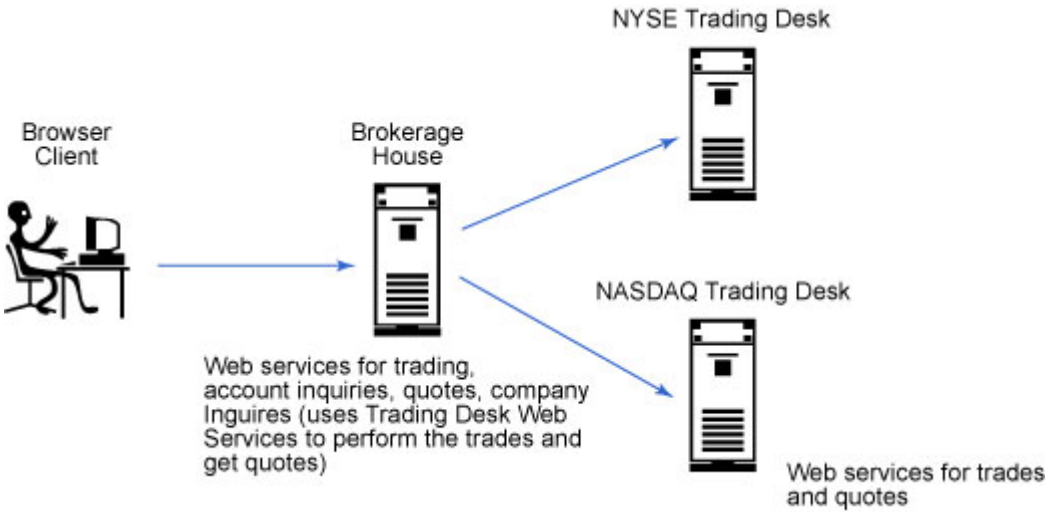
Figur 20: En Java klient anropar en Java webbtjänst som anropar en .NET webbtjänst

4.3 Säkerhet

I kapitel 3 nämndes ett antal olika tekniker för att skapa säkra webbtjänster, däribland HTTP identifikation, SSL samt WS-Security. Generellt för Java applikationer skapade i någon av de utvecklingsmiljöer som nämndes i början av kapitlet, är WS-Security idag inte är ett alternativ. IBM som är med i utvecklingen av WS-Security erbjuder dock en implementation i sin *IBM WebSphere Studio Application Developer,* se Figur 21.

HTTP identifikation är som det påpekades under 3.5 ett alternativ som använder webbservern, HTTP:s inbyggda mekanismer, SSL samt befintliga användarkonton, för att erbjuda säkerhet. Ett annat alternativ är att skicka användarnamn och lösenord i elementet *Header* och kryptera kanalen med hjälp av SSL. Användaruppgifterna kan sedan kontrolleras mot t.ex. en databas eller ett användarkonto. Problemet med dessa tekniker är som sagt att de inte ger möjlighet till delkryptering och delsignering av SOAP meddelandet. HTTP identifikation kräver dessutom att klienten har ett användarkonto på servern och det är inte praktiskt användbart i alla lägen.

Figur 21 visar en intressant demonstration på interoperabilitet mellan webbtjänster som använder WS-Security på plattformarna .NET och IBM WebSphere. Den visades upp på konferensen *XML Web Services One* i Boston den 26-30 augusti 2002. De olika lagren kan implementeras på antingen .NET eller IBM WebSphere. Klienten skickar meddelanden till ”Brokerage House” och identifierar sig via en *UsernameToken*, se 3.6.3. Kanalen krypteras med SSL. ”Brokerage House” kommunicerar med olika ”Trading Desk” och identifierar sig med ett X.509 certifikat. SOAP meddelandet signeras och krypteras med WS-Security tekniker innan det skickas iväg. Det är ett vanligt scenario idag att klienten inte har certifikat, utan istället använder användarnamn och lösenord. Webbservrar och webbtjänster kan däremot med fördel identifiera sig med certifikat. [21]



Figur 21: Demonstration av WS-Security mellan .NET och IBM WebSphere

4.4 Slutsats

De implementationer som har gjorts i detta kapitel är väldigt förenklade, men de visar ändå exempel på fungerande kommunikation mellan .NET och Java. Det skulle vara möjligt att testa betydligt fler datatyper som klasser, arrayer av klasser, osv., men det har jag av tidsskäl prioriterat bort.

När det gäller säker kommunikation mellan .NET och Java, är det inom den närmsta framtiden SSL och HTTP identifikation eller identifiering via information i elementet *Header*, som gäller. Detta om inte kommunikationen sker mellan .NET och IBM WebSphere, då även WS-Security kan användas. Möjligen kommer fler utvecklings-miljöer för Java att stödja WS-Security i framtiden. Det skulle i så fall innebära större möjligheter för säker kommunikation mellan plattformarna .NET och Java.

5 Kommunikation med COM

Det finns idag en enorm mängd implementerade COM komponenter och det är viktigt att .NET kan kommunicera med dessa. I vissa fall kan det även vara önskvärt att anropa .NET komponenter från en COM komponent. .NET och COM bygger dock på olika teknologier och därför krävs en brygga för att integrera dem.

Detta kapitel kommer att utreda möjligheterna för integration mellan .NET och COM, dessutom diskuteras *COM+* roll i .NET. Utöver detta kommer eventuella begränsningar och egenheter hos COM komponenter skapade i *Visual Basic 6.0 (VB6)* att belysas. I C++ finns möjligheten att initiera COM komponenter på en annan server (via DCOM), med funktionen *CoCreateInstanceEx*. En utredning skall göras för att kontrollera om det går att utföra en liknande operation i .NET med hjälp av *C#*.

5.1 .NET → COM

Det finns två olika tekniker för att anropa en COM komponent från .NET. Den första är att använda dialogrutan ”Add Reference” i VS.NET. Detta skapar automatiskt en *proxyklass* eller *Runtime Callable Wrapper (RCW)* för den aktuella COM komponenten. Denna kan sedan användas för att instantiera och anropa COM komponenten som vilken annan klass som helst. Inuti RCW klassen skickas alla anrop vidare till COM komponenten. Allt integreringsjobb sköts alltså automatiskt av proxyklassen.

Det är viktigt att tänka på att COM komponenter inte frigörs förrän .NET:s *Garbage Collector (GC)* frigör RCW klassen. COM komponenter bör dock frigöras så fort de har använts färdigt. För att åstadkomma detta kan följande anrop användas:
System.Runtime.InteropServices.Marshal.ReleaseComObject(RCW_obj_ref)

För att bättre kunna styra skapandet av RCW klassen kan programmet *TlbImp.exe* användas från kommandoraden. Här kan bl.a. RCW klassen specificeras som en *Primary Interop Assembly (PIA)*. Detta är en speciell typ av RCW som kan delas mellan flera olika .NET applikationer som anropar samma COM komponent. Det är även möjligt att anropa *Active X* komponenter från .NET. Proceduren för detta är samma som ovan, men Active X komponenten måste först köras genom programmet *AxImp.exe* från kommandoraden, för att skapa de dll-filer som ska refereras.

Tekniken som beskrivs ovan för att integrera .NET och COM, bygger på *tidig bindning*, dvs., adresserna till COM komponentens metoder är kända vid kompilering. Den andra tekniken som kan användas i .NET är *sen bindning* till COM komponenter och här bestäms adresserna under exekvering. Vid sen bindning behövs ingen RCW, utan proceduren går istället till enligt följande:

1. Ett objekt av klassen *Type* som motsvarar COM komponentens typ, skapas med hjälp av klassmetoden *Type.GetTypeFromProgID( )*.
2. *Type* objektet används tillsammans med *Activator.CreateInstance( )* för att skapa COM komponenten.
3. *Type* objektets metod *InvokeMember,* används för att anropa COM komponentens metoder.

Denna teknik är som synes något krångligare att använda än tidig bindning. Den är dessutom långsammare och lättare att göra fel på. Det finns dock tillfällen då sen bindning kan vara användbart. Det ger t.ex. möjlighet till mer sofistikerade programmeringstekniker som *polymorfism* och dessutom visade det sig att problemet som nämndes i början av kapitlet: att i C# kod initiera COM komponenter på en annan server, kan lösas med sen bindning och klassmetoden *Type.GetTypeFromProgID*, som nämndes ovan.

I de allra flesta fall uppstår inga problem med COM komponenter som har skapats i VB6. .NET sköter automatiskt konverteringen mellan i stort sett alla inbyggda datatyper på de olika plattformarna. De datatyper som kan översättas allra effektivast är VB:s *Byte, Integer* och *Long* samt endimensionella arrayer och structar, som enbart består av dessa datatyper. Detta beror på att de representeras på samma sätt i minnet på de olika plattformarna. Alla andra datatyper kräver mer arbete eftersom de tvingar fram konvertering från ett format till ett annat.

Den datatyp i VB6 som kan ställa till mest problem är *Variant*. Denna datatyp kan innehålla vilken annan datatyp som helst, förutom strängar av fix längd. .NET kan automatiskt konvertera *Variant* om den innehåller enkla datatyper som *Integer*, *Date*, *Boolean*, osv. Om den däremot innehåller komplexa, användardefinierade datatyper, kan inte .NET lista ut vilken datatyp den ska konverteras till. Lösningen är då att manuellt göra ändringar i *MSIL-koden* (Microsoft Intermediate Language) för RCW klassen, så att konverteringen utförs korrekt. Detta är ingen trivial uppgift och därför är det rekommenderat att om möjligt inte returnera *Variant* från COM komponenter som ska användas av .NET.

Det finns även andra specialfall då MSIL-koden för RCW klassen måste ändras för hand av utvecklaren. Några av dessa uppstår vid användningen av In/Out C-style arrayer, flerdimensionella C-style arrayer, null-värden för referenser till värdetyper, mm. För ytterligare specialfall, samt exempel på hur de kan lösas, se [24]. [22][23]

5.2 COM → .NET

I vissa lägen kan det vara önskvärt att från en COM komponent anropa en .NET komponent, även om det motsatta är betydligt vanligare. .NET erbjuder programmet *RegAsm.exe* för att skriva in en .NET komponents typinformation i systemregistret, så att olika COM tjänster kan komma åt den. COM komponenter kan nu skapa objekt av .NET komponenten med sen bindning. Som det nämndes ovan har sen bindning ett antal nackdelar och därför finns även möjligheten till tidig bindning. För att detta ska vara möjligt måste en typbiblioteksfil (*.tlb) skapas för .NET komponenten, med hjälp av programmet *TlbExp.exe*. Utvecklingsmiljöer som VB6 kan nu referera till tlb-filen, för att skapa tidig bindning till .NET komponenten. [22]

5.3 COM+

COM+ tjänster fanns redan på Windows NT som ett *add-on* vid namn *Microsoft Transaction Server (MTS)*. Numera finns det dock inbyggt i Windows 2000/XP. .NET har på inget sätt ersatt COM+, utan utnyttjar dess tjänster via speciella attribut och klasser, som återfinns under namnrymden *System.EnterpriseServices*. Programmet *RegSvcs.exe* kan användas för att registrera och konfigurera en .NET komponent för COM+ tjänster. COM+ gör livet lättare för utvecklaren genom att erbjuda följande tjänster:

- Transaktionshantering
- Object pooling
- Just-In-Time (JIT) object activation
- Rollbaserad säkerhet
- Event support
- Component message queuing
- Component load balancing

Samtliga av ovan nämnda tjänster kan användas i .NET. Prestandan kan dock påverkas negativt vid användning av COM+ tjänster, eftersom de bygger på en annan teknologi än .NET. Skillnader mellan plattformarna kräver konverteringar av data och det är kostsamt. Fördelarna med COM+ tjänsterna bör därför vägas mot eventuella prestanda-krav, innan de tas i bruk i en applikation. [22]

5.4 Slutsats

Möjligheterna till kommunikation mellan .NET och COM är mycket goda. VS.NET kan automatiskt skapa proxyklasser som kapslar in COM komponenter och anropar dem. Det finns dessutom ett antal kommandoradsprogram som kan hjälpa till vid integrering med Active X komponenter och vid kommunikation mellan COM komponenter och .NET komponenter. Det finns stöd för både sen och tidig bindning och COM+ tjänster är fullt användbara från .NET miljön, trots att de bygger på olika teknologier.

COM komponenter som är utvecklade i VB6 har inga direkta begränsningar och .NET klarar att översätta de flesta VB6 datatyper utan problem. De vanligaste problemen uppstår då datatypen *Variant* returneras. Denna bör därför i möjligaste mån undvikas. De sällsynta interoperabilitetsproblem som ändå kan uppstå mellan .NET och COM, kan i många fall lösas genom att manuellt gå in i MSIL-koden för RCW klassen och göra ändringar. Detta kan dock vara en ganska komplicerad process.

Slutligen kan sägas att det är fullt möjligt att initiera COM komponenter på en annan server med hjälp av sen bindning och metoden *GetTypeFromProgID* i klassen *Type*.

6 Transaktioner

En transaktion är en mängd sammanhängande operationer som lyckas eller misslyckas som en grupp. Om en av operationerna inte kan utföras skall alltså ingen annan operation heller utföras. Operationer som redan har utförts kan då behöva rullas tillbaka till ursprungsläget. Ett typiskt exempel på när transaktioner behövs är när en kund beställer en vara från en webbsida med sitt kreditkort. Det är då inte bara viktigt att det dras pengar från kundens konto, utan även att en order placeras i orderdatabasen. Om ordern inte kan placeras i databasen måste betalningen rullas tillbaka och beställningen avbrytas, annars kommer kunden betala för en vara som den aldrig får.

Detta kapitel ska utreda vilka möjligheter det finns för transaktionshantering i .NET. Eftersom transaktionssystem är komplicerade är det viktigt att det finns bra inbyggt stöd för dem i utvecklingsmiljön. Under 5.3 nämndes COM+ som en möjlig metod, men det finns även andra alternativ som skall undersökas och jämföras, se 6.1.

Det talas ibland om *lokala* eller *distribuerade* transaktioner. Lokala transaktioner består av operationer som utförs mot en enda lokal datakälla, t.ex. en *SQL Server*. Distribuerade transaktioner kan ha två olika betydelser. Den ena är när en transaktion består av operationer som utförs mot flera, ofta heterogena, datakällor, t.ex. en *SQL Server*, en *MSMQ (Microsoft Message Queue)* och en *Oracle databas*. Den andra betydelsen är när transaktioner består av operationer som utförs via objekt på olika servrar. I denna rapport kommer den första typen av distribuerade transaktioner att kallas *heterogena* transaktioner för att undvika missförstånd. Den andra typen av distribuerade transaktioner finns det för närvarande inte stöd för i .NET. Det finns dock planer på att stödja det i kommande versioner av .NET Remoting, se 7. Den som redan idag vill utnyttja distribuerade transaktioner hänvisas till att använda DCOM.

Transaktioner mellan webbtjänster är för närvarande inte möjligt. Webbtjänster är till naturen dåligt lämpade för transaktioner, eftersom de är tillståndslösa och har relativt långa anropstider, vilket leder till långa låsningar av datakällorna. Under 6.2 diskuteras dock *WS-Transaction,* en specifikation som i framtiden ska tillåta transaktioner mellan webbtjänster.

Redan idag finns det stöd i .NET för transaktioner *inom* enskilda webbtjänster. En webbtjänst kan t.ex. med attributet *TransactionOption* tala om att den vill utnyttja COM+ transaktioner för de uppgifter som den ska utföra. Transaktionen är lokal eller heterogen och ska inte förväxlas med distribuerade transaktioner *mellan* webbtjänster, som erbjuds av WS-Transaction.

Under 6.3 beskrivs implementationen av en webbtjänst som använder attributet *TransactionOption* för att hantera transaktioner med en SQL Server. Under samma sektion implementeras även en klass som använder *ADO.NET:s (ActiveX Data Objects.NET*) inbyggda stöd för transaktioner för att kommunicera med samma SQL Server. Utöver detta beskrivs implementationen av en databastransaktion som använder *Transact-SQL (T-SQL)*. [25]

6.1 Metoder

Det finns ett antal olika metoder för transaktionshantering i .NET. Följande sektioner kommer att ta upp deras respektive fördelar, nackdelar och användningsområden. Tillsist jämförs metodernas prestanda.

6.1.1 Databastransaktioner

Databastransaktioner använder lagrade procedurer och språket *Transact SQL (T-SQL)*, för att direkt i *SQL (Structured Query Language)* koden hantera transaktioner. Transaktioner kan på detta sätt utföras med en enda *round-trip* (vägen från klienten till servern och tillbaka) till databasservern. Det finns även stöd för nästlade transaktioner, vilket innebär att det går att starta en ny transaktion inuti en aktiv transaktion.

Nackdelen med denna metod är dock att utvecklaren måste implementera transaktionen i T-SQL, vilket är krångligare och mer begränsande än att koda i ett .NET kompatibelt programmeringsspråk. En annan nackdel är att det inte finns stöd för heterogena transaktioner.

Under 6.3.1 beskrivs implementationen av en klass som använder T-SQL och en lagrad procedur, för att hantera en transaktion med en SQL Server. Källkoden för den lagrade proceduren finns i Bilaga B, Figur 1. [25]

6.1.2 Manuella transaktioner

Vid manuella transaktioner kan utvecklaren själv kontrollera transaktionsgränserna med speciella kommandon för att starta och avsluta transaktionen. Även denna metod stödjer nästlade transaktioner. Det finns dock inget inbyggt stöd för heterogena transaktioner och även om det går att kontrollera sådana manuellt, är det ingen lätt uppgift.

.NET stödjer manuella transaktioner via olika klasser i ADO.NET. Dessa klasser är enkla att jobba med, även om de ofta kräver något mer jobb av utvecklaren än automatiska transaktioner, som diskuteras i nästa sektion. Transaktioner i ADO.NET är mindre effektiva än databastransaktioner, eftersom de kräver en round-trip för varje anrop till datakällan som utförs inuti transaktionen, plus round-trips för att starta och avsluta transaktionen.

Under 6.3.2 beskrivs implementationen av en klass som använder ADO.NET för att hantera en transaktion med en SQL Server. Källkoden för klassen finns i Bilaga B, Figur 2.

.NET stödjer även manuella *MSMQ (Microsoft Message Queue)* transaktioner genom klassen *MessageQueueTransaction*. [25]

6.1.3 Automatiska transaktioner

.NET använder COM+ tjänster för att stödja automatiska transaktioner. COM+ använder i sin tur Microsofts *Distributed Transaction Coordinator (DTC),* som transaktionshanterare. DTC:n implementerar ett *two-phase-commit* protokoll som garanterar att transaktionen lämnar alla deltagande datakällor i ett konsistent tillstånd. Alla typer av datakällor som implementerar ett DTC-kompatibelt gränssnitt kan delta i automatiska transaktioner. För närvarande finns implementationer för Microsoft SQL Server, MSMQ, Oracle, Sybase, m.fl. Med hjälp av COM+ och DTC:n kan .NET alltså erbjuda stöd för heterogena transaktioner.

Automatiska transaktioner implementeras på ett enkelt sätt i .NET via speciella attribut. ASP.NET webbsidor, se Figur 22, webbtjänster, se Figur 23 och .NET klasser, se Figur 24, kan alla använda dessa attribut. .NET klasser måste dessutom ärva från klassen *ServicedComponent*. COM+ kommunicerar sedan med DTC:n och skapar automatiskt en heterogen transaktion innehållande alla anrop till DTC-kompatibla datakällor som kan hittas i koden.

```
<@ Page Transaction="Required">
```

Figur 22: En ASP.NET webbsida som har markerats för automatiska transaktioner

```
public class Class1 : WebService
{
    [WebMethod(TransactionOption = TransactionOption.Required)]
    public void Method1()
    {
    }
}
```

Figur 23: En webbtjänstmetod som har markerats för automatiska transaktioner

```
[Transaction(TransactionOption.Required)]
public class Class1 : ServicedComponent
{
}
```

Figur 24: En .NET klass som har markerats för automatiska transaktioner

Nackdelen med automatiska transaktioner i .NET är den prestandasänkning som uppkommer vid kommunikationen med COM+ och DTC:n. .NET och COM+ bygger, som tidigare nämndes under 5.3, på olika teknologier. Det finns dessutom inget stöd för nästlade transaktioner.

Om en .NET applikation behöver en transaktion som innefattar heterogena datakällor finns det dock inget annat val än att använda automatiska transaktioner.

Under 6.3.3 beskrivs implementationen av en webbtjänst som använder en automatisk transaktion vid kommunikationen med en SQL Server. Källkoden för klassen finns i Bilaga B, Figur 3. [25]

6.1.4 Prestandajämförelse

Vi vet redan att databastransaktioner tar en round-trip medan manuella transaktioner tar en round-trip för varje anrop till datakällan som utförs inuti transaktionen, plus round-trips för att starta och avsluta transaktionen. Vi vet dessutom att automatiska transaktioner tappar prestanda vid kommunikationen med COM+ och DTC:n. Vi vet däremot inte hur mycket detta påverkar prestandan i praktiken. Under [26] görs ett prestandatest där de olika metoderna testas med 1-120 samtidiga användare och med transaktioner innehållande 11 samt 101 insättningar till en SQL Server.

Som väntat visar det sig att databastransaktioner är något snabbare än manuella transaktioner vid 11 insättningar, men skillnaden är marginell. Automatiska transaktioner är dock ca. 30% långsammare än de andra två och det beror på att tiden för att sätta upp transaktionen via COM+ och DTC:n är stor jämfört med tiden för transaktionen (11 insättningar). Det visar sig även att det är ganska stor skillnad om COM+ körs som en *library-activated application* i samma process som klienten, jämfört med att den körs som en *server-activated application* i en utomstående process. COM+ i en utomstående process är långsammare, men har dock fördelen att om den kraschar drar den inte med sig klienten i fallet.

Vid 101 insättningar är prestandaskillnaden mellan de olika transaktionsmetoderna betydligt mindre, även om databastransaktioner fortfarande är något snabbare än manuella transaktioner, som i sin tur är marginellt snabbare än automatiska transaktioner. Anledningen till att automatiska transaktioner i detta fall är i stort sett lika effektiva som de andra, är att tiden för att sätta upp transaktionen via COM+ och DTC:n är liten jämfört med tiden för transaktionen (101 insättningar). [25][26]

6.2 Transaktioner mellan webbtjänster

Webbtjänster är till naturen dåligt lämpade för transaktioner. De är tillståndslösa och har relativt långa anropstider, vilket leder till långa låsningar av datakällorna. Det finns ändå situationer där det kan vara önskvärt att sätta upp distribuerade transaktioner mellan webbtjänster. Detta gäller kanske framförallt i intranät, där anropstiderna är kortare och förtroendet mellan de deltagande webbtjänsterna är större än för ett Internetscenario, där mindre betrodda webbtjänster kan låsa andras datakällor under lång tid, medan de själva utför en operation.

Under 6.3.3 visas implementationen av en webbtjänst som använder attributet *TransactionOption* för att sätta upp en automatisk transaktion. Webbtjänster i .NET kan dessutom även utnyttja de andra två transaktionsmetoderna: databastransaktioner och manuella transaktioner. Dessa tre typer av transaktioner fungerar dock inte *mellan* webbtjänster, utan endast *inom* enskilda webbtjänster.

För närvarande finns inget stöd för distribuerade transaktioner mellan webbtjänster, men *WS-Transaction* [27] är en specifikation som i framtiden kan göra detta möjligt. Den har utvecklats av Microsoft, IBM och BEA och ingår som en del i ramverket *GXA*, där även *WS-Security*, se 3.6, ingår. Till skillnad från WS-Security finns det dock ännu ingen implementation av WS-Transaction på marknaden.

Tillsammans med *WS-Coordination*, som också ingår i GXA, specificerar WS-Transaction upp olika tillägg i elementet *Header*, på samma sätt som WS-Security gör. Dessa tillägg innehåller den information som behövs för att sätta upp och genomföra transaktioner, så att underliggande datakällor lämnas i ett konsistent tillstånd.

WS-Transaction erbjuder två olika modeller för transaktioner. Den första *är Atomic Transactions (AT)* och den använder liksom DTC:n ett *two-phase-commit* protokoll, där alla deltagande webbtjänster låser respektive datakälla tills transaktionen är klar. Detta kan som sagt vara godtagbart i ett intranät, men det kan ställa till med problem över Internet.

För transaktioner som tar längre tid och där förtroendet mellan deltagarna är lägre erbjuder WS-Transaction modellen *Business Activity (BA)*. Denna passar ofta bättre för Internetscenarion än AT, eftersom den inte tvingar deltagarna att låsa datakällan. Vid en avbruten transaktion skrivs ursprungsdata tillbaka till datakällan. En annan transaktion kan tyvärr under tiden ha läst in felaktigt data, eftersom datakällan inte är låst. BA ger alltså inte samma starka skydd mot inkonsistent data som AT. [27][28]

6.3 Implementation

För att testa de olika transaktionsmetoderna implementerades tre applikationer som kommunicerar med en SQL Server databas. Databasen simulerar ett banksystem med en tabell för kunder och en annan för konton. Samtliga applikationer utför en transaktion mot tabellen med konton. I transaktionen tas en summa pengar ut från det första kontot och sätts sedan in på det andra kontot. Om någon av dessa operationer misslyckas, rullas transaktionen tillbaka och avbryts. Kontona och den summa pengar som ska överföras, anges av användaren.

Källkoden till alla applikationer samt databasen finns att hämta under ”Dokument” på webbsidan för detta examensarbete, se [37].

6.3.1 Databastransaktion med T-SQL

Denna applikation körs från kommandoraden och den låter först användaren mata in konton och överföringssumma. Efter detta anropas den lagrade procedur som finns i Bilaga B, Figur 1. Denna procedur sköter hela transaktionen. Om alla operationer kan utföras görs *commit* för transaktionen, dvs., transaktionen avslutas på ett lyckat sätt. Om någon operation misslyckas rullas däremot transaktionen automatiskt tillbaka. Detta anges med kommandot: *SET XACT_ABORT ON*.

För dessa enkla operationer är det inte svårt att använda T-SQL för att utföra transaktionen. Det kan däremot vara svårt att avgöra vilken operation som misslyckades vid en avbruten transaktion. Det är även krångligt att i den lagrade proceduren avbryta transaktionen om ett angivet konto inte finns i databasen. Just nu finns ingen kontroll för detta. Dessa typer av kontroller är som vi ska se betydligt enklare att utföra med de andra transaktionsmetoderna.

6.3.2 Manuell transaktion med ADO.NET

Även denna applikation körs från kommandoraden, se källkoden i Bilaga B, Figur 2. Användaren får först mata in konton och överföringssumma. Efter detta öppnas kopplingen till databasen och transaktionen startas. De två operationerna utförs efter varandra och om de lyckas görs *commit* för transaktionen. Alla fel som uppstår genererar undantag som fångas av *catch*-blocket. Inuti detta rullas transaktionen tillbaka och felet presenteras för användaren. Om ett konto inte finns i databasen genereras ett undantag som talar om detta.

6.3.3 Automatisk transaktion med COM+

Denna applikation utvecklades som en webbtjänst för att testa användningen av attributet *TransactionOption* och dess automatiska COM+ transaktioner. Observera att det i detta fall bara finns en datakälla i transaktionen. För att automatiska transaktioner skall komma till sin fulla rätt bör det finnas flera heterogena datakällor. Källkoden för klassen finns i Bilaga B, Figur 3. Webbtjänsten testas via *asmx*-sidans automat-genererade webbgränssnitt, där användaren kan mata in konton och överföringssumma. I övrigt är koden väldigt lika den för manuella transaktioner, förutom att transaktionen startar automatiskt och att kommandona för *commit* och *rollback* ser annorlunda ut.

Tillsist kan sägas att det är fullt möjligt att baka in flera operationer i en lagrad procedur och anropa denna inuti en manuell eller automatisk transaktion. På detta sätt minskas antalet round-trips till databasen. Nackdelen är dock att kontrollen över enskilda operationer blir sämre.

6.4 Slutsats

Jag tycker att detta kapitel har visat att det finns ett bra stöd för transaktionshantering i .NET. Visserligen är det negativt att det ännu inte finns stöd för distribuerade transaktioner, men det kommer troligtvis i framtida versioner av .NET Remoting.

När det gäller de olika transaktionsmetoderna räcker det i de allra flesta fall med manuella ADO.NET transaktioner. Vid lokala transaktioner mot en enda datakälla, som i implementationerna ovan, är detta den metod som är enklast och bäst att använda i förhållande till prestanda. Om prestandan är av yttersta vikt är däremot databas-transaktioner med T-SQL det bästa alternativet, även om de kräver mycket av utvecklaren. Vid kommunikation med heterogena datakällor finns det dock bara ett alternativ och det är automatiska transaktioner med COM+ och DTC:n. Dessa är kostsamma att sätta upp, men om transaktionen är tidskrävande blir denna kostnad försumbar och metoden kan då prestandamässigt mäta sig väl med de andra två. I övrigt kan sägas att om prestandan är viktig bör COM+ om möjligt köras som en *library-activated application*.

Transaktioner mellan webbtjänster kan bli en realitet i framtiden med WS-Transaction och även om jag anser att transaktioner mellan webbtjänster i möjligaste mån bör undvikas, kan det uppstå många situationer då det är mycket användbart.

7 .NET Remoting

.NET Remoting är en teknologi som tillåter kommunikation mellan objekt i ett distribuerat system. Det objekt som anropas körs normalt på en fjärrbelägen server och det är .NET Remoting:s uppgift att skicka iväg eventuella parametrar, anropa objektet och skicka tillbaka returvärdet. Detta kan lika väl utföras av en webbtjänst, så varför finns då .NET Remoting och när bör det användas istället för webbtjänster? Under 7.2 diskuteras skillnaderna utförligt, men tills vidare kan det konstateras att .NET Remoting erbjuder mer komplex funktionalitet än webbtjänster, t.ex. stöd för referensparametrar, händelser (events), aktivering och livstidsunderhåll av objekt, mm.

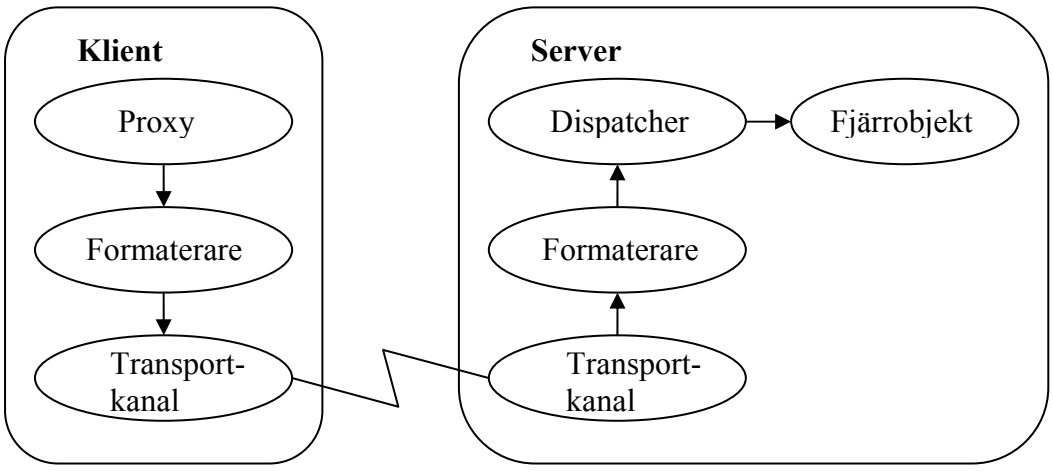
.NET Remoting är ersättaren till DCOM och som ersättare bör den ha ett antal fördelar gentemot sin föregångare. DCOM är t.ex. inte tillräckligt flexibelt och effektivt för att användas över Internet. .NET Remoting kan däremot köras över både *TCP (Transmission Control Protocol)* och HTTP och kan därmed användas i såväl intranät- som Internetlösningar. Att .NET Remoting är flexiblare än DCOM visas även av att formateringen av meddelanden är utbytbar. För närvarande finns binär formatering och SOAP, men det är möjligt att ange en egen formaterare. Detta innebär att webbtjänster kan erbjudas av .NET Remoting, dock med en del begränsningar, se 7.2.

Andra liknande teknologier är t.ex. CORBA och *Java RMI (Java Remote Method Invocation)*. .NET Remoting är flexiblare än dessa och fungerar ofta bättre i Internetlösningar. Inlärningströskeln är lägre än för CORBA, som dock har fördelen att det är plattformsoberoende. .NET Remoting kräver att både klient och server använder .NET, om inte SOAP över HTTP används vid kommunikationen.

Under 7.1 beskrivs arkitekturen för .NET Remoting och under 7.3 implementeras några applikationer som testar .NET Remoting i praktiken. [29][30]

7.1 Arkitektur

.NET Remoting bygger på en mycket flexibel och utbyggbar arkitektur. I Figur 25 visas en något förenklad bild över denna.



Figur 25: Arkitektur för .NET Remoting

Då klienten skapar en instans av ett fjärrobjekt skapas i själva verket ett lokalt proxyobjekt som gör att fjärrobjektet kan användas som om det vore ett lokalt objekt. Proxyn skickar vidare metodanrop till en formaterare som konverterar det till ett meddelande av ett visst format. För närvarande finns formaterare för binärt format samt SOAP, men det är möjligt att skriva egna formaterare. Nästa lager i kedjan är transportkanalen som även den är utbytbar. Transportkanalen skickar meddelandet till servern via TCP, HTTP eller ett alternativt eget protokoll. Hos servern passerar meddelandet lagren i motsatt ordning tills det når komponenten *Dispatcher*. Denna anropar rätt objekt på servern och skickar sedan tillbaka resultatet till klienten genom alla lagren.

Till skillnad från andra arkitekturer som DCOM och CORBA kan de flesta lagren antingen byggas ut eller ersättas helt. Dessutom kan nya lager sättas in i kedjan för att behandla meddelandet på något sätt. Det skulle t.ex. vara möjligt att lägga in ett lager mellan formateraren och transportkanalen, vars uppgift är att kryptera meddelandet.

Det är lätt att byta ut olika lager utan att göra ändringar i källkoden. Via speciella konfigurationsfiler kan bland annat formaterare och transportkanal anges. Det är även möjligt att ange flera transportkanaler på servern. Detta är intressant ur integreringssynpunkt, eftersom ett .NET Remoting objekt kan använda TCP och binär formatering för klienter på intranätet, samtidigt som det erbjuder ett webbtjänstgränssnitt via HTTP och SOAP för Internetklienter, som inte nödvändigtvis behöver använda .NET. Denna konfiguration testas under 7.3.2, där den ena klienten är implementerad i Java.

Det finns tre olika typer av fjärrobjekt i .NET Remoting. Den första typen är *SingleCall* och denna skapas på nytt vid varje nytt metodanrop och slängs därefter. Den sparar inte tillstånd mellan olika anrop och kan därför med fördel användas i ett lastbalanserat system. Den andra objekttypen är *Singleton* och den existerar som en enda instans på servern. Den delas mellan alla klienter och sparar tillstånd mellan olika anrop. Dessa båda typer kallas tillsammans *Server Activated Objects (SAO)*. Den tredje typen av fjärrobjekt är *Client Activated Objects (CAO)*. Dessa är unika för en viss klient och sparar tillstånd mellan olika anrop. CAO är av naturliga skäl den typ av objekt som skalar sämst och bör därför användas med förstånd.

SAO skapas med *new*-operatorn eller klassmetoden *Activator.GetObject()*. För att skapa CAO används *new* eller *Activator.CreateInstance()*. Det är viktigt att frigöra fjärrobjekt som inte längre används för att spara på serverns resurser. DCOM löser detta genom att klienten ”pingar” servern med jämna mellanrum. Så fort servern inte får några meddelanden frigörs objektet. Denna lösning fungerar i intranät, men tar även där upp onödigt mycket bandbredd med alla ping-meddelanden. .NET Remoting har en mycket mer skalbar lösning i form av *Leasing Distributed Garbage Collector (LDGC)*. Här bestäms en lånetid (leasetime) för CAO och Singletonobjekt av servern. Vid varje nytt metodanrop från klienten ökas lånetiden. När lånetiden går ut frigörs objektet från servern. Servern kan även skicka en varning till klienten innan objektet frigörs.

.NET Remoting kan skicka objekt *by-value* (en kopia) eller *by-reference* (en referens). Ett objekt som ska skickas som kopia måste markeras med attributet *Serializable* eller implementera gränssnittet *ISerializable*. Referensobjekt måste ärva från klassen *MarshalByRefObject*.

Det finns ett antal olika applikationer och komponenter som kan vara värd för .NET Remoting objekt, t.ex. Windows Services, kommandoradsprogram, Windows Forms samt webbservern *IIS (Internet Information Server)*. Det är dessutom möjligt att använda COM+ som värd, om ett .NET Remoting objekt vill utnyttja dess tjänster.

Det finns inget inbyggt stöd för säkerhet i .NET Remoting. Genom att använda IIS som värd och HTTP som transportkanal kan dock .NET Remoting använda IIS inbyggda säkerhet, som HTTP identifikation, se 3.5, och SSL, se 3.4. Det går även att utnyttja .NET:s rollbaserade säkerhetsprinciper för att endast ge access till vissa grupper av användare. Om någon annan värd än IIS används måste dock utvecklaren själv implementera all säkerhet. Med IIS som värd ökar dessutom skalbarheten och .NET Remoting objekt aktiveras automatiskt när ett anrop anländer. Nackdelen med IIS är dock att prestandan blir något sämre om inte antalet samtidiga användare är tillräckligt stort, se 7.2 för en prestandajämförelse av olika konfigurationer.

Till sist kan sägas att .NET Remoting stödjer asynkrona anrop till fjärrobjekt. Detta kan vara användbart om klienten inte har tid att vänta på returvärdet, utan vill utföra andra operationer först. Det finns även stöd för händelser (events), där servern anropar en metod hos klienten då en viss händelse uppstår. [22][29][30][32]

7.2 .NET Remoting kontra webbtjänster

.NET Remoting används precis som webbtjänster till att anropa fjärrobjekt. Denna sektion skall försöka reda ut vilka skillnader som finns mellan de båda teknologierna. De webbtjänster som åsyftas nedan är ASP.NET webbtjänster.

Till att börja med bör det nämnas att .NET Remoting i sig faktiskt kan användas för att erbjuda webbtjänster. Detta är möjligt om HTTP används som transportkanal och SOAP som formaterare. Denna typ av webbtjänster kan dock inte använda *Document Encoding*, se 2.6.2, utan är bundna till *RPC Encoding*, se 2.6.1. Ett annat problem är att de WSDL-filer som genereras av .NET Remoting ofta innehåller .NET specifika konstruktioner, vilket gör att klienter som inte kör .NET får problem att använda objektet, se 7.3.2. Detta problem kan ofta lösas genom att endast använda enkla eller egendefinierade datatyper och undvika .NET specifika datatyper som t.ex. klasserna *DataSet* och *Hashtable*.

.NET Remoting är lyckligtvis väldigt flexibelt och både transportkanal och formaterare kan bytas ut efter behag, se Figur 25. SOAP specifikationen kräver ingen specifik transportkanal, men ASP.NET webbtjänster måste trots det använda HTTP, eftersom det är den enda transportkanal som stöds av ASP.NET. En annan viktig skillnad är att en mängd olika applikationer och komponenter kan vara värd för .NET Remoting objekt. ASP.NET webbtjänster är däremot knutna till IIS.

Både ASP.NET webbtjänster och .NET Remoting använder de inbyggda säkerhets-mekanismer som erbjuds av IIS, för att skapa en säker kommunikation. Utan IIS som värd finns ingen säkerhet i .NET Remoting, om utvecklaren inte själv implementerar ett lager som hanterar säkerheten. Webbtjänster kan dock även säkras med specifikationer som *WS-Security*, se 3.6.

Webbtjänster är i grunden tillståndslösa och ett nytt objekt skapas för varje nytt anrop. För att behålla tillstånd mellan anrop kan dock komponenterna *Session* och *Application* användas. .NET Remoting har som det visades under 7.1, tre olika typer av objekt: *SingleCall* (tillståndslösa), *Singleton*, samt *CAO*. Detta erbjuder utvecklaren en större valfrihet än för webbtjänster. Om IIS används som värd stöds dock inte CAO.

.NET Remoting erbjuder, förutom de som nämnts ovan, ett antal tjänster som inte stöds av webbtjänster. Några av dessa är: referensparametrar, livstidsunderhåll av objekt via *Leasing Distributed Garbage Collector (LDGC)*, samt stöd för händelser.

Under [31] görs ett prestandatest där ASP.NET webbtjänster samt olika .NET Remoting konfigurationer testas med 1-100 samtidiga användare. Metoden som anropas returnerar 1, 20 eller 50 poster från en SQL Server. Posterna returneras i ett objekt. Det visar sig att .NET Remoting objekt med TCP som transportkanal och binär formatering är klart snabbast. Detta beror dels på att TCP är effektivare än HTTP och dels på att binär formatering är effektivare än SOAP. Skillnaden minskar dock något med antalet poster som returneras. ASP.NET webbtjänster visar sig vara snabbare än .NET Remoting objekt som använder SOAP som formaterare. Detta beror på att ASP.NET webbtjänster hanterar SOAP på ett mer effektivt sätt. .NET Remoting objekt som använder IIS som värd är något långsammare än de som använder en annan värd. Detta är logiskt eftersom IIS utgör ett extra lager som måste passeras. När antalet användare ökar har de dock en fördel, eftersom de då kan utnyttja skalbarheten hos IIS. Till sist kan sägas att ASP.NET webbtjänster och .NET Remoting objekt med HTTP och binär formatering är prestandamässigt jämförbara, även om den senare är något snabbare.

SOAP formateraren i .NET Remoting har som sagt vissa begränsningar både när det gäller interoperabilitet och prestanda. Det visar sig även att binär formatering prestanda-mässigt är att föredra, oavsett om transportkanalen är TCP eller HTTP. Detta gör att .NET Remoting i de allra flesta fall kräver att både klient och server kör .NET, eftersom binär formatering inte är plattformsoberoende. [30][31]

7.3 Implementation

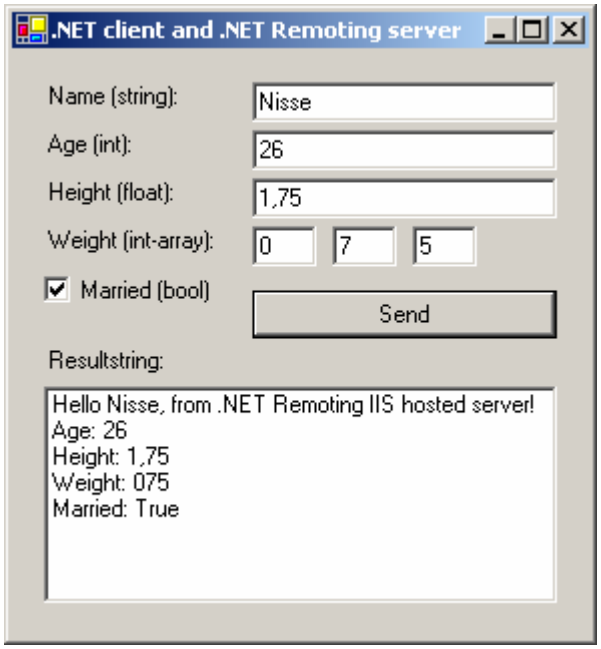
För att testa .NET Remoting i praktiken implementerades två applikationer. I den första använder både klient och server .NET, medan den andra visar prov på integrering mellan .NET Remoting och Java. Klienten är där implementerad i *JBuilder 6 Enterprise* med *Web Services Kit for Java*. Serverobjektet (fjärrobjektet) är konfigurerat för *Single-Call* i båda exemplen, eftersom det inte behöver spara tillstånd mellan olika anrop.

Källkoden till alla applikationer finns att hämta under ”Dokument” på webbsidan för detta examensarbete, se [37].

7.3.1 .NET klient → .NET Remoting server

Det grafiska gränssnittet för denna applikation ser exakt ut som det i Figur 17, där en .NET klient kommunicerar med en Java webbtjänst. Även metoden är densamma och den tar alltså en *string*, en *int*, en *float*, en *int-array* samt en *bool* som parametrar. Dessa slås samman till en sträng och returneras. Kommunikationen sker via en TCP kanal med binär formatering. Kanalen sätts upp hos klienten med en XML-konfigurationsfil. Det är

därför lätt att ändra till t.ex. HTTP med SOAP formatering, även efter kompilering. Serverobjektet startas med ett kommandoradsprogram, som också använder en konfigurationsfil för att sätta upp en TCP, samt en HTTP kanal. Några bort-kommenterade kodrader har dock lämnats för att visa hur servern kan sättas upp utan konfigurationsfil, men detta är ofta sämre, eftersom det då inte går att ändra konfiguration utan att kompilera om applikationen. Bortkommenterade kodrader har lämnats även hos klienten och genom att använda dessa kan klienten kommunicera via HTTP och binär formatering med ett annat serverobjekt, som har IIS som värd. Meddelandet som returneras avslöjar vilken värd som används, se Figur 26. IIS konfigureras som värd genom att sätta katalogen (i detta fall *RemotingServerIIS*) med serverobjektet som en *virtuell katalog*, samt via inställningar i filen *web.config*.



Figur 26: .NET klient som pratar med ett .NET Remoting objekt med IIS som värd

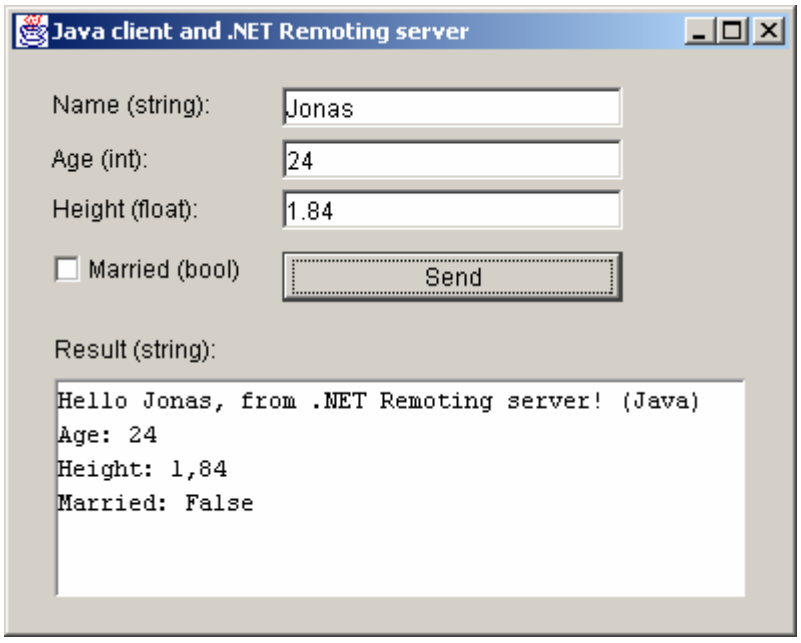
För att klienten ska veta hur serverobjektet och dess metoder ser ut, behöver den känna till någon typ av gränssnitt (interface) för objektet. Detta problem uppstår hos alla RPC system. CORBA löser det med sina *idl*-filer som beskriver serverobjektets gränssnitt. DCOM använder typbiblioteksfiler (*tlb*-filer) till samma sak. Problemet med båda dessa filer är att de använder ett eget språk som utvecklaren måste lära sig och detta kan vara ganska krångligt. .NET Remoting löser dock problemet på ett bättre sätt. En teknik är att implementera serverobjektet utifrån ett gränssnitt (som är en del av programspråket) och sedan använda detta gränssnitt hos klienten. Problemet är då att klienten inte kan utnyttja konfigurationsfiler och det är en stor nackdel. Detta problem löses om kommandoradsprogrammet *SOAPSuds* används på serverobjektet för att kompilera ner metadata som beskriver objektets gränssnitt i en *dll*-fil. Denna fil kan sedan användas hos klienten, som nu vet hur serverobjektet ser ut. I applikationen har båda dessa tekniker använts i studiesyfte, men normalt väljer man en av dem. Konfigurationsfilen hos klienten använder metadata från *SOAPSuds*, medan de bortkommenterade kodraderna som används för kommunikation med ett serverobjekt på IIS:en, använder gränssnittstekniken.

7.3.2 Java klient → .NET Remoting server

Ur integreringssynpunkt är det mycket intressant att .NET Remoting erbjuder HTTP kanalen med SOAP formatering. Detta innebär i praktiken att ett .NET Remoting objekt kan kommas åt från alla klienter som stödjer HTTP och SOAP, dvs. alla typer av webbtjänstklienter. Klienten kan t.ex. vara implementerad i Java. En begränsning är dock att endast *RPC Encoding*, se 2.6.1, stöds.

Ett möjligt scenario är att ett .NET Remoting objekt utvecklas för användning i ett intranät, där alla klienter kör .NET. Av effektivitetsskäl används TCP med binär formatering. Senare uppstår ett behov av att komma åt objektet från en Java klient. Detta är oftast inget problem, eftersom det är enkelt att i konfigurationsfilen lägga till en kanal för HTTP och SOAP, som Java klienten kan använda.

En Java klient utvecklades i JBuilder för att ansluta till serverobjektet under 7.3.1, men precis som det konstaterades under 7.2 kan det bli problem med .NET specifika konstruktioner i WSDL-filen som beskriver metoden. I detta fall kunde inte parametern som skickar en *int-array* accepteras av JBuilder. Därför implementerades ett nytt serverobjekt med en liknande funktion, men utan int-arrayen. Denna gång gick det bra att ansluta från Java klienten, se Figur 27. Resultatsträngen visar att kommunikationen lyckades och att det nya serverobjektet anropades (”Java” anges inom parantes för att visa detta).



Figur 27: Java klient som pratar med ett .NET Remoting objekt

.NET Remoting objektet på servern erbjuder även en TCP kanal med binär formatering. För att återskapa scenariot ovan utvecklade jag en .NET klient liknande den i Figur 27, som anropar serverobjektet via denna kanal. Detta visar att .NET och Java klienter kan användas för att kommunicera med samma .NET Remoting objekt, i detta fall via olika kommunikationskanaler.

7.4 Slutsats

Jag anser att .NET Remoting är en mycket flexibel, utbyggbar och användbar arkitektur. Även om det kräver mycket av utvecklaren är det väldigt kraftfullt att möjligheten finns att utveckla egna transportkanaler, formaterare och mellanlager för modifiering av meddelanden.

Det är även mycket intressant ur integreringssynpunkt att inte bara .NET klienter kan anropa .NET Remoting objekt. Implementationen under 7.3.2 visade att det är fullt möjligt att använda Java klienter, även om det finns klara begränsningar för vilka datatyper som kan skickas. För att utnyttja .NET Remoting till fullo krävs dock att både klient och server använder .NET.

Jag tror att den största användningen för .NET Remoting är i intranät, precis som fallet är för dess föregångare DCOM. I intranät är kommunikationen ofta skyddad bakom en brandvägg och då är det bästa och snabbaste alternativet TCP med binär formatering. Det är dock bra att möjligheten finns att använda .NET Remoting över Internet och genom att använda IIS som värd kan säkerheten och skalbarheten garanteras. Det är visserligen något mindre effektivt att använda IIS, men om säkerheten är viktigast är detta det klart enklaste alternativet, eftersom säkerhet redan finns inbyggd i IIS.

När det kommer till att välja mellan .NET Remoting och ASP.NET webbtjänster finns det ett antal riktlinjer att följa. För det första är webbtjänster det självklara valet om interoperabilitet med andra plattformar är av högsta prioritet. Om både klient och server kör .NET och det är viktigt att kunna skicka alla .NET datatyper eller om specifika funktioner som CAO, händelser, referensparametrar eller livstidshantering av objekt är önskvärda, är .NET Remoting den arkitektur som ska väljas. Om prestandan är viktigare än säkerhet och interoperabilitet skall .NET Remoting med TCP och binär formatering väljas. Även om det går att erbjuda webbtjänster via HTTP kanalen och SOAP, anser jag att denna möjlighet endast bör utnyttjas i specialfall som vid implementationen under 7.3.2. Om syftet är att skapa en ren webbtjänst som ska kunna användas av alla är det bäst att använda ASP.NET webbtjänster. Dessa webbtjänster är enklare att implementera, de stödjer både *Document-* och *RPC Encoding* och de är dessutom prestandamässigt effektivare.

Jämfört med ASP.NET webbtjänster är .NET Remoting helt enkelt en mer komplex arkitektur som erbjuder fler alternativ, men samtidigt kräver mer av utvecklaren. Detta bekräftades även under implementationen. Personligen anser jag dock att .NET Remoting är enklare att bemästra än t.ex. CORBA. *SOAPSuds* och gränssnitt i .NET är exempelvis enklare att hantera än CORBA:s idl-filer. Det går givetvis att testa mycket mer av .NET Remoting än det som har implementerats i exemplen. Det skulle t.ex. vara intressant att prova CAO, klasser som värde- och referensparametrar, lånetider för objekt (leasetime) och händelser, men det har av tidsskäl prioriterats bort.

Till sist kan sägas att .NET Remoting är en värdig ersättare till DCOM. Vid lanseringen av .NET har det dock kommit lite i skymundan av ASP.NET webbtjänster, men i vissa situationer är som sagt .NET Remoting att föredra.

8 UDDI och löst kopplade system

Universal Description Discovery and Integration (UDDI) kallas ofta webbtjänsternas ”gula sidor”. Detta är en ganska bra jämförelse eftersom UDDI tillåter utvecklaren att göra sökningar efter en viss webbtjänst baserat på nyckelord, kategorier, gränssnitt, mm. Sökresultatet kan t.ex. bestå av WSDL-filer och sökvägar till webbtjänsten. Dessa kan utvecklaren sedan använda i designfasen för att skapa en proxyklass som kommunicerar med webbtjänsten.

UDDI är med andra ord ett register för webbtjänster. Faktum är att UDDI i sig är en webbtjänst och därför är det plattformsoberoende. Det finns idag ett antal stora *publika* UDDI noder, där företag och privatpersoner kan registrera sina webbtjänster. Microsoft och IBM är två av de företag som erbjuder dessa noder för allmänheten. De webbtjänster som endast ska användas i intranät bör dock inte registreras i publika UDDI noder, utan de registreras med fördel i *privata* UDDI noder. Dessa noder är specifika för en viss organisation och kan inte kommas åt utifrån. För närvarande finns få implementationer av privata UDDI noder. Microsofts nya operativsystem *Windows .NET Server* innehåller dock en inbyggd privat UDDI nod som kan användas.

UDDI kan inte bara användas i designfasen av webbtjänster, utan även under körning (runtime). Detta kapitel ska främst undersöka hur UDDI kan användas under körning för att skapa löst kopplade system, dvs. system där sökvägarna till webbtjänster inte är hårdkodade. Det ska vara möjligt att flytta webbtjänster till nya servrar och klienterna ska automatiskt upptäcka detta under körning och använda UDDI för att slå upp de nya adresserna. Under 8.1 implementeras en applikation som testar detta scenario. [3][33]

8.1 Implementation

För att testa användningen av UDDI för löst kopplade system utvecklades webbtjänsten ”Minus”, som används av en .NET klient. Webbtjänsten subtraherar sina två parametrar och returnerar resultatet i en sträng. ”Minus” registrerades på Microsofts UDDI nod: *https://test.uddi.microsoft.com/inquire*, som används vid testning. Registreringen utfördes med en applikation som utvecklades speciellt för detta ändamål. För att kunna registrera och underhålla webbtjänster hos Microsofts UDDI noder krävs ett *Passport* login.

Eftersom att både webbtjänsten och klienten i detta fall finns på samma intranät, vore det bäst att använda en privat UDDI nod, men tyvärr fanns ingen sådan tillgänglig. Applikationen skulle dock se ut och fungera på samma sätt även med en privat UDDI nod. Källkoden till applikationen finns att hämta under ”Dokument” på webbsidan för detta examensarbete, se [37]. Klienten är uppbyggd enligt följande mall:

1. I designfasen görs en sökning i UDDI registret baserad på webbtjänstens namn, som är ”Minus” i detta fall. Webbtjänsten hittas sedan och dess WSDL-fil används av klienten för att skapa en proxyklass.
2. Sökvägen till UDDI noden och en unik nyckel som pekar ut webbtjänsten i UDDI registret sparas i en konfigurationsfil hos klienten. När klienten startas hämtas aktuell sökväg till webbtjänsten från UDDI och sparas i en variabel. Senast

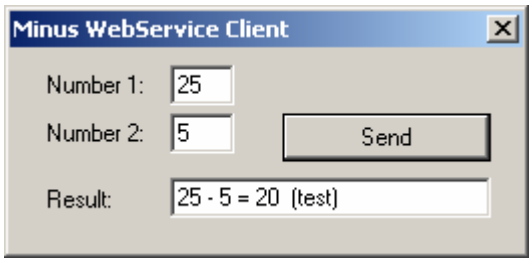
fungerande sökväg kan även med fördel hämtas från disk (om den finns lagrad där), för att undvika onödig kommunikation med UDDI registret i detta läge.

3. Klienten anropar webbtjänsten och använder sökvägen som erhölls ovan.
4. Om anropet misslyckas kan det bero på att webbtjänsten har förflyttats och att sökvägen därmed inte längre är aktuell. Klienten använder nu den unika nyckeln och anropar UDDI för att hämta den aktuella versionen av sökvägen.
5. Den nya sökvägen jämförs med den gamla. Om de är olika anropas webbtjänsten på nytt med den nya sökvägen. Om detta anrop lyckas ersätts den gamla sökvägen med den nya och sparas eventuellt till disk för senare användning (vid punkt 2). Om sökvägarna däremot är lika har webbtjänsten inte förflyttats. Klienten kastar då ett undantag. Ett fel genereras också om webbtjänsten inte kan anropas trots att sökvägarna är olika.

Observera att denna mall endast fungerar om ägaren av webbtjänsten är noga med att uppdatera UDDI registret med den nya sökvägen, så fort webbtjänsten har förflyttats.

Ett möjligt scenario är att en testversion av en webbtjänst läggs upp på en server under utvecklingsfasen. Olika klienter ansluter till denna och använder UDDI enligt mallen ovan. Releaseversionen av webbtjänsten använder samma gränssnitt, men den ska ligga på en annan server. Det enda som nu behöver göras för att få klienterna att börja använda releaseversionen, är att ta bort testversionen och uppdatera sökvägen i UDDI registret. Klienterna kommer automatiskt hämta hem och använda den nya sökvägen.

Applikationen som har utvecklats bygger på precis det scenario som beskrivs ovan. Figur 28 visar en klient som anropar testversionen av webbtjänsten ”Minus” (”test” anges inom parantes i resultatsträngen för att visa detta). Figur 29 visar samma klient, men innan anropet gjordes uppdaterades sökvägen i UDDI registret och testversionen togs ur produktion. Klienten upptäckte detta automatiskt under körning och gick därför över till att använda releaseversionen, som Figur 29 visar.



Figur 28: Testversionen av webbtjänsten ”Minus” anropas

Figur 29: Releaseversionen av webbtjänsten ”Minus” anropas

Mallen som beskrivs ovan erbjuder löst kopplade system. Tekniken kallas *Retry on Failure* och är bara ett av flera möjliga användningsområden för UDDI under körning (runtime). En annan teknik är *Optimized Access Point Discovery*, där flera webbtjänster med samma gränssnitt är utspridda på olika servrar. En klient vill av prestandaskäl ansluta till den webbtjänst som geografiskt är närmast belägen. UDDI sökningar under körning kan användas även för detta ändamål. [33][34]

8.2 Slutsats

Implementationen visar att UDDI kan användas för att skapa löst kopplade system, där klienten är både robustare (Retry on Failure) och mer dynamisk (automatisk uppdatering av sökvägen under körning) än då sökvägen till webbtjänsten är hårdkodad.

Webbtjänster är som det har visats i tidigare kapitel viktiga för integration mellan olika system. Jag anser att UDDI i sin tur är viktigt för webbtjänster, eftersom det gör dem ännu kraftfullare. Löst kopplade webbtjänstsystem kan visserligen uppnås med egna implementationer, men UDDI är plattformsoberoende och den allmänt accepterade standarden, så varför lägga ner tid på att uppfinna hjulet igen, när UDDI fungerar så bra som det gör?

Det är dock värt att tänka på att kommunikationen med UDDI är kostsam. Av denna anledning bör lyckade sökresultat alltid sparas (cachas), så att de kan användas vid ett senare tillfälle, istället för att behöva anropa UDDI registret på nytt. UDDI bör endast kontaktas under körning om inget lokalt sparat sökresultat finns att tillgå, eller om webbtjänsten inte kan anropas.

UDDI är en kraftfull tjänst som kan användas till mycket mer än det som har beskrivits i detta kapitel. Det har dock inte funnits tid till ytterligare fördjupning och därför har fokuseringen riktats mot löst kopplade system.

9 Distribution av .NET

En applikation som har utvecklats i .NET kräver att .NET Framework finns installerat hos användaren som ska köra applikationen. Det är därför viktigt att inte bara applikationen, utan även .NET Framework, kan distribueras på ett enkelt och effektivt sätt. I .NET Framework ingår samtliga .NET klasser samt *Common Language Runtime (CLR)*, som är den exekveringsmotor som används för att köra .NET applikationer.

För närvarande finns .NET Framework i version 1.0, samt i en betaversion 1.1. Dessa båda versioner kan existera samtidigt hos en klient. Vilken version som används av en applikation bestäms av en förutbestämd policy eller via konfigurationsfiler. Om en applikation t.ex. är utvecklad i version 1.0 föredras denna version, men om den inte kan hittas väljs version 1.1. En applikation som är utvecklad i version 1.1 kan inte köras på .NET Framework version 1.0, om det inte anges något annat i en konfigurationsfil. Konfigurationsfiler har högre prioritet än den vanliga policyn och ibland kan det vara värdefullt att kunna specificera vilka versioner en viss applikation kan använda. I Figur 30 visas en konfigurationsfil där version 1.1 prövas först, men om denna inte kan hittas väljs version 1.0. Om inte heller denna hittas kan inte applikationen startas.

```
<configuration>
   <startup>
      <supportedRuntime version="v1.1.5000.0"/>
      <supportedRuntime version="v1.0.3300.0"/>
   </startup>
</configuration>
```

Figur 30: Konfigurationsfil där godkända versioner av .NET Framework specificeras

Attributet *supportedRuntime* stöds tyvärr inte av .NET version 1.0. Microsoft har för avsikt att släppa ett servicepack som rättar till detta problem, men tills dess kan en konfigurationsfil liknande den i Figur 31 användas.

```
<startup>
   <requiredRuntime version="v1.0.3300.0" safemode="true"/>
</startup>
```

Figur 31: Konfigurationsfil där den godkända versionen av .NET Framework anges

Version 1.1 ska vara bakåtkompatibel med version 1.0, men Microsoft garanterar inte att detta alltid gäller för framtida versioner. Java metoder som kommer att försvinna i framtida versioner av Java kan markeras som *Deprecated*. .NET har en liknande konstruktion i attributet *Obsolete*, som kan sättas framför en metod som troligtvis kommer att tas bort i senare versioner och därför bör undvikas. Kompilatorn ger en varning om utvecklaren försöker anropa en sådan metod. Detta kan underlätta för utveckling av applikationer som ska fungera med framtida versioner av .NET.

Eftersom .NET applikationer kräver olika versioner av .NET Framework för att fungera, är det praktiskt om de båda kan distribueras tillsammans och installeras via samma installationsprogram. Det går givetvis att kräva att användaren först manuellt installerar rätt version av .NET Framework, men det är ingen bra lösning. I nästa sektion beskrivs en applikation som löser detta problem. [22][35][36]

9.1 Implementation

Det finns bra stöd för paketering av applikationer i VS.NET. Genom att skapa ett ”Setup and Deployment” projekt är det möjligt att skapa ett *Windows Installer Package* (msi-fil) av applikationen. Denna fil kan sedan distribueras på t.ex. en CD tillsammans med en Setup-fil. Tyvärr är det inte möjligt att baka in .NET Framework i samma msi-fil och det är precis det som vore önskvärt.

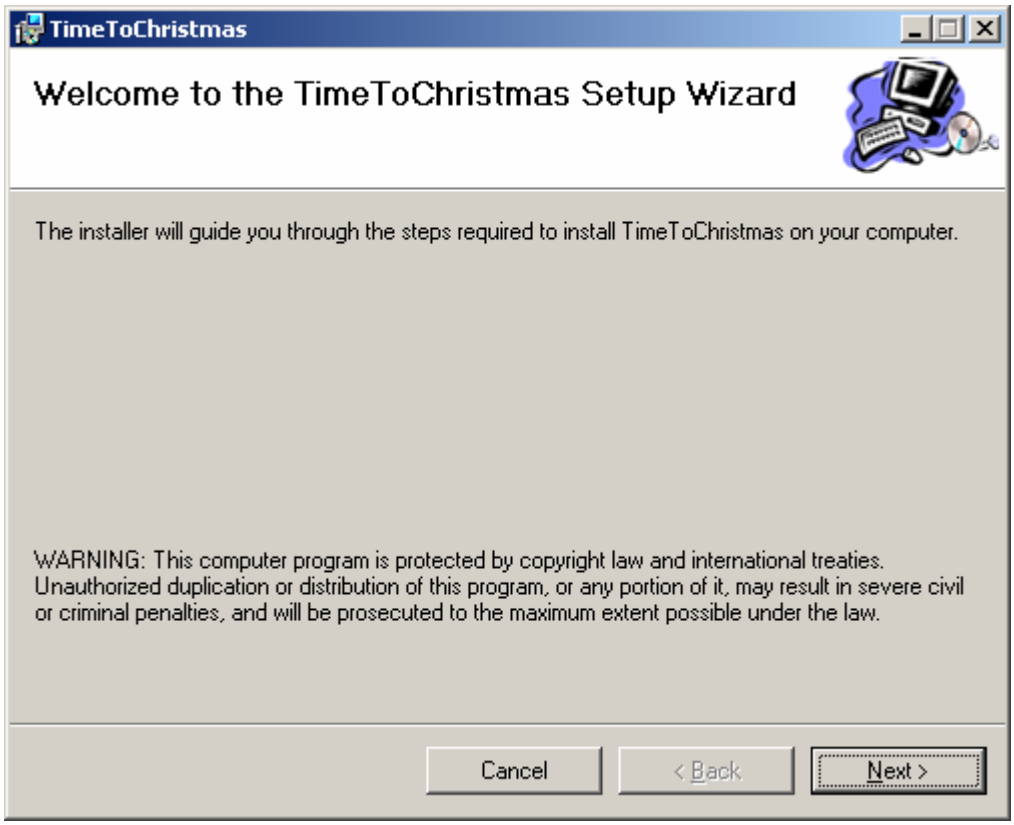
För att visa en lösning på detta problem har Microsoft implementerat en installationsapplikation (”Setup.exe Bootstrapper Sample”), som först kontrollerar om en viss version av .NET Framework finns installerad hos användaren. Om den inte hittas installeras den och därefter körs msi-filen som innehåller applikationen som ska installeras. .NET Framework kan laddas ner som en installerbar fil *dotnetfx.exe* och sökvägen till denna kan anges i installationsapplikationens fil *settings.ini*. I samma fil anges även sökvägen till msi-filen. Notera att administratörsrättigheter krävs för att köra filen *dotnetfx.exe*. Installationsapplikationens källkod kan laddas ner hos Microsoft och därför är det möjligt att modifiera den efter egna behov. Om en applikation t.ex. använder databasfunktioner krävs det att *Microsoft Data Access Components (MDAC)* 2.6 finns installerat. Det är då möjligt att lägga till kod i installationsapplikationen för att söka efter, och eventuellt installera MDAC, innan applikationen installeras.

För att testa Microsofts installationsapplikation utvecklades en enkel applikation, se Figur 32, som sedan paketerades i en msi-fil. För att denna ska kunna installeras måste .NET Framework version 1.0 finnas hos användaren. Detta kontrolleras som sagt av installationsapplikationen. Filen *dotnetfx.exe* laddades sedan ned och sökvägen till denna, samt sökvägen till msi-filen, angavs i filen *settings.ini*.

Figur 32: Applikation som talar om hur många dagar det är kvar till julafton

Samtliga filer brändes sedan till CD, eftersom det är det vanligaste mediet vid distribution av programvara. Efter detta kördes Setup-filen för installationsapplikationen på en dator som inte hade .NET Framework installerat. Installationsapplikationen upptäckte detta och körde därför automatiskt filen *dotnetfx.exe*. När installationen av .NET Framework var klar kördes msi-filen och då visades fönstret i Figur 33. I följande fönster kunde information anges om var på disken applikationen skulle installeras, samt vilka användare på datorn som skulle ha tillgång till den. Under installationen lades även en genväg till applikationen under Start–>Program. Installationen var därefter klar och applikationen kunde startas utan problem. Det gick sedan liksom för vanliga Windowsprogram att avinstallera applikationen och .NET Framework i kontrollpanelen under ”Lägg till/Ta bort program”.

Källkoden och alla filer finns att hämta under ”Dokument” på webbsidan för detta examensarbete, se [37]. [35]



Figur 33: Installationsprogram för applikationen TimeToChristmas

9.2 Slutsats

Efter att ha testat paketering, distribution och installation av .NET applikationer och .NET Framework i praktiken, vill jag påstå att det kan göras på ett både enkelt och effektivt sätt. Notera även att ”Setup and Deployment” projekt kan användas till att skapa betydligt mer avancerade och mer anpassade installationsprogram än det jag skapade till min enkla testapplikation.

Det är fullt möjligt att distribuera .NET applikationer och .NET Framework tillsammans och installera dem via samma installationsprogram. Det krävs dock en extern applikation för detta. Microsoft erbjuder en sådan installationsapplikation och den visade sig vara lätt att konfigurera och använda. Jag laddade även hem den fria källkoden för denna applikation. Den är skriven i Visual C++ och är inte alltför svår att modifiera om det skulle behövas.

Möjligheten att köra flera versioner av .NET Framework på samma maskin är bra och konfigureringen via XML-filer är lätt att förstå sig på. Kompatibilitetsproblemet med dessa filer och .NET Framework version 1.0 måste dock lösas innan de är till full belåtenhet.

10 Diskussion

Många anser att webbtjänster är framtiden för integration mellan olika system och eftersom de vanligtvis körs över SOAP protokollet, var detta den första fördjupning som gjordes, se kapitel 2. SOAP visade sig vara ett mycket intressant protokoll, trots att det saknar många av de funktioner som finns hos existerande RPC system. .NET har i grunden mycket bra stöd för SOAP och webbtjänster.

SOAP saknar tyvärr inbyggt stöd för säkerhet och därför gjordes en undersökning i kapitel 3, där möjligheterna för säkra webbtjänster togs upp. HTTP identifikation och SSL är den teknik som idag vanligtvis används. Den har dock ett antal begränsningar som nya tekniker som WS-Security försöker undvika. Tyvärr finns det idag ännu ingen accepterad standard för säkra webbtjänster och det är en klart bromsande faktor för spridningen av webbtjänster.

Vid kommunikation mellan .NET och Java är webbtjänster ändå det naturliga valet. I kapitel 4 implementerades därför ett antal scenarier med webbtjänster från de olika plattformarna. Kommunikationen fungerade bra i samtliga fall.

Kommunikation med COM komponenter är ett annat viktigt integreringsproblem. .NET löser även detta på ett tillfredsställande sätt, som det visades i kapitel 5.

En bra integreringsplattform bör ha ett väl inbyggt stöd för transaktionshantering och det har .NET, se kapitel 6. Databastransaktioner, manuella transaktioner och automatiska transaktioner kan implementeras på ett effektivt sätt. Det finns dock ännu inget stöd för distribuerade transaktioner i .NET.

I kapitel 7 undersöktes .NET Remoting, som är uppföljaren till DCOM. Det visade sig vara en mycket kraftfull och utbyggbar arkitektur, som i vissa situationer är att föredra framför webbtjänster. .NET Remoting kan erbjuda ett webbtjänstgränssnitt till objekten och det är väldigt intressant ur integreringssynpunkt.

Vid användningen av webbtjänster anges vanligtvis hårdkodade sökvägar. Detta är ofta opraktiskt och inte speciellt dynamiskt. I kapitel 8 undersöktes därför hur UDDI kan användas för att skapa löst kopplade webbtjänstsystem. UDDI är ingen .NET specifik teknologi, men det finns ändå ett bra stöd för UDDI i .NET.

I kapitel 9 visades till sist möjligheterna för distribution av .NET applikationer och .NET Framework. En integreringsplattform bör kunna distribueras och installeras tillsammans med sina applikationer, på ett enkelt och effektivt sätt och det kan .NET.

Målsättningen med examensarbetet var att undersöka och klargöra de olika punkter som nämndes under 1.2. Jag anser att denna rapport har uppfyllt målsättningen och dessutom visat att .NET är en mycket kraftfull och kompetent integreringsplattform.

11 Tack

Jag vill tacka mina handledare David Selin och Mathias Westin på DataVis, för givande diskussioner och bollande av idéer. Jag vill även tacka övriga anställda på DataVis i Örnsköldsvik, för ett trevligt och hjälpsamt bemötande.

12 Referenser

[1] Box Don, Ehnebuske David, Kakivaya Gopa, Layman Andrew, Mendelsohn Noah, Nielsen Henrik Frystyk, Thatte Satish, Winer Dave: *Simple Object Access Protocol (SOAP) 1.1*, 2000-05-08, <http://www.w3.org/TR/SOAP/>, 2002-09-03

[2] Box Don: *A Young Person's Guide to The Simple Object Access Protocol: SOAP Increases Interoperability Across Platforms and Languages*, 2000-03, <http://msdn.microsoft.com/msdnmag/issues/0300/soap/toc.asp>, 2002-09-03

[3] MSDN Training: *Developing XML Web Services Using Microsoft Visual C#.NET Beta 2*, 2001-10

[4] MSDN Library: *Customizing SOAP Messages*, 2001, <http://msdn.microsoft.com/library/en-us/cpguide/html/cpconcustomizingsoapinaspnetwebserviceswebserviceclients.asp>, 2002-09-07

[5] Biron Paul V., Malhotra Ashok: *XML Schema Part 2: Datatypes*, 2001-05-02, <http://www.w3.org/TR/xmlschema-2/>, 2002-09-05

[6] Seely Scott: *Designing Your Web Service for Maximum Interoperability*, 2001-12-05,<http://msdn.microsoft.com/library/en-us/dnservice/html/service12052001.asp>, 2002-09-07

[7] DeJesus Edmund X.: *Security Implications of Web Services*, 2001-06-06, <http://www.webservicesarchitect.com/content/articles/deJesus01.asp>, 2002-09-18

[8] Coulouris George, Dollimore Jean, Kindberg Tim: *Distributed Systems – Concepts and Design, Third Edition*, 2001 Addison-Wesley

[9] Powell Matt: *Real SOAP Security*, 2001-11-21, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnservice/html/service11212001.asp>, 2002-09-19

[10] Robinson Peter: *Understanding Digital Certificates and Secure Sockets Layer(SSL)*, 2001-01, <http://www.entrust.com/resources/pdf/understanding_ssl.pdf>, 2002-09-19

[11] Djajadinata Ray: *Yes, you can secure your Web services documents, Part 1*, 2002-08-23, <http://www.javaworld.com/javaworld/jw-08-2002/jw-0823-securexml.html>, 2002-09-19

[12] MSDN Library: *XML Web Services Security*, 2002-02, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwssecur/html/xmlwssec.asp>, 2002-09-19

[13] Bartel Mark, Boyer John, Fox Barb, LaMacchia Brian, Simon Ed: *XML-Signature Syntax and Processing*, 2001-08-20, <http://www.w3.org/TR/2001/PR-xmldsig-core-20010820>, 2002-09-20

[14]	Imamura Takeshi, Dillaway Blair, Schaad Jim, Simon Ed: *XML Encryption Syntax and Processing*, 2001-06-26, <http://www.w3.org/TR/2001/WD-xmlenc-core-20010626>, 2002-09-20
[15]	MSDN Library: *Security in a Web Services World: A Proposed Architecture and Roadmap*, 2002-04-07, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnwssecur/ html/securitywhitepaper.asp>, 2002-09-20
[16]	Apshankar Kapil: *WS-Security: Security for Web Services*, 2002-07-24, http://www.webservicesarchitect.com/content/articles/apshankar04.asp, 2002-09-20
[17]	Författare, se länk: *Web Services Security (WS-Security)*, 2002-04-05, <http://www-106.ibm.com/developerworks/library/ws-secure>, 2002-09-20
[18]	Nielsen Henrik Frystyk, Thatte Satish: *Web Services Routing Protocol (WS-Routing)*, 2001-10-23, - <http://msdn.microsoft.com/webservices/default.asp?pull=/library/en-us/dnglobspec/html/ws-routing.asp>, 2002-09-20
[19]	Powell Matt: *Using WS-Security with the Web Services Development Kit Technology Preview*, 2002-08, <http://msdn.microsoft.com/webservices/ building/wsdk/default.asp?pull=/library/en-us/dnwssecur/ html/wssecwithwsdk.asp#wssecwithwsdk_topic7>, 2002-09-25
[20]	Basiura Russ, Conway Richard, Gaster Brady, Kent Dan, Lakshminarayanan Sitaraman, Sabbadin Enrico, Seven Doug, Sivakumar Srinivasa: *Professional ASP.NET Security*, 2002-08 Wrox Press
[21]	Farrell Joel: *Web services interoperability between the WebSphere and .Net platforms*, 2002-08, <http://www-106.ibm.com/developerworks/webservices/library/i-wasnet>, 2002-10-14
[22]	Robinson Simon, Cornes Ollie, Glynn Jay, Harvey Burton, McQueen Craig, Moemeka Jerod, Nagel Christian, Skinner Morgan, Watson Karli: *Professional C#*, 2001 Wrox Press
[23]	Lhotka Rockford: *Call VB6 From .NET*, 2002, <http://www.123aspx.com/redir.aspx?res=28322>, 2002-10-16
[24]	MSDN Library: *Editing An Interop Assembly*, 2001, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/cpguide/html/cpconeditinginteropassembly.asp>, 2002-10-16
[25]	Dhawan Priya: *Transaction Control*, 2001-11, <http://msdn.microsoft.com/library/en-us/dnbda/html/bdadotnettransact1.asp>, 2002-10-23
[26]	Dhawan Priya: *Performance Comparison: Transaction Control*, 2002-02,<http://msdn.microsoft.com/library/en-us/dnbda/html/bdadotnetarch13.asp>, 2002-10-24

[27]	Cabrera Felipe, Copeland George, Cox Bill, Freund Tom, Klein Johannes, Storey Tony, Thatte Satish: *Web Services Transaction (WS-Transaction)*,2002,<http://msdn.microsoft.com/library/en-us/dnglobspec/html/ws-transaction.asp>, 2002-10-24
[28]	DeMichillie Greg: *WS-Transaction Specification Released*, 2002-09-23,<http://www.directionsonmicrosoft.com/sample/DOMIS/update/2002/10oct/1002gdffws_sb1.htm>, 2002-10-24
[29]	Rammer Ingo: *Advanced .NET Remoting*, 2002 Apress
[30]	Dhawan Priya, Ewald Tim: *ASP.NET Web Services or .NET Remoting: How to Choose*, 2002-09, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/ bdadotnetarch16.asp>, 2002-11-06
[31]	Dhawan Priya: *Performance Comparison: .NET Remoting vs. ASP.NET Web Services*, 2002-09, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnbda/html/bdadotnetarch14.asp>, 2002-11-07
[32]	Srinivasan Paddy: *An Introduction to Microsoft .NET Remoting Framework*, 2001-07, <http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dndotnet/html/introremoting.asp>, 2002-11-07
[33]	Januszewski Karsten: *Using UDDI at Run Time, Part I*, 2001-12, <http://msdn.microsoft.com/library/en-us/dnuddi/html/runtimeuddi1.asp>, 2002-11-20
[34]	Januszewski Karsten: *Web Service Description and Discovery Using UDDI, Part I*, 2001-10-03, <http://msdn.microsoft.com/library/en-us/dnservice/html/service10032001.asp>, 2002-11-20
[35]	MSDN Library: *Redistributing the .NET Framework*, 2002-01,<http://msdn.microsoft.com/library/en-us/dnnetdep/html/redistdeploy.asp>, 2002-11-27
[36]	Löwy Juval: *Prepare for .NET 1.1 and Beyond*, 2002-12, <http://www.fawcette.com/vsm/2002_12/magazine/columns/desktopdeveloper/default_pf.asp>, 2002-12-01
[37]	Forsberg Jonas: *Integrering med .NET*, 2003-01, <http://www.cs.umu.se/~c98jfg>, 2003-01-08

Bilaga A: Ordlista

.NET	Microsofts nya plattform för att skapa allt från webbtjänster och webbsidor till formulärbaserade Windowsapplikationer.
Active X	Speciell typ av COM komponent som stödjer olika gränssnitt, vilket gör att den kan användas i grafiska utvecklingsmiljöer.
ADO.NET	*Active Data Object.NET*, speciella klasser som används vid kommunikation med databaser som t.ex. SQL Server.
ASP.NET	Uppföljaren till ASP. Används för att bygga dynamiska webbsidor med kompilerad skriptkod i t.ex. C#.
C#	Uttalas C-sharp. Programmeringsspråk som har anpassats för .NET. Bygger vidare på C++ och Java.
CA	*Certification Authority*, myndighet eller tredjepartsföretag som kan garantera att ett visst digitalt certifikat är giltigt.
CAO	*Client Activated Objects*, .NET Remoting objekt som är unikt för varje klient. Sparar tillstånd.
CLR	*Common Language Runtime*, .NET:s exekveringsmotor. Laddar koden, sköter typkontroller, säkerhet, mm.
COM	*Component Object Model*, COM-kompatibla språk kan anropa COM-objekt skrivna i t.ex. C++, VB och Delphi.
COM+	Erbjuder utvecklaren tjänster som transaktionshantering, object pooling, Just-In-Time (JIT) object activation, mm.
CORBA	*Common Object Request Broker Architecture*, plattforms-oberoende arkitektur för att anropa objekt på en fjärrmaskin.
DCOM	*Distributed Component Object Model*, Microsofts gamla arkitektur för att anropa objekt på en fjärrmaskin.
Digest	Skapas av en hashfunktion. Olika texter ska resultera i olika digest och det ska vara omöjligt att få texten givet en digest.
Digital signatur	Säkrare än en handskriven signatur eftersom den baseras på innehållet i dokumentet som signeras.
Digitalt certifikat	Innehåller bl.a. en publik nyckel och ett namn, samt en signatur av en CA som garanterar bindningen mellan dessa.
DOS	*Denial of Service*, attack mot en server som går ut på att överösa den med meddelanden och göra den otillgänglig.
DTC	*Distributed Transaction Coordinator*, Windowskomponent som hanterar transaktioner mot SQL Server, MSMQ, m.fl.
GC	*Garbage Collector*, skräphanterare i .NET som rensar minnet (heapen) från objekt när de inte längre används.
GUI	*Graphical User Interface*, Grafiskt användargränssnitt för en godtycklig applikation.

GXA	*Global XML Web Services Architecture*, ett ramverk för olika protokoll som specificerar framtidens webbtjänster.
Hash-funktion	Funktion som givet en text skapar en unik digest av fix längd.
HTML	*HyperText Markup Langauge*, språk som beskriver utseende och innehåll på webbsidor.
HTTP	*HyperText Transfer Protocol*, det protokoll som används på Internet idag för att överföra olika dokument, t.ex. HTML.
HTTP-EF	*HTTP-Extension Framework*, utökning av HTTP med tillägg för bl.a. obligatoriska attribut i huvudfältet.
IIOP	*Internet Inter Orb Protocol*, protokoll som beskriver formatet på meddelanden som skickas av CORBA.
IIS	*Internet Information Server*, Microsofts webbserver. Har bl.a. inbyggt stöd för skalbarhet och säkerhet.
Java RMI	*Java Remote Method Invocation*, Javas arkitektur för att anropa objekt på en fjärrmaskin.
MDAC	*Microsoft Data Access Components*, komponenter som behövs vid databaskommunikation, t.ex. ADO och OLE DB.
Message passing	System där olika parter kommunicerar via asynkrona meddelanden. MSMQ är ett sådant system.
MSIL	*Microsoft Intermediate Language*, det lågnivåspråk som t.ex. C# och VB.NET kompileras till. Motsvarar Javas byte code.
MSMQ	*Microsoft Message Queue*, Message passing system som tillåter kommunikation via asynkrona meddelanden.
ORB	*Object Request Broker*, tar emot ett CORBA meddelande och ser till att rätt objekt på servern anropas.
Passport	Microsofts standard för att uppnå målet med en inloggning för alla tjänster på Internet.
PIA	*Primary Interop Assembly*, en speciell typ av RCW som delas av flera olika .NET applikationer.
RCW	*Runtime Callable Wrapper*, proxyklass som erbjuder ett lättanvänt gränssnitt mot en inkapslad COM-komponent.
RPC	*Remote Procedure Call*, system som tillåter metodanrop till objekt på fjärrmaskiner.
SAO	*Server Activated Objects*, .NET Remoting objekt som delas (Singleton) eller nyskapas (SingleCall) för varje klientanrop.
Sessions-nyckel	Symmetrisk nyckel som nyskapas för en tillfällig, tidsbegränsad användningsperiod.
SMTP	*Simple Mail Transfer Protocol*, protokoll för att skicka mail.
SOAP	*Simple Objekt Access Protocol*, XML baserat protokoll som bl.a. används av webbtjänster för RPC anrop.

SQL	*Structured Query Language*, språk som används för att hämta och lagra data på databaser.
SQL Server	Kraftfull databasserver som t.ex. innehåller funktioner för säkerhet och transaktionshantering.
SSL	*Secure Socket Layer*, säker kommunikation över t.ex. HTTP.
TCP	*Transmission Control Protocol*, ligger under HTTP och sköter återsändningar av borttappade TCP-paket.
Triple-DES	*Triple-Data Encryption Standard*, vanlig symmetrisk krypteringsalgoritm. Använder en 128 bitars nyckel.
T-SQL	*Transact-SQL*, utökning av SQL med stöd för transaktioner.
UDDI	*Universal Description Discovery and Integration*, ett register med webbtjänster. Webbtjänsternas ”Gula sidor”.
URI	*Universal Resource Identifier*, pekar ut adressen till en resurs på webben.
VS.NET	*Visual Studio.NET*, Microsofts utvecklingsmiljö för .NET baserade applikationer. Stödjer bl.a. C++, C#, VB.NET och ASP.NET.
W3C	*World Wide Web Consortium*, standardiseringsorgan för Internetteknologier.
WSDK	*Web Services Development Kit*, .NET bibliotek som bl.a. stödjer WS-Security.
WSDL	*Web Service Description Language*, beskriver gränssnittet för en webbtjänst med hjälp av ett XSD schema.
XML	*eXstensible Markup Language*, syntax som gör att data kan representeras på ett strukturerat sätt.
XSD	*XML Schema Definition*, beskriver strukturen för ett XML dokument, elementens ordning, datatyper, osv.

Bilaga B: Källkod

```
CREATE PROCEDURE Transfer
          (@AMOUNT money,
           @KONTONR_1 bigint,
           @KONTONR_2 bigint) AS

SET NOCOUNT OFF;
SET XACT_ABORT ON

BEGIN TRANSACTION

UPDATE KONTON
SET SALDO = SALDO - @AMOUNT
WHERE KONTONR = @KONTONR_1;

UPDATE KONTON
SET SALDO = SALDO + @AMOUNT
WHERE KONTONR = @KONTONR_2;

COMMIT TRANSACTION

GO
```

Figur 1: Lagrad procedur som utför en databastransaktion med hjälp av T-SQL

```
using System;
using System.Data.SqlClient;

namespace ADO_NET
{
  class TransMain
  {
    [STAThread]
    static void Main(string[] args)
    {
      string source = "server=localhost;Trusted_Connection=true;" +
                      "user id=Jonas;password=opensesam;database=Banken";
      SqlConnection conn = new SqlConnection(source);
      SqlTransaction trans = null;

      Console.Write("Enter amount of money to be transfered: ");
      string amount = Console.ReadLine();
      Console.Write("Enter name of first account: ");
      string firstAccount = Console.ReadLine();
      Console.Write("Enter name of second account: ");
      string secondAccount = Console.ReadLine();

      // Withdraw money from the first account
      string firstUpdate = "UPDATE KONTON " +
                           "SET SALDO = SALDO - " + amount +
                           " WHERE KONTONR = " + firstAccount;

      // Insert money to the second account
      string secondUpdate = "UPDATE KONTON " +
                            "SET SALDO = SALDO + " + amount +
                            " WHERE KONTONR = " + secondAccount;
      try
      {
        conn.Open();
        Console.WriteLine("\nConnection open.");

        trans = conn.BeginTransaction();
        Console.WriteLine("Transaction started.");
```

```
        // Execute the first update
        SqlCommand cmd = new SqlCommand(firstUpdate, conn, trans);
        if (cmd.ExecuteNonQuery() == 0)
         throw new Exception(String.Format("Account \"{0}\" was not found.",
                             firstAccount));
        Console.WriteLine("Account \"{0}\" was updated.", firstAccount);
        Console.Write("Press enter to proceed...");
        Console.ReadLine();

        // Execute the second update
        cmd.CommandText = secondUpdate;
        if (cmd.ExecuteNonQuery() == 0)
         throw new Exception(String.Format("Account \"{0}\" was not found.",
                             secondAccount));
        Console.WriteLine("Account  \"{0}\"  was  updated.",  secondAccount);

        trans.Commit();  // Commit the transaction
      }
      catch(Exception e)
      {
        if (trans != null)
        {
          trans.Rollback();
          Console.WriteLine("Transaction was rollbacked.");
        }

        Console.WriteLine(e.Message);
      }
      finally
      {
        conn.Close();
        Console.WriteLine("\nConnection closed.");
        Console.Write("Press enter to quit...");
        Console.Read();
      }
    }
  }
}
```

Figur 2: .NET klass som utför en manuell transaktion med hjälp av ADO.NET

```
using System;
using System.EnterpriseServices;
using System.Collections;
using System.ComponentModel;
using System.Data;
using System.Diagnostics;
using System.Web;
using System.Web.Services;
using System.Data.SqlClient;

namespace COMPlusTrans
{
  [WebService(Namespace="http://www.datavis.se")]
  public class COMPlusTrans : System.Web.Services.WebService
  {
    [WebMethod(TransactionOption = TransactionOption.Required)]
    public string Transfer(string amount, string firstAccount, string
                           secondAccount)
    {
      string source = "server=localhost;Trusted_Connection=true;
                       database=Banken";

      SqlConnection conn = new SqlConnection(source);
```

```
      // Withdraw money from the first account
      string firstUpdate = "UPDATE KONTON " +
                           "SET SALDO = SALDO - " + amount +
                           " WHERE KONTONR = " + firstAccount;

      // Insert money to the second account
      string secondUpdate = "UPDATE KONTON " +
                            "SET SALDO = SALDO + " + amount +
                            " WHERE KONTONR = " + secondAccount;
      try
      {
        conn.Open();

        // Execute the first update
        SqlCommand cmd = new SqlCommand(firstUpdate, conn);
        if (cmd.ExecuteNonQuery() == 0)
          throw new Exception(String.Format("Account \"{0}\" was not
                              found.", firstAccount));

        // Execute the second update
        cmd.CommandText = secondUpdate;
        if (cmd.ExecuteNonQuery() == 0)
          throw new Exception(String.Format("Account \"{0}\" was not
                              found.", secondAccount));

        ContextUtil.SetComplete();  // Commit the transaction
      }
      catch(Exception e)
      {
        ContextUtil.SetAbort();  // Rollback the Transaction
        return "Transaction was rollbacked: " + e.Message;
      }
      finally
      {
        conn.Close();
      }

      return "Transaction committed succesfully!";
    }
  }
}
```

Figur 3: Webbtjänst som utför en automatisk transaktion med hjälp av COM+