Master Thesis
Software Engineering
Thesis no: MSE-2006:15
August 2006

Comparison of J2EE and .NET from a Web Services point of view.

Andreas Areskoug

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

This thesis is submitted to the School of Engineering at Blekinge Institute of Technology in partial fulfillment of the requirements for the degree of Master of Science in Software Engineering. The thesis is equivalent to 20 weeks of full time studies.

Contact Information:
Author:
Andreas Areskoug
E-mail: andreas.areskoug@telia.com

University advisor:
Fredrik Wernstedt
School of Engineering

School of Engineering
Blekinge Institute of Technology
Box 520
SE – 372 25 Ronneby
Sweden

Internet : www.bth.se/tek
Phone : +46 457 38 50 00
Fax : + 46 457 271 25

ABSTRACT

This paper can be of interest for anyone interested in Web Services, (such as software developers and designers), companies choosing between the J2EE and .NET platforms to base their Web Services on etc.

The main purpose of Web Services is to achieve a higher level of interoperability between different programming languages, operating systems and platforms than was achieved with older technologies for distributed computer communication such as CORBA, Java RMI, and Microsoft DCOM. But Web Services is much more than just a new technology for achieving distributed computer communication. It is more like an architecture, defining how applications operating over networks such as the Internet is structured as well as how they communicate.

There are several available platforms/technologies to base Web Services on but this thesis will look at the two dominant platforms used today, J2EE from Sun Microsystems and .NET from Microsoft. The overall goal of this thesis is to compare Web Services based on both these platforms, considering mainly performance, through an experiment to see which one performs 'best'. The conclusion is that one of the platforms isn't superior to the other one. J2EE is fastest when a Web Service sorting data is used but .NET is fastest when just sending data back and forth between a client and the Web Service without the Web Service doing anything. When sending data between the platforms J2EE performs 'best'. This indicates that Web Services should be based on J2EE if the clients are based on both platforms. The main conclusion is therefore that other aspects such as the hardware and software already present in a company influences the decision more than what the performance aspect does.

Keywords: Web Services, .NET, J2EE

CONTENTS

1 INTRODUCTION

1.1 Background

Web Services is one of the latest innovations when it comes to achieving distributed computer communication. Web Services has mainly been developed to create interoperability between different programming languages and platforms. This high level of interoperability has not fully been achieved by existing methods for distributed computer communication such as CORBA, RMI, and DCOM. The older methods never gained the needed support from the software engineering industry to achieve wide level interoperability. Web Services and most of its supporting technologies are developed and standardized by the World Wide Web Consortium (W3C). W3C is supported by all major software vendors such as Microsoft and Sun Microsystems.

Web Services is a new approach to distributed computer communication and is still under development. There is therefore a need for continuous investigation and evaluation. The Java 2 Enterprise Edition (J2EE) from Sun Microsystems and the .NET platform from Microsoft are the most common platforms used today for implementing and running Web Services on. Both these two platforms have been selected as the platforms to use for implementing and running the Web Services during the experiment phase of this thesis.

1.2 Questions

The question that this thesis will answer is:

- What is the performance of Web Services in J2EE compared to .NET when the clients are a mix of J2EE and .NET based ones?

The purpose is to find out which platform should be used to run Web Services on, considering performance, when the client are a mix of J2EE and .NET based clients. Consider e.g., a company that have decided to invest in Web Services and currently are trying to decide which platform (.NET or J2EE) they should use to host their future Web Services on. These Web Services are going to be accessed by several other companies so the clients are going to be a mix of J2EE and .NET based ones.

1.3 Methods used

First a literature study was performed to collect general information abut Web Services. Then a series of experiments were conducted to evaluate the performance of J2EE and .NET-based Web Services in reality in different situations.

1.4 Related work

Since Web Services are such a new technology not that many relevant works exist and the work that do exist is made obsolete quickly because of the rapid development that is currently going on.

One work that heavily influenced this work was another master thesis done at BTH called "Comparison of Enterprise Java Beans and .NET from a component point of view", (Persson 2003). The focus of this work is to examine the component models present in J2EE and .NET. The work doesn't consider Web Services although it mentions it as a way of achieving interoperability between the platforms and should therefore be considered by future work. What makes this work interesting is that it examines the same platforms as this work (J2EE and .NET) and compares the performance of the platforms respectively native mean of communication (when just communicating 'inside' the platform i.e. no interoperability is needed). The conclusions drawn by the work is that .NET is 'much' slower than J2EE but on the other hand it always delivers although sometimes it can take a while. J2EE is superior in all evaluated areas except one, it needs more resources and may therefore cut clients of when not getting enough.

Another master thesis also done at BTH with more focus on Web Services is called "Web Services – aspects on new business possibilities", (Petrovic et. al. 2002). This thesis focuses on the interoperability aspect of Web Services and concludes that it's possible to minimize the interoperability problem by using the most standardized protocols and following certain guidelines when creating the Web Services. They also concluded that it is possible to test the level of interoperability by using certain test tools such as XMLBus. This is important to know by developers in order to minimize the problems when launching the service.

A work focused on the performance of Web Services is done by Microsoft and called "Performance of ASP.NET Web Services, Enterprise Services, and .NET Remoting", (Microsoft 2005). The purpose of this work is to compare and contrast performance characteristics of real-life ASP.NET Web Services, .NET Enterprise Services components, and .NET Remoting components. Some of the most important observations made are:

- ASMX (ASP.NET based Web Services), is not the fastest technology, but its performance is more than adequate for most business scenarios. Combined with the facts that ASMX services are easy to scale out and provide a wealth of other benefits such as interoperability abd future compatability, ASMX should be the primary choice for building services today.

- If performance is of outmost concern, Enterprise Services components should be used to build the most time-critical portions of a system. The rest should be build using ASMX.

1.5 Limitations

In order to limit the scope of this thesis only the J2EE and .NET platforms are considered, mainly because they are the most frequently used platforms to run Web Services on available today. Also the main aspect to look at when evaluating Web Services in this thesis is performance since more factors would take too much time to do. The software used, everything from the operating system to the web server, is also limited and described in Chapter 3 since using other types of software or running the software on different platforms would take a lot more time as well and the end result will be much harder to evaluate and to compare. Also worth noting is that this thesis is about Web Services which means that the other forms of communication available on the J2EE and .NET platforms are not considered.

2 TECHNOLOGY

2.1 SOA

Before defining Web Services an introduction to SOA might be useful since most information about Web Services mention it in one way or the other. SOA, or Service Oriented Architecture, is one of the most used buzz words today although the ideas behind SOA are not anything new. A software system based on SOA is, just as the name implies, divided into a set of services and a set of consumers or clients consuming these services. Several parallels can be drawn between these services and the notation of objects and components that are widely spread in the software community, (Microsoft 2004). Just as with objects and components, the main idea behind services is to provide natural building blocks that allows us humans to organize a software application in ways that are familiar and feels more natural to us. Just as objects, services combine information and behaviour, hides the internal workings from outside intrusion and presents a relatively simple interface towards the outside world, (Microsoft 2004). A good way to understand the service concept is simply to look at how humans work. Humans constantly request services and these are often provided, not just by a single person, but several persons working together to provide the service. SOA is basically the same approach in the software context. Client's requests services and these services are often not just provided by a single object or component. Instead several objects and/or components work together to provide a service and just as humans get different services from different sources clients can requests services from different providers which means a consumer can choose the service that fits its needs best. The difference between SOA and Web Services, which is the main topic in this report, is very subtle and therefore hard to explain. The most important difference is that SOA is on a more abstract level than Web Services. SOA is an architecture consisting of clients communicating with different types of services and is very general. Web Services can also be considered to be a sort of an architecture, (which is argued later in the report), since it isn't dependent on any particular set of technologies (although it wouldn't exists in any concrete form without them). But Web Services are more concrete than SOA since it is limited in various ways such as it only supports machine-to-machine interaction (explained in the next section) and the services are offered through the Web which means standard Internet protocols has to be used. Web Services can simply be though of as the first true SOA implementation today although other techniques exist. Since it is very hard to define the border between SOA and Web Services only the term Web Services is used in this thesis although the architecture behind Web Services is essentially SOA.

2.2 Definition of Web Services

The W3C once tried to define the term Web Service (Webservices.org 2002). In February 2002 the W3C Web Services Architecture Working Group exchanged almost 400 e-mails over a two week period to agree on a definition of the term. Eventually they abandoned the effort to reach consensus. Instead of creating yet another definition of Web Services, let's look at some earlier definitions from companies/organizations working with them one way or the other and identify the main characteristics. These definitions are not selected randomly or because they give a certain picture that suits this thesis. The first definition comes from W3C because they are responsible for the development and standardization of some key technologies used today to create Web Services. The second and third definitions come from Microsoft and Sun Microsystems since their platforms, .NET and J2EE respectively, are the main platforms for building and maintaining Web Services in use today. The fourth definition is taken from a well known software book (Szyperski et al. 2002), and the fifth

and final definition is taken from a programming book (Allamaraju et al. 2001). The last book has been selected since it implements real Web Services and thereby gives a more practical picture in contrast to the previous definitions which concentrate on how Web Services work in theory.

- The first definition from W3C (W3C 2004a) is:
  A Web service is a software system designed to support interoperable machine-to-machine interaction over a network. It has an interface described in a machine-processable format (specifically WSDL). Other systems interact with the Web service in a manner prescribed by its description using SOAP messages, typically conveyed using HTTP with an XML serialization in conjunction with other Web-related standards.

- The second definition from Microsoft (Microsoft 2001) is:
  A Web Service is programmable application logic accessible using standard Internet protocols. Web services combine the best aspects of component-based development and the Web. Like components, Web services represent black-box functionality that can be reused without worrying about how the service is implemented. Unlike current component technologies, Web services are not accessed via object-model-specific protocols, such as the Distributed Component Object Model (DCOM), Remote Method Invocation (RMI), or Internet Inter-ORB Protocol (IIOP). Instead, Web services are accessed via ubiquitous Web protocols and data formats, such as HyperText Transfer Protocol (HTTP) and Extensible Markup Language (XML). Furthermore, a Web Service interface is defined strictly in terms of the messages the Web Service accepts and generates. Consumers of the Web Service can be implemented on any platform in any programming language, as long as they can create and consume the messages defined for the Web Service interface.

- The third definition from Sun Microsystems (Sun Microsystems 2001) is:
  Web services are application components that are designed to support interoperable machine-to-machine interaction over a network. This interoperability is gained through a set of XML-based open standards, such as the Web Services Description Language (WSDL), the Simple Object Access Protocol (SOAP), and Universal Description, Discovery, and Integration (UDDI). These standards provide a common and interoperable approach for defining, publishing, and using web services.

- The fourth definition from a software book (Szyperski et al. 2002) is:
  Web services are services offered through the web. Unlike traditional web servers that serve web pages for consumption by people, web services offer computational services to other systems.

- Finally the fifth definition that comes a programming book (Allamaraju et al. 2001) is:
  An application component accessible via standard web protocols.
  A web service is like a unit of application logic. It provides services and data to remote clients and other applications. Remote clients and applications access web services via ubiquitous Internet protocols. They use XML for data transport, and SOAP for using services. Due to the use of XML and SOAP, accessing the service is independent of the implementation. Thus, a web service is like a component architecture for the web.

A number of common characteristics can be identified from these definitions:

- Machine-to-machine interaction
- Distributed over a network
- Services offered through the web
- Using standard Internet protocols
- A unit of application logic
- XML-based open standards

The first characteristic, machine-to-machine interaction is what distinguishes Web services from other types of Web-based applications. Other Web applications support human-to-human communication, e.g., email and instant messaging, or human-to-machine communication, e.g., browsers. The next characteristic is that Web services are distributed in a network. Web services are in a sense about providing access to services located somewhere in a network.

The third characteristic is that services are offered through the Web (read Internet). A service is some computational service like calculating or providing access to some stored information. The service exposes its functionality through a programmatic interface which means it more or less has to be accessed by another application and not by a human. Since Web services are to be offered through the Internet, the next characteristic comes automatically. Web Services uses standard Internet protocols, e.g., HTTP and TCP/IP.

The fifth characteristic claims that a Web Service is a unit of application logic. This simply means that there has to be some application logic behind the Web Service. The Web Service has to provide a service like storing/retrieving data or to compute data sent to it. The definition from Sun Microsystems, (Sun Microsystems 2001), claims that Web Services are application components but, as the definition from Microsoft (Microsoft 2001) implies, Web Services are not components in the traditional sense. Microsoft (Microsoft 2001) said that components are similar to Web services since they both represent black-box functionality that can be reused without worrying about how the actual service is implemented. But unlike current component technologies, Web services are not accessed via object-model-specific protocols such as DCOM, RMI, IIOP etc. Instead they are accessed via Web protocols and formats, such as HTTP, and XML. Szyperski (Szyperski et al. 2002) agrees with Microsoft that Web services are not components. Traditional components can, according to Szyperski (Szyperski 2002), be modified to be used in many different contexts. Web services can't be reused through modification as traditional components; instead they are good as they are and should no be modified.

The last and final characteristic connects Web Services to XML-based standards. XML is a widely accepted industry standard which is used today by Web Services as the message data format. Almost all information about Web Services mentions XML in one way or the other. Web Services are however not tied by definition to XML. Web Services represents an architecture and not a particular set of specific technologies such as XML although Web Services are limited in various ways such as using Internet protocols. Sources like (Szyperski et al. 2002) and (Muschamp P. 2004) basically claim that XML is what made Web Services practically possible today. Web Services are about providing services through the Internet and has nothing to do with how an actual Web Service should be implemented or which technologies to use.

2.3 Historical perspective on application architectures

In the previous section a Web Service was seen more as a concept or an architecture than a set of specific technologies. Webservices.org discusses how Web Services fit into the evolution of software and argues that Web Services is an architecture (Webservices.org 2003). In this discussion the evolution of software architectures can roughly be divided into three generations (Webservices.org 2003). The first generation is simple client-server applications which will be discussed in the next sections. The second generation is web application servers and will be discussed in section 2.2.2. The third and final generation is Web Services which will be discussed in section 2.2.3.

2.3.1 First Generation: Databases and Client-Server Application

According to Webservices.org (Webservices.org 2003), the first generation of client-server applications were built on a basic client-database design. In the client-database design the application were divided into two parts instead of the older designs were the whole application were running on the same computer, e.g., a mainframe. The application consisted of a client application running on a PC issuing requests to a database using e.g., the Structured Query Language (SQL). This client-server model, referred to as a two-tier model by Allamaraju (Allamaraju et al. 2001), had according to Webservices.org (Webservices.org 2003) a number of advantages. Since the database is separate from the client, standardized database products emerged during the 1980's. This enabled the application developers to use Commercial-Of-The-Shelf (COTS) databases instead of developing everything from scratch every time new applications were created. The standardization of databases together with the major advancements in computer hardware with lower computer prices and higher computing power enabled a rapid development of database driven applications. The client could instead focus on presenting the data and on business logic. A client containing presentation and business logic is referred to as a fat client. There were also according to Allamaraju (Allamaraju et al. 2001) some disadvantages with the client-server approach. Often the client computer had limited computational resources for the fat client to run on and the communication between the client and the server tend to increase since the client had to do multiple requests for data before presenting anything to the end-user since the computation had to be done on the client machine. Another disadvantage is the maintenance of the application. If the application is updated all clients had to be changed individually.

A client-server scenario

(1987) A company called Modern Jackets uses a client-database application to keep track of their inventory. Before using this system the employees had to run to the warehouse to manually see if a jacket is currently in store. The store manager bought a complete inventory systems based on a database packet that could be easily customized to store data types needed by the store such as size, colour, style etc. Since they bought a general database packet the price was affordable but the system had some limitations. The inventory system could be accessed by many clients but it required a very modern PC and all clients had to be connected to the company's internal network.

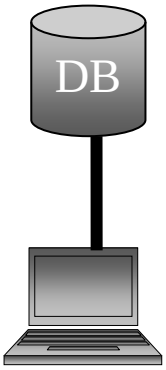


Fig.2.1 An example of a client-server architecture.

2.3.2 Second Generation: Web Application Servers

The introduction of Internet in the mid-1990's created a lot of new opportunities. According to Webservices.org (Webservices.org 2003) the biggest difference between the previous described model that used a dedicated communication line between the client and the server and Web applications, is that Web applications are designed to communicate "through the cloud", i.e. over the Internet. In this new model a client can access a server using Internet standards such as HTTP over TCP/IP, no matter if the server is located in the same network as the client or if it is located in another network on the other side of the globe. The largest first generation client-server applications were designed to support hundreds, or even up to, thousands of simultaneous clients. The introduction of Internet exposed applications to hundreds of thousands, or even millions of simultaneous clients.

A web server scenario.

(1996) Now Modern Jackets has begun to use a Web-based inventory system. The system can be accessed through an intranet by any of its branches. The company's stores just need access to Internet to be able to have a secure access to the inventory system. The employees in the stores can now reserve a Jacket for a customer and have it delivered to the store in 5 working days.

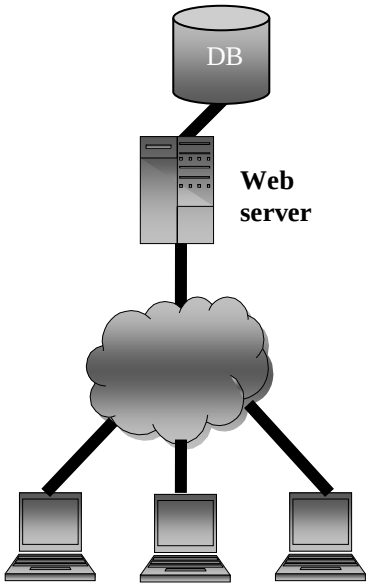


Fig.2.2 An example of an architecture based on a web server.

Unlike dedicated PC clients, Web browsers have no inherent knowledge about the system they are communicating with because of the stateless nature of the protocols. Therefore, according to Webservices.org (Webservices.org 2003), the high workload placed on the web

servers required the servers to be very scalable to handle the increased number of requests from coming in from the clients. This increased demand for processing power combined with a greater need for adaptability and flexibility lead to a new type of architecture for web applications. First application servers emerged to provide a common platform for a wide variety of specialized applications. Then based on these applications servers Sun Microsystems J2EE platform was released and followed by Microsoft's .NET platform. These platforms provide a complete environment for application development, Web-based communication, and database access. The platforms contain a component-based architecture for adding, removing and updating parts of the application with minimal disruption of the system. These platforms bring together a wide range of standards for networking, e.g. TCP/IP and HTTP, for presentation, e.g., HTML, for application logic, e.g. Java, JavaScript, CGI, etc.

A web application scenario.

(1998) Modern Jackets has now put its store online, making it possible for customers to order jackets directly from the company Web site. The company bought a standardized software application that provides functions such as search, shopping cart, transaction security, express delivery options, and e-mail confirmations.

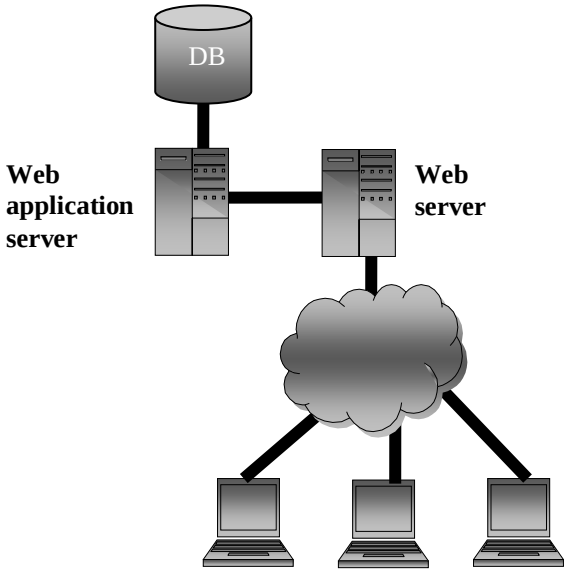


Fig.2.3 An example of an architecture based on a web application server.

2.3.3 Third Generation: Web Services

According to Webservices.org (Webservices.org 2003) Web Services represents the latest model for software architecture. It is based on the convergence of two technologies. The first technology is the Internet with its standard communication protocols such as HTTP. The second one is Service Oriented Computing (SOC), where data and business logic is exposed through programmatic interfaces.

The difference between Web Services and traditional Web applications can best be described by the following points (Webservices.org 2003):

- Unlike Web applications, a Web Service client doesn't have to be a browser on a PC. Instead the discovery of Web Service can be made by any Internet-enabled application and device. For example a Web Service could be accessed from a cell phone using the Wireless Markup Language (WML).

- The user or operator of a client doesn't have to always be a person. Instead it can be another Web Service or application. A Web Service can be designed just to summaries other Web Services. For example, a Web Service that keeps track of the current stock price for a certain company can call the Web Service belonging to the stock exchange the company is registered in as well as other Web Services.
- The accessible information is much broader to support a wider range of clients and applications. Unlike HTML, which is designed specifically for browsers, a Web Service can request that information be transported using any protocol (SOAP, ebXML, etc.) and presented in any format.
- The exchanged information between the client and the Web Service is much deeper, as it not only gives a pinpoint description of the application, but also precisely instructs the application on what action to take, i.e. what data should be processed and how the response should be delivered.
- A Web Service may issue requests to other Web Services, which can issue request to other Web Services and so on. So an application can simply be based on the Web Services already provided by other applications.
- Finally, the interaction doesn't have to end at the system that initiates the Web Service transaction. This is a big difference compared to the client-server model.

A Web Services scenario.

(2002) The modern Jackets company has now integrated its online store with its retailers by exposing its inventory system as a Web Service. When a customer looks after a specific Jacket on the company's online store, the Web Service offers to look for them by using the customer's preferences such as style, size, and colour to find a store with the Jacket closest to the customer's specified location. Unlike the company's inventory system, which is designed to search for a single product in a single location, the Web Service can perform a wide range of queries across a number of parameters, and return the results based on the customers expressed preferences. In this way the company can use the Web Service to better draw customers into its retail locations.

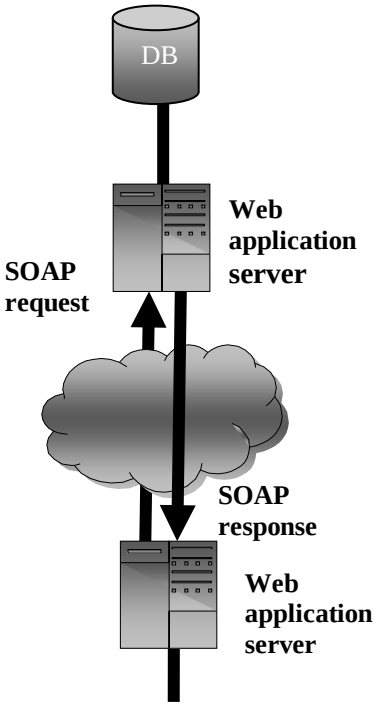


Fig.2.4 An example of an architecture based on Web Services.

2.4 Web Service technologies

Just as other technologies for achieving distributed communication, the Web Service technology consists of three parts (Borland 2002):

- A service provider
- A service requestor
- A service broker.

The service provider is simply the provider of the Web Service. The service requestor is the client requesting the Web Service. The problem with this approach is that the service provider can't just provide a service. Potential clients must be able to find and know how to communicate with the service. Therefore a service broker is needed. Its purpose is to act as a look up service between the service requestor and the service provider so they cam find each other.

Fig 2.5 below illustrates the relationship between the Web Service components.

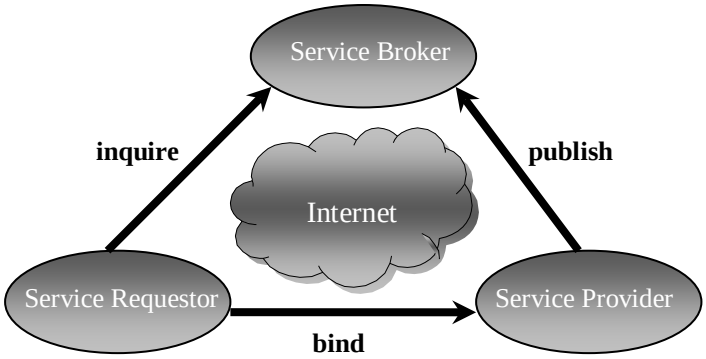


Fig.2.5 The relationship between Web Service components.

According to Borland (Borland 2002) Web Services are made up of four essential components which form the so called Web Services stack. These four components are:

- XML (Extensible Markup Language)
- SOAP (Simple Object Access Protocol)
- WSDL (Web Services Definition Language)
- UDDI (Universal Discovery, Description, and Integration)

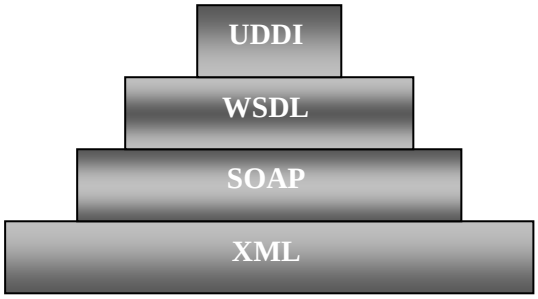


Fig 2.6 The Web Services stack.

2.4.1 SOAP

SOAP originally stood for Simple Object Access Protocol and was initially intended to be used for accessing objects (Microsoft 2003a). But after some time it was felt that SOAP was too restricted with only accessing objects. To gain the acceptance of a wider audience the SOAP specification moved from an object-centric one to a generalized XML messaging framework. This created a problem with the 'O' in the SOAP acronym. Since the name SOAP was so popular the W3C decided to keep the name but to stop spelling it out to avoid misunderstandings. A definition of SOAP from the recommended SOAP 1.2 specification (W3C 2003a) looks like this:

"SOAP Version 1.2 (SOAP) is a lightweight protocol intended for exchanging structured information in a decentralized, distributed environment. It uses XML technologies to define an extensible messaging framework providing a message construct that can be exchanged over a variety of underlying protocols. The framework has been designed to be independent of any particular programming model and other implementation specific semantics."

According to Microsoft (Microsoft 2003a) SOAP essentially provides a way to move XML messages from point A to point B (see Fig 2.7) by providing an XML-based messaging framework that is 1) extensible, 2) usable over a variety of underlying networking protocols, and 3) independent of programming models.

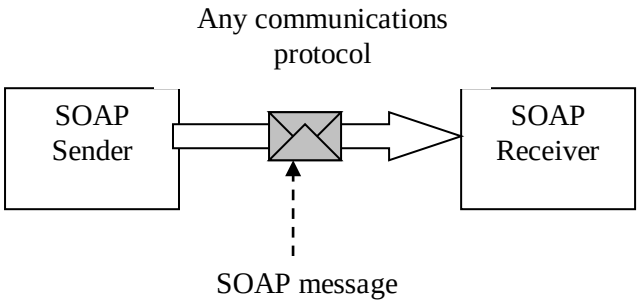


Fig 2.7 Simple SOAP messaging.

When the acronym SOAP stood for something, the 'S' stood for Simple. Microsoft (Microsoft 2003a) thinks that the most important lesson they have learned from the Web, is that simplicity always wins over efficiency or technical purity, and when interoperability is at stake, it's an absolute requirement. One of SOAP's primary design goals are simplicity which SOAP's lack of various distributed system features such as security, routing, and reliability just to name a few, is evidence of. Therefore software vendors such as Microsoft and IBM are working on a common suit of SOAP extensions that will add many of these features that developers lack. The initiative of creating SOAP extensions is referred to as the Global XML Web Services Architecture (GXA) by Microsoft.

SOAP can be used over a variety of transport protocols such as TCP, HTTP, SMTP, or even MSMQ, (Microsoft 2003a). To maintain interoperability, standard protocol bindings need to be defined that outlines the rules for each environment. Currently the SOAP v.1.2 specification provides explicit bindings for both HTTP and SMTP (W3C 2003b). The specification also provides a flexible framework for defining arbitrary protocol bindings. Defining other bindings than HTTP and SMTP is currently entirely up to the application developers.

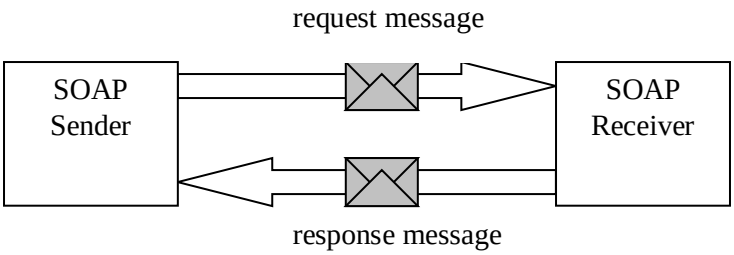


Fig 2.8 A request/response message pattern.

Early uses of SOAP emphasized on the use of request/response patterns as a mean for conveying Remote Procedure Calls (RPC), (W3C 2003a). But the request/response pattern should not be confused with RPC. RPC uses the request/response pattern but all request/response patterns are not necessarily RPC calls (Microsoft 2003a). RPC is a programming model for using method calls that uses the request/response pattern. Initially when SOAP was about accessing objects PRC was emphasized, but today SOAP has moved towards an XML messaging framework, allowing XML messages to be exchanged instead of performing method calls allowing far more sophisticated patterns then just simple request/response patterns (Microsoft 2003a). The SOAP v. 1.2 specification support a number of Message Exchanged Patterns (MEPs). The simplest MEP is the one-way message shown in Fig 2.7 where the sender sends a message without receiving any answer. Fig 2.8 shows the request/response pattern where the sender sends a request and then waits until a response is received. Other examples of MEPs include notifications, and long running peer-to-peer conversations.

At the time of writing the latest version of SOAP is v. 1.2. SOAP v. 1.2 reached the W3C status of recommendation in June 2003 (W3C 2003a). Recommendation is the last status which basically means that the document, after going through rigorous reviews, is finished and can be used for references. Since SOAP v. 1.2 is the latest SOAP specification to achieve the recommendation status this document is based on it.

2.4.1.1 SOAP messages

Fig 2.9 illustrates how a SOAP message for a travel reservation can look like (W3C 2003a). All SOAP messages starts with the Envelope element. This makes it easy for applications to identify SOAP messages by simply looking at the name of the root element. Applications can also determine which version of SOAP is being used by looking at the namespace that follows the envelope elements. The namespace used by SOAP v. 1.2 is http://www.w3.org/2003/05/soap-envelope. It is recommended by W3C that vendors continue to support SOAP v1.1. which is identified by the http://schemas.xmlsoap.org/envelope/ namespace (W3C 2003a).

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-
envelope/role/next"
           env:mustUnderstand="true">
   <m:reference>uuid:093a2da1-q345-739r-ba5d-
pqff98fe8j7d</m:reference>
   <m:dateAndTime>2001-11-29T13:20:00.000-05:00</m:dateAndTime>
  </m:reservation>
```

```
  <n:passenger xmlns:n="http://mycompany.example.com/employees"
          env:role="http://www.w3.org/2003/05/soap-
envelope/role/next"
           env:mustUnderstand="true">
   <n:name>Åke Jógvan Øyvind</n:name>
  </n:passenger>
 </env:Header>
 <env:Body>
  <p:itinerary
    xmlns:p="http://travelcompany.example.org/reservation/travel">
   <p:departure>
     <p:departing>New York</p:departing>
     <p:arriving>Los Angeles</p:arriving>
     <p:departureDate>2001-12-14</p:departureDate>
     <p:departureTime>late afternoon</p:departureTime>
     <p:seatPreference>aisle</p:seatPreference>
   </p:departure>
   <p:return>
     <p:departing>Los Angeles</p:departing>
     <p:arriving>New York</p:arriving>
     <p:departureDate>2001-12-20</p:departureDate>
     <p:departureTime>mid-morning</p:departureTime>
     <p:seatPreference/>
   </p:return>
  </p:itinerary>
  <q:lodging
   xmlns:q="http://travelcompany.example.org/reservation/hotels">
   <q:preference>none</q:preference>
  </q:lodging>
 </env:Body>
</env:Envelope>
```

Fig 2.9 A sample SOAP message for a travel reservation.

An overview of the structure of the SOAP message shown in Fig 2.9 can be found in Fig 2.10. After the root element Envelope a SOAP message has an optional Header element and a mandatory Body element (W3C 2003b). The Header contains control information that is not application payload. This information can influence the processing of the payload. An example of the sort of information that can be stored in the header element is credential information such as a password and username (Microsoft 2003a). This information controls the access to operations and should therefore be stored in the header and not inside the payload. The Header element is the main factor for achieving extensibility in SOAP. An application that needs credentials could develop their own solutions but then interoperability would suffer. Instead it's better to develop common SOAP Headers for such a common functionality such as security. Then software vendors can build support for this extended functionality into their generic SOAP infrastructure. In Fig 2.9 there are two header blocks; reservation and passenger. The reservation element contains a reference to this particular reservation and date and time when the reservation was made. The passenger element contains the name of the passenger. The reservation header block is followed by two attributes, role and mustUnderstand. Both these attributes are discussed in a later section about the SOAP processing model.

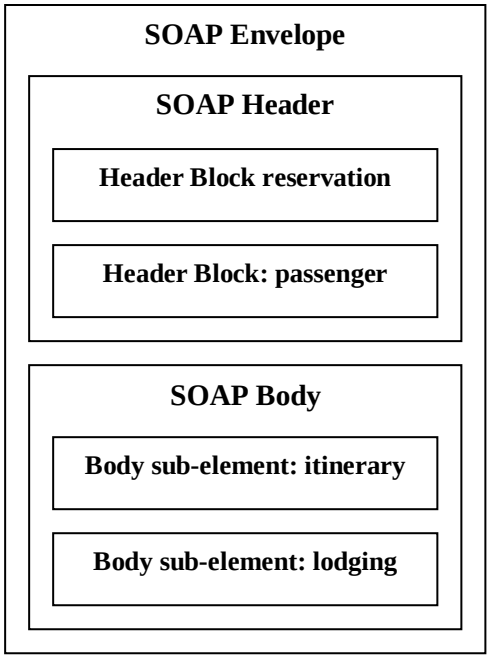


Fig 2.10 A pictorial representation of the SOAP message in Fig 2.9.

The mandatory Body element is where the actual payload, i.e. the data that is exchanged by the SOAP message, is located. In the SOAP message illustrated by Fig 2.9 the Body element contains the sub-elements itinerary and lodging. Itinerary contains the departure and arrival dates and the cities the dates are connected to while lodging contains information about reserved lodgings (in this case none).

2.4.1.2 SOAP Fault messages

The SOAP specification (M3C 2003b) defines an element named Fault for representing errors within the Body element when things go wrong. The SOAP specification defines this element to have a single distinguished element to report SOAP-specific as well as application specific faults. This is essential since without a standard error representation every application would have to develop their own, making it impossible for a generic infrastructure to distinguish between success and failure (Microsoft 2003a). Fig 2.11 gives an example of how an error message that is send in response to a payment message with a reservation and credit card information can look like.

```
<?xml version='1.0' ?>
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope"
            xmlns:rpc='http://www.w3.org/2003/05/soap-rpc'>
  <env:Body>
   <env:Fault>
     <env:Code>
       <env:Value>env:Sender</env:Value>
       <env:Subcode>
        <env:Value>rpc:BadArguments</env:Value>
       </env:Subcode>
     </env:Code>
     <env:Reason>
      <env:Text xml:lang="en-US">Processing error</env:Text>
      <env:Text xml:lang="cs">Chyba zpracování</env:Text>
     </env:Reason>
     <env:Detail>
```

```
      <e:myFaultDetails
        xmlns:e="http://travelcompany.example.org/faults">
        <e:message>Name does not match card number</e:message>
        <e:errorcode>999</e:errorcode>
      </e:myFaultDetails>
     </env:Detail>
   </env:Fault>
 </env:Body>
</env:Envelope>
```

Fig 2.11 An example of a SOAP message containing a fault code.

According to the SOAP v.1.2 specification (W3C 2003b) an error message starts with the Fault element inside the SOAP Body. The Fault element contains two or more child elements. The first child element is the Code element which is mandatory. The Code element contains a mandatory Value element with a SOAP fault code and an optional Subcode element that contains an application specific description of the fault. A list with four of the fault codes is given in Table 2.1.The next child element after Code is Reason which is also mandatory. It contains an application specific human readable description of the fault. The last three child elements after Reason are all optional. The first optional element is Node. It contains an URI to the SOAP node that generated the fault. The second optional element is Role. It contains information to identify which role the SOAP node that generated the fault was operating in when the error occurred. Both SOAP nodes and roles will be explained later in the section about the SOAP Processing model. The last child element is Detail. It is intended to carry application specific error information to better explain what happened.

Table 2.1 SOAP Fault Codes

Name	Meaning
VersionMismatch	The faulting node found an invalid namespace for the Envelope element.
MustUnderstand	An immediate child element of the SOAP header element that was not understood by the faulting node contained a SOAP mustUnderstand attribute with a value of “true”.
Sender	The message was incorrectly formed or did not contain the appropriate information in order to succeed. For example, the message could lack the proper authentication or payment information. It is generally an indication that the message is not to be resent without change.
Receiver	The message could not be processed for reasons attributable to the processing of the message rather than to the contents of the message itself. For example, processing could include communicating with an upstream SOAP node, which did not respond. The message could succeed if resent at a later point in time.

2.4.1.3 SOAP Processing model

SOAP defines a process model which describes the actions taken by a SOAP node when receiving a SOAP message (Microsoft 2003a). Fig 2.7 illustrated the simplest SOAP scenario with one application (SOAP sender) sending a SOAP message to another application (SOAP receiver). The SOAP processing model allows for more advanced architectures like the one in Fig 2.12.

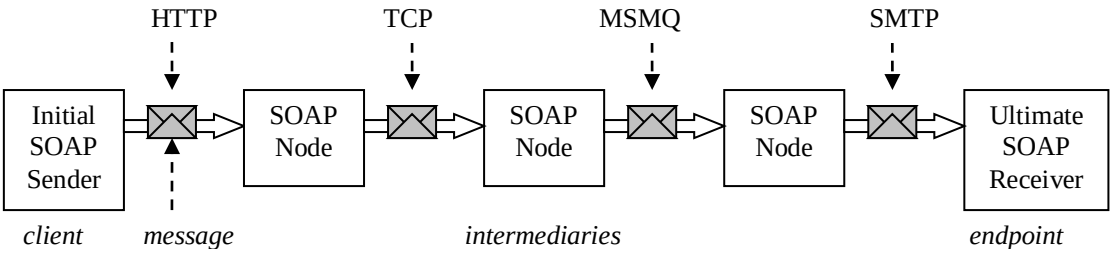


Fig 2.12 Sophisticated SOAP messaging.

Fig 2.12 contains several intermediary nodes between the SOAP sender and the SOAP receiver. In this document the term SOAP node refers to any application that processes SOAP messages, whether it's the sender, an intermediary, or the ultimate receiver of the message.

A SOAP intermediary acts as both a sender and a receiver at the same time and intercepts SOAP messages (W3C 2003b). The intermediary processes SOAP headers targeted at it and forwards SOAP messages towards the ultimate SOAP receiver. According to Microsoft (Microsoft 2003a), intermediary nodes make it possible to design flexible networking architectures that can be influenced by message content. A good example of what intermediary nodes can be used for is routing.

When a SOAP node processes a message it's said to act in one or more SOAP roles according to the SOAP v.1.2 specification (W3C 2003b). Each role is identified by a unique URI.

Table 2.2 SOAP Roles.

SOAP Role name	**Description**
http://www.w3.org/2003/05/soap-envelope/role/next	Each SOAP intermediary and the ultimate SOAP receiver MUST act in this role and MAY additionally assume zero or more other SOAP roles.
http://www.w3.org/2003/05/soap-envelope/role/ultimateReceiver	To establish itself as an ultimate SOAP receiver, a SOAP node MUST act in this role. SOAP intermediaries MUST NOT act in this role.

Table 2.2 shows two of the roles defined in SOAP v1.2, a complete list can be found in the SOAP v1.2 specification (W3C 2003b). When a SOAP node receives a message for processing it has to determine which role it should assume. This is done by inspecting the SOAP message. When defining a header block using the Header element, the attribute role can be used to target a specific role. An example of how to use the attribute role is shown in Fig 2.13. In Fig 2.13 the Header block reservation targets a SOAP node with the role next. Since all nodes must act as "next", (see Table 2.1), the SOAP Header will be processed by the first node (after the SOAP sender) the message encounters. The ultimate receiver of the message is acting in the ultimateReceiver role and processes both headers as well as the actual content of the message inside the body element.

```
<env:Envelope xmlns:env="http://www.w3.org/2003/05/soap-envelope">
 <env:Header>
  <m:reservation
xmlns:m="http://travelcompany.example.org/reservation"
          env:role="http://www.w3.org/2003/05/soap-
envelope/role/next"
           env:mustUnderstand="true">
```

...

Fig 2.13 Example of the role attribute.

When a SOAP node has determined its role it has to process all mandatory Headers and can also choose to process optional Headers. A mandatory Header element is indicated by the mustUnderstand attribute with the value "true". If the attribute is missing or has the value "false" it is optional and may or may not be processed. Processing an optional header is up to the application (W3C 2003b). If the SOAP node can't process the Header correctly it has to stop immediately and return a fault message indicating what went wrong. After a SOAP node has successfully processed a SOAP message it's required to remove the Header block. The SOAP node can then reinsert the Header with the same or a changed value or insert a completely new Header if it wants to. This is up to the application and is not regulated by the SOAP specification (W3C 2003b).

2.4.1.4 SOAP Summary

SOAP is a protocol for communication in a distributed environment by exchanging XML-based messages. SOAP is very simple since it doesn't provide common features such as security and routing functionality. Instead it provides an extensible framework for adding these features as extensions. SOAP does this by specifying the Header element which vendors can use to add their own functionality. In the future vendors will probably define standard SOAP headers that everyone agrees on, especially for such common needs such as security (Microsoft 2003a). SOAP achieves a high level of interoperability since it is based on XML which has been adopted by most of the software industry. Fig 2.5 showed the relationship between the Web Service components and Fig 2.14 shows how SOAP fits into this relationship (Borland 2002).

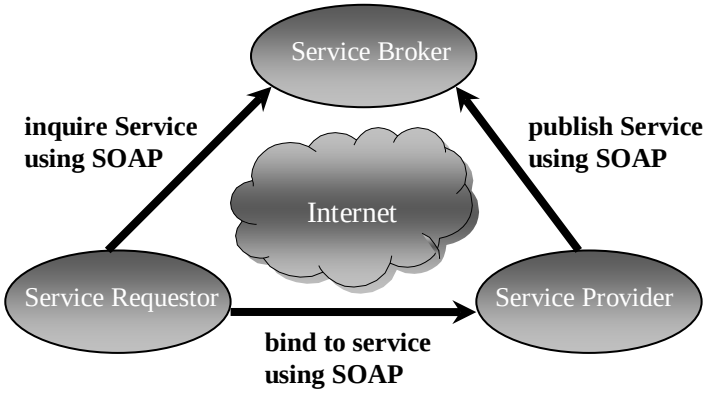


Fig 2.14 How SOAP fits into the Web Service picture.

In Web Services, SOAP is used to transport all messages, whether it is method calls between a client and the service or if a client is trying to discover the service by sending a message to the service broker.

2.4.2 WSDL

Before a client can contact a Web Service the client has to know how to communicate with it. To be able to communicate the following information has to be available for the client (Muschamp P. 2004):

- Information about all available functions and their respective calling parameters.

- Information about the data types for all XML messages, including the value specifications.
- Information about binding to the specific protocol to be used.
- Information on the address used for locating the specified service.

When undertaking an integration of a client with a Web Service the software developer could extract this information from the documentation for the Web Service (Muschamp P. 2004). Extracting all this information for a complex Web Service would be very time-consuming and prone to human errors. Ideally the integration should be handled automatically instead. WSDL has been developed for this purpose.

WSDL stands for Web Services Description Language. A WSDL file is an XML document that describes a Web Service by providing the information listed above (Muschamp P. 2004). The WSDL file therefore acts as a contract between a client and the Web Service, stating what the Web Service provides and how the client can communicate with it.

Since WSDL is a machine-readable language (since it is just another XML file), tools and infrastructure can easily be build around it (Microsoft 2003b). When creating a new client (or a new Web Service for that matter) that is going to communicate with a certain Web Service the WSDL file associated with the specific Web Service is the only thing needed in order to communicate effectively. When the WSDL file has been found a tool can take the file as input and generate the necessary code for the client to use when communicating with the Web Service.

It's not just enough to know which messages a Web Service accepts (and what the calling parameters are). The client also has to know about the possible message exchange patterns that is supported by the Web Service (e.g. if you send an **Add** message you get an **AddResponse** message back). When talking about WSDL such a message pattern is referred to as an operation (Microsoft 2003b). Operations are very important for a client since they define which messages the Web Service accepts and which messages it sends in return (see Fig 2.15).

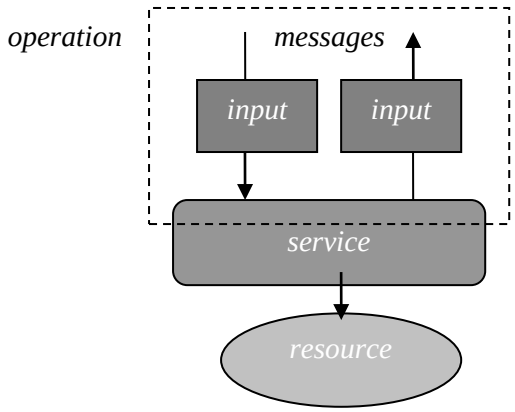


Fig 2.15 Messages and operations.

Just as in the world of object-orientation, a group of related operations grouped together is called an interface in WSDL (Microsoft 2003b). This is especially important for developers working with object-oriented environments to be aware of since the XML interfaces can map to programmatic interfaces (or abstract classes) in their own language of choice and therefore have an impact on how the developers write their code.

A Web Service consumer must also know which communication protocol (e.g. HTTP, TCP, SMTP) they should use for sending messages to the service along with information about the

specific mechanics that the given protocol use such as the commands, headers, and error codes. In WSDL a binding specifies the concrete details about what goes on the wire by outlining how to use an interface with a specific binding. An interface can have multiple bindings but each binding should be accessible at a unique address identified by a URI, in WSDL referred to as a Web Service endpoint (see Fig 2.16).

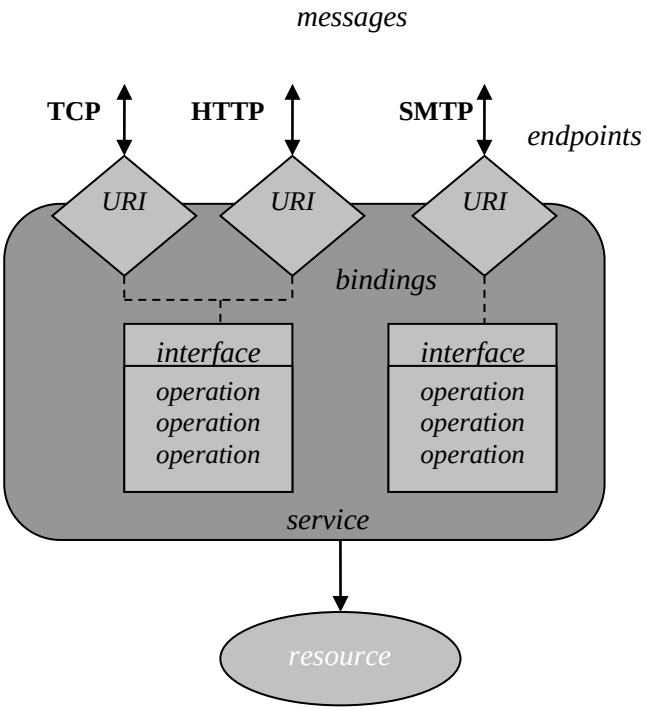


Fig 2.16 Interfaces and bindings.

A WSDL (version 2.0) file contains four types of elements (W3C 2006):

- The *types* element describes the kinds of messages that the service is allowed to send and receive.
- The *interface* element describes what functionality the service provides.
- The *binding* element describes how to access the service.
- The *service* element describes where to access the service.

To better explain how a WSDL document is structured an example will be used based on an example from W3C (W3C 2006). The example involves a hotel that wants to offer travel agencies with a reservation system to reserve rooms directly over internet instead of reserving rooms through the phone as they have done before. This reservation functionality should be offered by a Web Service. To keep things simple the example Web Service will only implement CheckAvailability functionality. When using this functionality a client has to specify a check-in date, a check-out date and the room type. The service will then return the room rate (a floating point number in USD$) if a room is available, a zero rate if no room is available or an error if any of the input data are invalid.

The first thing to do when creating a WSDL 2.0 file to describe a Web Service is to define a target namespace (W3C 2006). A target namespace must be an absolute URI and should be dereferenceable to a WSDL 2.0 document. For example, the hotel should make the WSDL 2.0 document available at this URI (and if the WSDL 2.0 document is split into several documents then the target namespace should refer to a master document that includes all the WSDL 2.0 documents needed for that service description).

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    . . . >

  . . .
</description>
```

Fig 2.17 An initial empty WSDL 2.0 document.

Fig 2.17 shows an initial empty WSDL 2.0 document. The root element for WSDL 2.0 documents are the Description element (W3C 2006). It basically just acts as a container for all other elements. The document then references the namespace used by WSDL 2.0 documents called http://www.w3.org/2006/01/wsdl. Then the targetNamespace is defined to be http://greath.example.com/2004/wsdl/resSvc. This WSDL document should be placed on this URI as described previously. Last a prefix, tns, is defined to be the same URI as the targeNamespace, allowing the prefix to be used in the rest of the document instead of the whole URI adress.

2.4.2.1 Types

The first element do describe a Web Service is the Types element. Under this element all message types used by the Web Service is defined, including error messages. Fig 2.18 shows how the WSDL document will look like when the Types element has been added. A new namespace, http://greath.example.com/2004/schemas/resSvc, has now been referenced and associated with the attribute ghns. This namespace is used to reference the XML Schema target namespace for the message types that is defined under the Types element. Under the Types element the targetNamespace is set to the XML Schema target namespace that was created for this reservation service. The message types that are defined under the Types element will therefore be associated with this namespace. The first type to define is the checkAvailability message type. It consists of the checkInDate and checkOutDate elements with the type date and the roomType element with the type string. These three elements are used when the Web Service receives a checkAvailability message as described in the previous section when the Web Service's functionality was described. The second message type is the checkAvailabilityResponse which has the type double since it should return a float number with the price of the hotel room in USD$. The last message type is the error message that should be returned if any of the input data were invalid. It simply has the type string to describe the fault.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    . . . >

  ...
```

```
  <types>
    <xs:schema
        xmlns:xs="http://www.w3.org/2001/XMLSchema"

targetNamespace="http://greath.example.com/2004/schemas/resSvc"
        xmlns="http://greath.example.com/2004/schemas/resSvc">

      <xs:element name="checkAvailability"
type="tCheckAvailability"/>
      <xs:complexType name="tCheckAvailability">
        <xs:sequence>
          <xs:element  name="checkInDate" type="xs:date"/>
          <xs:element  name="checkOutDate" type="xs:date"/>
          <xs:element  name="roomType" type="xs:string"/>
        </xs:sequence>
      </xs:complexType>

      <xs:element name="checkAvailabilityResponse"
type="xs:double"/>

      <xs:element name="invalidDataError" type="xs:string"/>

    </xs:schema>
  </types>
  . . .
</description>
```

Fig 2.18 Hotel WS Types definition.

2.4.2.2 Interface

The second element to describe a Web Service is the Interface element. This element defines the abstract interface of the Web Service as a set of abstract operations (W3C 2006). Each operation represents a simple interaction between the client and the service and specifies the types of messages the service can send or receive as part of that operation. An operation also specifies a Message Exchange Pattern (MEP) that indicates the sequence in which the associated messages are to be transmitted between the parties. An example of a MEP is the *in-out* pattern which indicates that if a client send a message in to the service, the service will either send a reply message back out to the client or it will send a fault message back indicating an error.

The hotel service uses a single operation (checkAvailability) which has to be defined in the interface. The operation uses the checkAvailability and checkAvailabilityResponse message types already defined in the Types section and the in-out pattern since the operation should use a simple request-response interaction (W3C 2006). In addition to the normal input and output messages a fault message also needs to be specified to be used by the operation if something goes wrong. The interface part of the hotel Web Service is shown in Fig 2.19.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    . . .
    xmlns:wsdlx="http://www.w3.org/2006/01/wsdl-extensions">

  . . .
```

```
  <types>
        ...
  </types>

  <interface  name = "reservationInterface" >

    <fault name = "invalidDataFault"
           element = "ghns:invalidDataError"/>

    <operation name="opCheckAvailability"
           pattern="http://www.w3.org/2006/01/wsdl/in-out"
           style="http://www.w3.org/2006/01/wsdl/style/iri"
           wsdlx:safe = "true">
        <input messageLabel="In"
             element="ghns:checkAvailability" />
        <output messageLabel="Out"
             element="ghns:checkAvailabilityResponse" />
        <outfault ref="tns:invalidDataFault" messageLabel="Out"/>
    </operation>

  </interface>

  . . .
</description>
```

Fig 2.19 Hotel WS Interface definition.

The interface definition starts with giving the interface the unique name "*reservationInterface*" by using the interface name attribute.

Then a name is defined for a fault so that when operations are later defined they can refer to the fault by name.

The third part defines the only operation used by this Web Service. The operation is given the name "*opCheckAvailability*" so the operation can be referenced in the next section when defining bindings. The definition specifies through the *pattern* attribute that the operation will use the *in-out* pattern previously discussed. The *wsdl:safe* attribute last in the operation definition indicates that the operation will not obligate the client in any way (like the client agreeing to buy something).

The fourth element in the interface definition is the Input element which specifies an input message. The *in-out* MEP has already been defined but it represents a template for a message sequence and could therefore in theory consist of multiple input and/or output messages. Therefore we must also indicate which potential input message in the pattern this particular input message represents. Since we use the *in-out* pattern this is trivial since there is only one input message.

The fifth part specifies the output message similar to the input message already described.

The last element is the *outfault* element which associates a fault with this operation. Since an operation can involve a sequence of several messages a fault can potentially occur at various points in the message sequence. Therefore the *messageLabel* attribute is used to indicate the desired point for this particular fault.

2.4.2.3 Binding

So far only *what* abstract messages the hotel Web Service can exchange has been specified, not *how* those messages can be exchanged (W3C 2006). This is the purpose of a binding. For every interface, a binding specifies the concrete message format and transmission protocol details used. This information has to be specified for each operation and fault in the interface. Generally binding details for each operation and fault inside an interface are specified using the Operation and Fault elements respectively (W3C 2006). The hotel Web Service will use SOAP v 1.2 as message format and HTTP as the underlying transmission protocol, which is shown is Fig 2.20.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
        ...
  </interface>

  <binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
          type="http://www.w3.org/2006/01/wsdl/soap"

wsoap:protocol="http://www.w3.org/2003/05/soap/bindings/HTTP">

    <operation ref="tns:opCheckAvailability"
      wsoap:mep="http://www.w3.org/2003/05/soap/mep/soap-
response"/>

    <fault ref="tns:invalidDataFault"
      wsoap:code="soap:Sender"/>

  </binding>

  . . .
</description>
```

Fig 2.20 Hotel WS Binding definition.

A binding starts with the element Binding. Its first attribute is the *name* attribute for defining a name for this particular binding. The second attribute *interface* specifies which previously defined interface we are specifying a binding for. The third attribute *type* specifies which kind of concrete message format to use, in this case SOAP v.1.2. The last attribute specifies the underlying transmission protocol that should be used, in this case HTTP.

The second element, Operation, references the previously (in the interface section) defined "opCheckAvailability" operation in order to specify the binding details for it. The attribute *wsoap:mep* specifies the SOAP MEP that will be used to implement the abstract WSDL MEP (*in-out*) that was previously specified. When HTTP is used as the underlying transmission protocol this attribute controls whether HTTP GET or POST should be used as

the underlying HTTP method. In this case the use of "http://www.w3.org/2003/05/soap/mep/soap-response "causes HTTP GET to be used by default.

The last element used when defining a binding is the Fault element. Its first attribute references the fault to define a binding for (in this case the invalidDataFault previously defined in the *interface* section) and its second attribute specifies the SOAP v.1.2 fault code that will cause this fault message to be sent.

2.4.2.4 Service

The binding specified how messages will be transmitted but we also need to specify where the service can be accessed by using the Service element (W3C 2006). In a WSDL 2.0 document a service specifies a single interface that the service supports, and a list of endpoint locations where the service can be accessed. An endpoint references a previously described binding to indicate which protocols and transmission formats that should be used at this particular endpoint.

```
<?xml version="1.0" encoding="utf-8" ?>
<description
    xmlns="http://www.w3.org/2006/01/wsdl"
    targetNamespace= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:tns= "http://greath.example.com/2004/wsdl/resSvc"
    xmlns:ghns = "http://greath.example.com/2004/schemas/resSvc"
    xmlns:wsoap= "http://www.w3.org/2006/01/wsdl/soap"
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope">
  . . .

  <types>
    . . .
  </types>

  <interface  name = "reservationInterface" >
    . . .
  </interface>

  <binding name="reservationSOAPBinding"
          interface="tns:reservationInterface"
        . . . >
    . . .
  </binding>

  <service name="reservationService"
       interface="tns:reservationInterface">

     <endpoint name="reservationEndpoint"
```

```
                binding="tns:reservationSOAPBinding"
                address
="http://greath.example.com/2004/reservation"/>

  </service>

</description>
```

Fig 2.21 Hotel WS Service definition.

A service specification starts with the Service element with the first attribute *name* to give the service a name, and the second attribute *interface* to specify which interface the service endpoints will support. As Fig 2.16 illustrates, each interface can have several bindings and each binding must be accessed by an endpoint, so an interface can therefore have several endpoints, each supporting its own set of protocols and transmission formats. A client can therefore choose which protocols and transmission format they want to use by simply choosing the right endpoint, supporting them.

The Endpoint element gives the endpoint a name, specifies which previously described binding it should use, and gives the endpoint a unique address at which this service can be accessed using the specified binding.

2.4.2.5 WSDL Summary

WSDL is an XML document for describing a Web Service by providing all information necessary about the service so a client can communicate with it. WSDL therefore acts as a contract between a client and the Web Service, stating what the Web Service provides and how the client can communicate with it.

To get a good overview of a WSDL 2.0 document Fig 2.22 illustrates the different elements and attributes involved in such as document (W3C 2006). Fig 2.22 illustrates the document structure using the XML Infoset which is a set of definitions for use in other specifications that need to refer to the information in an XML document (W3C 2004b).

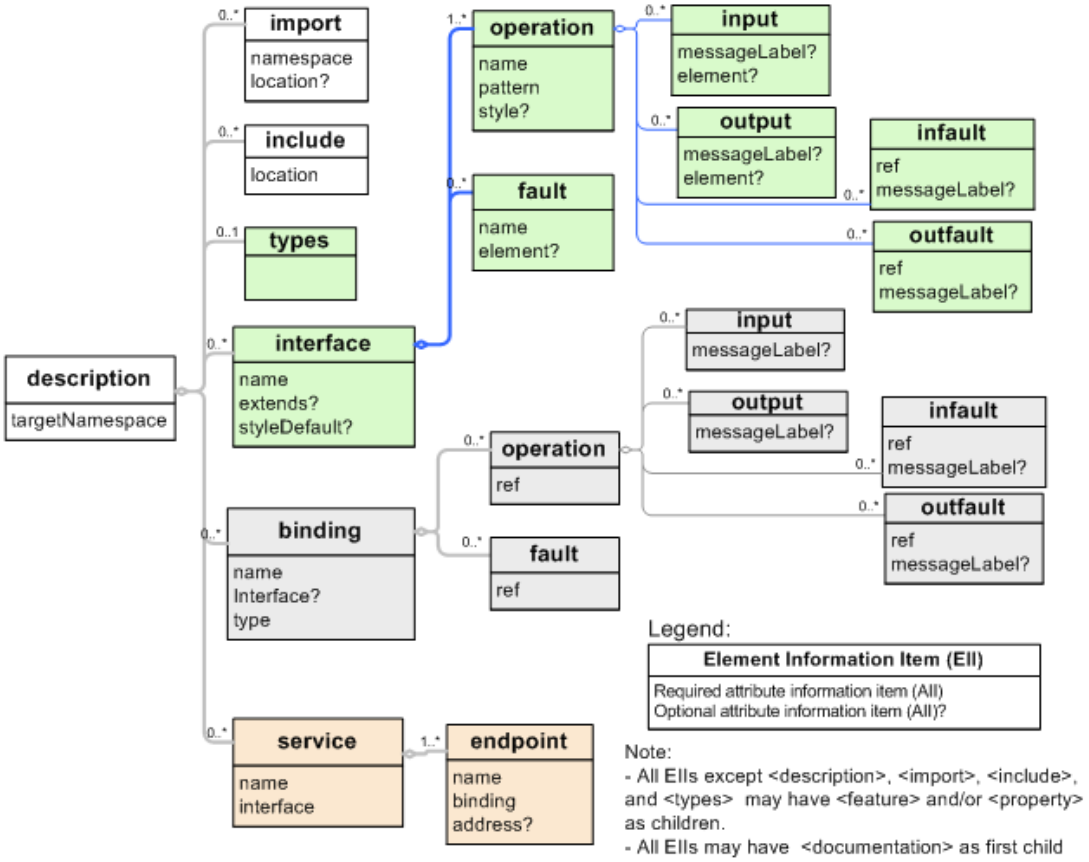


Fig 2.22 WSDL 2.0 Infoset Diagram.

Fig 2.23 illustrates how WSDL fits into the relationship of Web Service components originally illustrated in Fig 2.5 (Borland 2002).

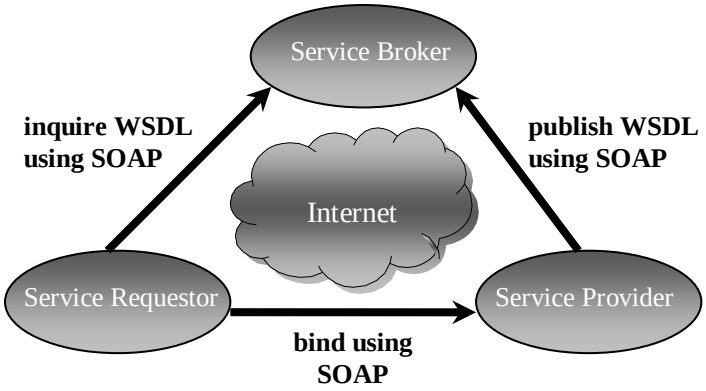


Fig 2.23 How WSDL fits into the Web Service picture.

2.4.3 UDDI

When a Web Service has been created potential clients must have a way to locate a suitable Web Service (Muschamp, P. 2004). If the location of the WSDL file of a desired Web Service is known it's just a matter of pointing the development software to the file and to implement the integration (Muschamp, P. 2004). This approach assumes the location of the WSDL file is known, the Web Service provides the desired functionality, and that the party behind the Web Service is known and trusted.

A better approach to locate a suitable Web Service than discovering the WSDL file manually would be to have a directory where suitable Web Services can be discovered automatically. This is the idea behind UDDI which stands for Universal Description, Discovery and Integration (Szypeski et al. 2002).

UDDI was originally proposed by Ariba, IBM and Microsoft but have found over 600 members by early 2006 (UDDI 2006). The UDDI specification is administered by the Organization for the Advancement of Structured Information Services (OASIS) which is a not-for-profit, international consortium that drives the development, convergence, and adoption of e-business standards according to (UDDI 2006) and (Muschamp, P. 2004).

UDDI enables companies to (Muschamp, P. 2004):

- describe its business and its services,
- discover companies that offer these services,
- and to integrate with these services.

Say for example that a client under development relies on a credit checking function in order to validate a company's customers (Muschamp, P. 2004). A UDDI registry can be used to find this function. When creating a request for a credit checking function other requirements such as cost limits, security needs, performance criteria's etc. can be added to find the most suitable function available. The registry will then return one or more companies that provide such a function (Muschamp, P. 2004).

An important part of the UDDI specification is of course the actual registry that stores references to Web Services. This registry is called UDDI Business Registry (UBR) and provides a central place for businesses to publish their services and discover what services other businesses offer (UDDI 2004a). The public UBR registry is in many ways similar to the root node of the DNS database which is another successful example of a distributed

registry infrastructure. Although the UBR is an important piece of the UDDI standard, it represents only one aspect of the overall effort (UDDI 2004a). Just as the majority of DNS activity occurs within a company's own network, so too do most UDDI implementations support a business' own Web Service infrastructure (UDDI 2004a).

UBR is currently operated by four companies (UDDI 2002). A list of these companies with the location of their respectively UDDI registers can be found in Table 2.3 below.

An important note is that UDDI is in itself a set of Web Services since every UDDI enquiry is an XML message wrapped in a SOAP envelope (UDDI 2004a) and (Szypeski et al. 2002).

Table 2.3 Public UDDI registries.

Company	UDDI Location
Microsoft	http://uddi.microsoft.com/
IBM	http://uddi.ibm.com/
SAP	http://uddi.sap.com/
NTT Com	www.ntt.com/uddi/

Many source such as (Borland 2002) compares the UBR with a telephone book, since the information stored in the registry can be divided into three categories; the "white pages", "yellow pages" and "green pages". White pages contain information about a business, yellow pages categorize businesses by standard taxonomies such as by a particular industry or a product category and green pages contain the technical information about offered services.

2.4.3.1 UDDI Data structures

The UDDI specification doesn't just specify the UBR registry. It also specifies which information should be stored about a Web Service (UDDI 2004b). A UDDI registry (whether it is a public one like UBR or a private one behind a firewall) offers a standard mechanism to classify, catalogue and manage Web services, so that they can be discovered and consumed (UDDI 2004b).

The top-level data structure within UDDI is the businessEntity structure (UDDI 2004b). The businessEntity structure represents businesses and providers within UDDI and the Web Services they offer. The structure contains information such as names and descriptions in multiple languages, contact information and classification information. Although the name of the structure implies that businesses are represented by the businessEntity structure, any parent service provider such as a department, an application or even a server can be represented (UDDI 2004b). Service descriptions and technical information are expressed by referencing businessService and bindingTemplate structures explained below.

The second data structure is the businessService structure. The businessService structure represents a logical grouping of Web Services (UDDI 2004b).The structure doesn't contain any technical information; instead the structure allows the ability to assemble a set of services under a common rubric (UDDI 2004b). For example, a businessService structure could contain a set of Purchase Order Web services (submission, confirmation and notification) that are provided by a business. Each businessService structure is the logical child of a single businessEntity structure.

The third data structure is the bindingTemplate. Each bindingTemplate structure represents an individual Web Service and contains, in contrast to the two previous structure, technical

information needed by applications to bind and interact with the Web Service being described (UDDI 2004b). The bindingTemplate must either contain the access point for a given service or an indirection mechanism that will lead to the access point. Each bindingTemplate is the child of a single businessService.

The fourth and last data structure specified by UDDI is the Technical Model called tModel for short (UDDI 2004b). Each distinct specification, transport, protocol, or namespace is represented by a tModel. An example of a tModel is WSDL and other documents that specify and outline the contract and behaviour that a Web Service may wish to comply with. To describe Web Services that conforms to a particular set of specifications references to the tModels representing these concepts are placed in the bindingTemplate (UDDI 2004b). The UDDI registry itself doesn't store these technical documents; instead the tModel stored in the registry contains the addresses to where these documents can be found.

Fig 2.24 illustrates how the four data structures fit together (UDDI 2004b).

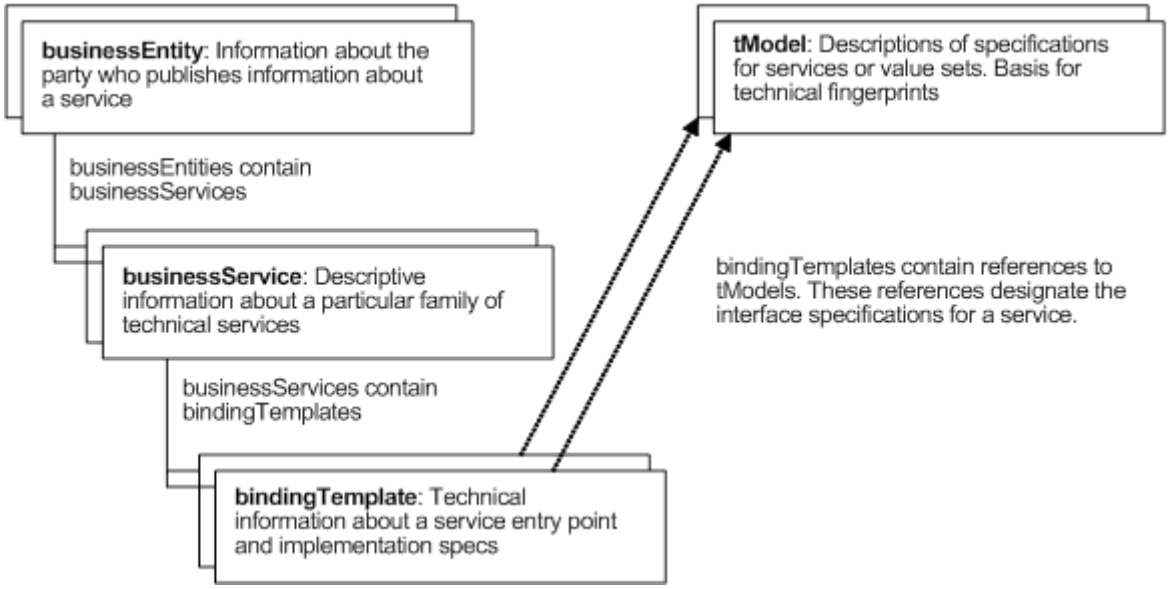


Fig 2.24 UDDI core data structures.

2.4.3.2 UDDI APIs

To allow businesses to publish their Web Service and make inquiries for services in the UBR, UDDI offers two sets of Application Programmer Interfaces (APIs) called the Inquiry API and the Publishing API (UDDI 2004b).

The Inquiry API allows clients to locate and obtain details on entries in a UDDI registry. The Inquiry API supports three distinct patterns for making inquiries; the browse pattern, the invocation pattern and the drill-down pattern (UDDI 2004b).

The browse pattern allows businesses to perform a search for registry entries based on a set of search criteria's (UDDI 2004b).

The invocation pattern allows businesses to only retrieve information on entries that are updated (UDDI 2004b). This pattern assumes that a business caches the entries that are interesting for that business.

The drill-down pattern allows a business to retrieve the complete information belonging to an entry with a given key (UDDI 2004b).

The publishing API provides functions such as four save_xx and four delete_xx functions, one for each of the four UDDI data structures (UDDI 2004b). Using the save_xx functions a business can publish information about a new Web Service or alter already published information. The delete_xx functions can be used to completely remove data structures from the registry.

2.4.3.3 UDDI Summary

The UDDI specification specifies a registry where companies can publish their Web Services and also to find Web Services provided by other companies. UDDI consists of:

- a UBR which is a public UDDI registry administrated by Microsoft, IBM, SAP and NTT Com,
- a set of data structures used for storing information about Web Services,
- and a set of APIs for publishing new Web Services and make inquiries about existing ones.

Fig 2.25 illustrates how UDDI fits into the relationship of Web Service components originally illustrated in Fig 2.5 (Borland 2002). Fig 2.25 now displays the complete picture of all the major parts involved in the Web Service architecture.

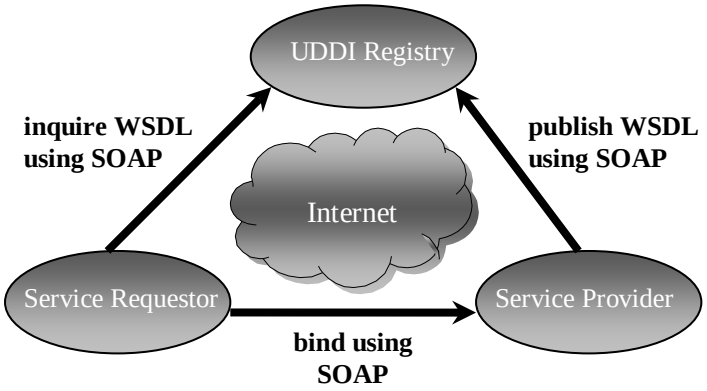


Fig 2.25 How UDDI fits into the complete Web Service picture.

2.5 Other techniques for distributed communication

Now that Web Services has been investigated its time to briefly look at some of the currently most used techniques for achieving distributed communication. Web Services isn't a completely original idea but more like an evolution of older and already existing techniques. It's therefore important to know the basics of the older techniques in order to better understand why Web Services looks like they do. There are many techniques in use today,

but since this chapter is just going to give a brief introduction to some of the main techniques used to achieve distributed computer communication, three of the most used techniques will be described. The first technique is CORBA which will be covered in section 2.4.1. The second technique is Java RMI which will be covered in section 2.4.2 and the last one is Microsoft's DCOM technique which will be covered in section 2.4.3.

2.5.1 CORBA

The Common Object Request Broker Architecture, CORBA, is a set of standards for enabling distributed object-oriented systems implemented in different programming languages and running on different platforms to interact according to Szypeski et al. (Szypeski et al. 2002). The CORBA specifications is defined and managed by the Object Management Group, OMG, which is a non-profit organization aiming to achieve interoperability on all levels of an open market for objects.

A CORBA application consists of objects which are individual units of running software that combine some functionality and data, (OMG 2006a), similar to a normal object in the world of object-orientation (Sommerville 2001). According to Siegel, (Siegel 2000), "CORBA only connects objects, not applications". However a CORBA object must have an interface that defines the operations and public attributes that are available from the object. These interfaces are defined using a standard, language-independent interface definition language called the OMG IDL (Sommerivlle 2001). The first thing to do when creating a CORBA application is to create the IDL interface (OMG 2006a). The IDL interface is then compiled into a client stub and an object skeleton. The stub and skeleton acts as proxies for the client requesting a service and the object providing the service respectively (OMG 2006a). In order to enable development of the client and object using a number of different programming languages there are bindings available to OMG IDL from several languages such as C, C++, Java, COBOL , Ada, Lisp an so on (OMG 2006a). Once the IDL Interface has been created it can be compiled using an IDL compiler to the specific programming language used (Szyperski et al.2002). Because IDL Interfaces are defined so strictly and both the stub and skeleton is generated from the same IDL interface the client stub has no problem to mesh perfectly with the object skeleton (OMG 2006a).

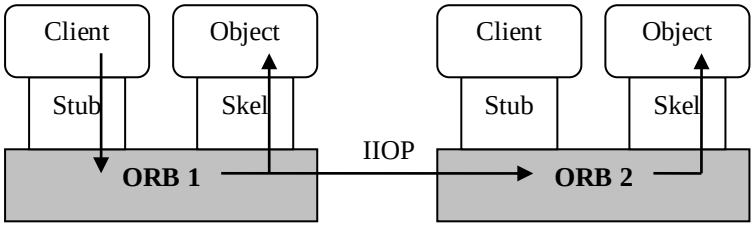


Fig 2.26 A request passing from client to object implementation.

In the heart of CORBA is the Object Request Broker, ORB. The ORB takes care of the communication between the objects in an application. It's also responsible for keeping track of all IDL interfaces so that the clients can use them and to provide different services such as transaction, security, naming etc. Since the client is shielded from the ORB (and from the object providing the requested service) by the IDL interface the client doesn't have to know anything about the location of the target object (Szyperski 2002). Fig 2.26 above illustrates how a communication between a client to either a local object (running in the same ORB) or a remote object is handled. When a client wants to communicate with a local object it calls the generated stub, believing that the stub is the actual object. The stub marshals (writes and transmits) the arguments and sends the request to the ORB which in turn sends the request to the destined object skeleton. The skeleton unmarshals (reads) the arguments and sends them to the correct operation in the object. The object skeleton then marshals the result it receives from the object and sends it back to the client in the same manner as previously described

(OMG 2006a). Since both the client stub and the object skeleton is generated from the same IDL interface in order to communicate, the client needs a stub for every object is wants to communicate with (OMG 2006a). When buying an ORB, these IDL compilers (one for each programming language) are also delivered since they are specific for the ORB. Therefore a stub that has been generated by an IDL compiler from vendor A can't be used on an ORB from vendors B and vice versa (Siegel 2000).When the client requests the service from a remote object (as can be seen in Fig 2.27), the client ORB has to call the ORB the object is located on. In order to communicate with each other both ORB's need to use the same protocol. This protocol has been defined to be the Internet Inter-ORB Protocol IIOP (OMG 2006a). Other protocols exist but the ORB must at least support this protocol for interoperability reasons and it's also required by OMG for compliance.

The previously described way to communicate is very static since the client stub and object skeleton have to be generated manually by a compiler before the communication can take place. Often the operation to be invoked needs to be selected at runtime instead and is simply not known beforehand (Szyperski 2002). CORBA therefore provides something called the Dynamic Invocation Interface (DII) and the Dynamic Skeleton Interface (DSI) to provide a mechanism for making dynamic invocations.

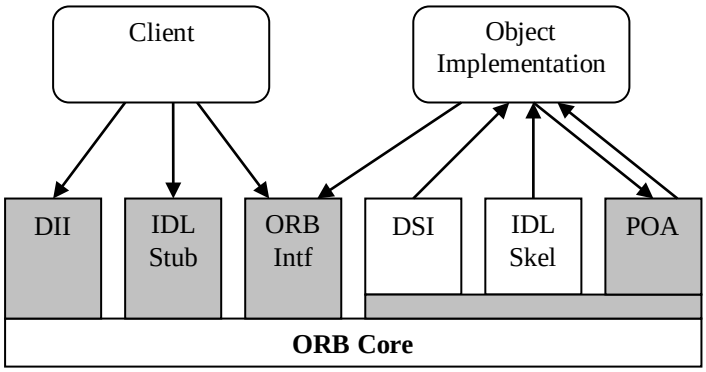


Fig 2.27 A more detailed picture of CORBA.

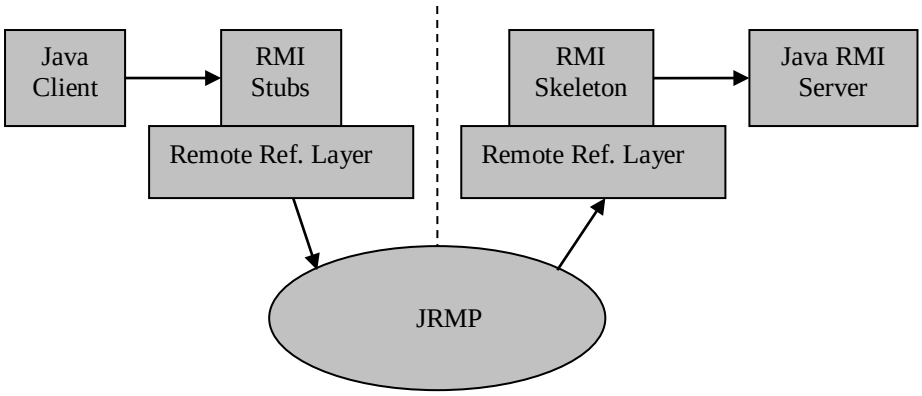
Fig 2.27 illustrates CORBA in some more detail with the addition of the DII and DSI interfaces. When a client has discovered an object its wants to communicate with (for example by using a CORBA service such as the Trader Service provided by the ORB) it contacts the DII which will in turn call the ORB. Since the marshalling is done by the ORB instead of interface-specific compiled code, only one DII is needed which serves all objects (OMG 2006b).
When an object has been requested by a client, the server ORB (the ORB the object is located on) has to create and start the object (if it isn't already running). The server ORB therefore use something called a Portable Object Adapter (POA) to keep track of the objects and to create and destroy the object when needed (OMG 2006b). The last part of Fig 2.27 is the ORB interface which is used by both the client and the object to accesses different services provided by the ORB such as directory services, transaction services and persistence services and so on (Sommerville 2001).

2.5.2 Java RMI

Java Remote Method Invocation (Java RMI) developed by Sun Microsystems, is a technique to enable distributed Java object-based application development using the Java environment (Nagappan et. al. 2003). A distributed Java application is defined to consist of objects running on different Java Virtual Machines (JVMs). Two objects running on different JVMs but both JVMs run on the same computer are therefore not considered to be a distributed system. One of the purposes with RMI is to make objects in separate JVMs look and act like local objects (Allamaraju et. al. 2001). A very important part of the RMI design is therefore

transparency since an application shouldn't have to see any difference between a local object and a remote one. RMI uses serialization, which is a "lightweight object persistence technique that allows the conversion of objects into streams", to send objects from a client to a server (Nagappan et. al. 2003).

Similar to the previously described CORBA technique, Java RMI also uses manually generated stubs and skeletons to communicate. The stub and skeleton are both generated by a tool called rmic which is a part of the Java Development Kit (JDK) (Nagappan et. al. 2003).



2.28 Java RMI architectural model.

The first part of a Java RMI application, (shown above in Fig 2.28), is the actual client. The client can be a Java applet or a stand-alone application that performs method invocations of a server object (Nagappan et. al. 2003). The client can pass primitive data types or serializable objects as arguments to the server object.

The client communicates with the generated RMI stub which acts as a proxy, encapsulating the network information of the server and delegating the client invocations to the server (Nagappan et. al. 2003). The stub also marshals the method arguments and unmarshals the return values.

The communication between the stub and skeleton goes through the RMI infrastructure which consists of two layers (Nagappan et. al. 2003). The first layer, called the remote reference layer, separates the specific remote reference behaviour from the client stub by handling certain reference semantics like connection retries. The second layer is the transport layer which provides the actual networking infrastructure. The RMI infrastructure is based on the Java Remote Method Invocation (JRMP) protocol on top of TCP/IP (Allamaraju et. al. 2001).

A later version of RMI called RMI-IIOP uses the Internet Inter-ORB Protocol developed by OMG instead of JRMP (Allamaraju et. al. 2001). By using the IIOP protocol instead, RMI can integrate with CORBA objects.

The skeleton receives the invocation requests from the stub and processes the arguments (unmarshalling) and delegates them to the RMI server (Nagappan et. al. 2003). After the request has been successfully executed the skeleton marshals the return values and passes them back to the stub through the RMI infrastructure.

The RMI server is the remote object that implements the exposed interfaces and executes the client requests (Nagappan et. al. 2003). The server processes the arguments that comes from the skeleton and returns the result.

To enable clients to get hold of references to the server objects they want to communicate with RMI provides a registry-oriented mechanism that provides a simple non-persistent

naming lookup service that is used to store the remote object references in and to enable lookups from client applications (Nagappan et. al. 2003). This registry has to be set up on a host with a port number that the clients know about (Allamaraju et. al. 2001).

2.5.3 Microsoft DCOM

In today's operating systems, processes are shielded from each other and therefore can't communicate directly for security reasons (DalmatianGroup 2001). Microsoft's Component Object Model (COM) has therefore been developed to enable interprocess communication by intercepting the calls made from clients and forwarding them to the target component in another process. Fig 2.29 below illustrates how the COM/DCOM run-time library provides the link between client and component (DalmatianGroup 2001).

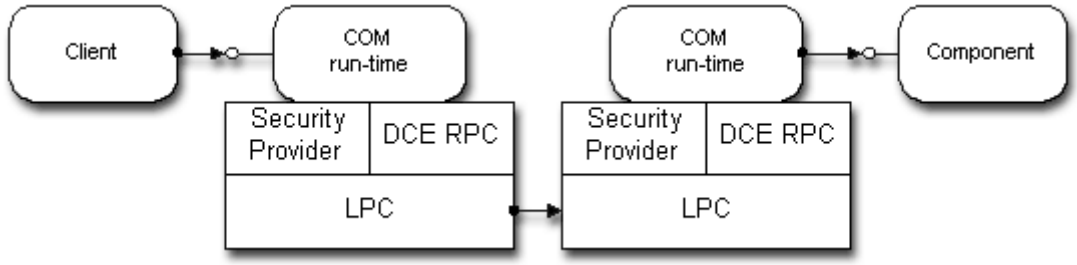


Fig 2.29 COM components in different processes.

The Distributed Component Object Model (DCOM) is basically a protocol developed by Microsoft to enable COM components to communicate directly over a network in a reliable, secure and efficient manner (DalmatianGroup 2001). When the client and component resides in the same machine, COM is used to enable interprocess communication. When the client and component resides in different machines DCOM simply replaces the local interprocess communication with a network protocol (DalmatianGroup 2001). Neither the client nor the component will notice any difference, just that the response time might be a bit longer then usual. Fig 2.30 below illustrates the overall DCOM architecture.

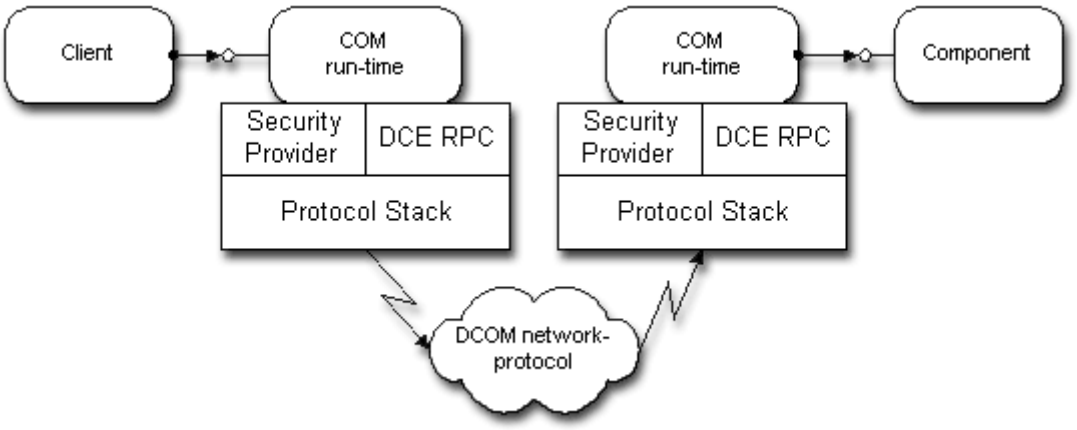


Fig 2.30 The overall DCOM architecture.

The COM run-time provides object-oriented services to clients and uses RPC and the security provider to generate standard network packets that conform to the DCOM wire-protocol standard (DalmatianGroup 2001).

As based on COM, DCOM is completely language-independent (DalmatianGroup 2001). COM components can be created using virtually any language, and those components can be used by even more languages. A DCOM developer can therefore choose the language and development tool that they are most familiar with.

3 EXPERIMENT

3.1 Introduction

The idea behind this experiment is to find out how well Microsoft's .NET and Sun Microsystems's J2EE platforms are working compared to each other when handling Web Services. This chapter describes the hardware and software environments used when running the experiment, the application that was developed and the measurements that were taken in order to evaluate how 'well' the two platforms handles Web Services. The following three chapters will handle the results from the experiment, the conclusions made, and some potential future work that could be done to continue evaluating Web Services, respectively.

3.2 Environment

The hardware environment used by the experiment consisted of three computers. Two of the computers were used to run the clients and therefore had to have the same configuration. The computer used to run the server needed more resources and therefore a faster computer with more memory was selected. All three computers were connected by a LAN in a closed network so that no external communication (like the Internet) could disturb the experiment. The specifications for the hardware used are listed Table 3.1.

Table 3.1 Hardware environment used in the experiment.

Hardware – Client Computers	**Description**
Computer Brand	Dell
RAM Memory	256 MB
Processor	Intel Celeron 700 MHz
Hardware – Server Computer	
Computer Brand	HP
RAM Memory	1 GB
Processor	AMD Athlon 64 3000+
Network	
LAN	10 Mbit

The software used in the experiment is listed in Table 3.2 followed by explanations of the choice of products. The software was used using their standard configuration during the experiment. No tweaking or adjustments were done since an adjustment could have a positive effect on one of the platforms and a negative one on the other. Since the computers were completely cleaned before the software were installed, and only the necessary software were installed, the impact unnecessary software would have on the computers performance during the experiments was minimized.

Table 3.2 Software environment used by the experiment.

Software	Description
Microsoft Windows XP Professional	Operating system used on all computers.
Microsoft Visual Studio .NET 2005	Development environment for the .NET-based application.
Microsoft .NET Framework v. 2.0 Redistributable Package	Contains everything needed to run .NET-based applications.
C#	The programming language used when developing the .NET-based application.
Internet Information Services 5,1 (IIS)	The Web Server used for hosting the .NET-based Web Service.
netBeans IDE 5.0	Development environment for the J2EE-based application.
Java	The programming language used when developing the J2EE-based application.
Sun Java System Application Server Platform Edition 8.2	The Web Server used for hosting the J2EE-based Web Service.

Windows XP Professional has been chosen as the operating system used simply because it is the newest, most used and most available operating system that support the .NET framework available today. Since both platforms have to run on similar conditions so that the tests are valid, Windows XP Pro was the best choice. The Pro version of Win XP was chosen because it contains the **IIS** Web Server that was used to host the .NET-based Web Service.

Visual Studio .NET 2005 was chosen since it's the newest development environment that support the C# language, it supports the .NET v.2.0 Framework and it's fully compatible with the **IIS** Web Server, meaning that a developed Web Service can be automatically deployed on the Web Server by the software tool.

Instead of installing Visual Studio .NET on all computers used in the experiment, **.NET Framework v2.0 Redistributable Package** was installed. The framework contains everything needed for compiling and running .NET applications. The framework is a part of Visual Studio .NET and is therefore automatically installed when Visual Studio is installed.

The **C#** language was chosen as the programming language for the .NET application since it's designed specifically for the .NET platform and therefore was an obvious choice.

The Web Server **Internet Information Services 5.1** (IIS) was chosen to host the .NET-based Web Service since it's already integrated into Win XP Pro and the chosen .NET development environment can easily deploy Web Services on **IIS**.

As development environment for the J2EE application, **netBeans IDE 5.0** has been chosen. NetBeans is a free development environment and can be downloaded for free bundled together with the **Sun Java System Application Server Platform Edition 8.2** that is capable of hosting J2EE-based Web Services from (NetBeans 2006). Initially, a version of Borland JBuilder was chosen but since it was difficult to get hold of it, an alternative environment had to be used instead.

Java was chosen as the programming language simply because there were no alternatives, since J2EE is based on Java.

3.3 Application developed

Since two platforms should be evaluated, two applications have been developed, one based on the .NET platform using the C# programming language, and another application based on the J2EE platform using the Java programming language. Both applications consist of two parts; one client and one server part. When running the experiment two clients are running simultaneous on different computers, communicating with the server.

Since more than one client is used, and the time for processing the client requests are measured, there is a problem with starting both clients at the same time to get accurate measurements. For this reason a simple application starter had to be developed. The application starter is a simple application developed using C# which just creates a socket connection to the clients it should initiate. When the clients receive a connection they start processing so there is no need to exchange any data over the connection. When creating an application starter that should be able to start both C# and Java based clients there were some thoughts about the interoperability problem that always arouses when applications developed using different programming languages should communicate. When testing different ideas practically it was discovered that a simple socket connection was sufficient since a C# based application starter could connect to a Java-based client and vice versa without any interoperability problems. There were no particular reasons behind choosing C# as the implementation language for the application starter since it's judged that the application starter won't have any significant impact on the end result anyway.

The clients developed reads a number of strings from a file and sends this data in chunks (the size varies from scenario to scenario and the exact number can be found in the next chapter where each scenario is described in detail) to the Web Service. The clients then records the time it takes from calling the Web Service until the result has been received. The actual Web Service that was developed for this experiment is in fact very simple. It simply takes an array of strings as inputs, sorts this array using the BubbleSort algorithm (some scenarios sorts the data other does not), and returns the sorted array to the client that requested the service.

The last thing developed was a simple program that generated a file of strings. This file is used as input to the clients, since they need a lot of unsorted strings to send to the Web Service. The program created a file containing strings with a length between 2 and 10 characters. Each string consists of the characters 'a' to 'z' randomly selected. A 'normal' text file could of course be used instead but since it takes some extra effort processing a text file and to divide the text into strings that can be sorted, and this extra effort doesn't really contribute to the end result in any significant way, the file just consists of a number of randomly generated strings instead.

When the Web Service and the clients had been developed they have to know how to communicate with each other. There are several ways to do this but the simplest one used in this experiment is to generate stubs based on the WSDL-file which is automatically created when the Web Service is created. For the interested, the WSDL-file generated by the .NET-

based Web Service is shown in Appendix A. The stub acts as a proxy for the Web Service which means that the client will communicate with the Web Service as with any 'local' object and has no idea that the Web Service is located on another machine. The client is in fact communicating with the stub which is a local object and takes care of all details involved in remote communication.

In .NET, the stub was manually generated by a tool called WSDL.exe which is included in the .NET Framework v2.0 Redistributable Package. When running the tool a new file with the same name as the Web Service is generated.

In J2EE the stub was generated by the development environment used, in this case netBeans IDE 5.0. In .NET only the stub was generated, but in J2EE some other classes, such as different types of exceptions that could occur and a data-type called ArrayOfString representing the array of strings that is sent between the client and the Web Service, is also generated.

When a .NET-based client should communicate with a J2EE-based Web Service it was just a matter of generating the stub based on the Web Service's WSDL-file. But when a J2EE-based client should communicate with a .NET-based Web Service the client had to be slightly altered because the array of strings sent between the client and the server had to be wrapped in the data type called ArrayOfString generated by the development environment, as previously mentioned. In summary the integration of J2EE- and .NET-based applications using Web Services were very easy since it was basically just a matter of getting hold of the WSDL-file. This fact has to some degree to do with the data types used. If more complex than just the basic data types had been used it could be much harder to integrated the platforms.

A last note about the developed applications is that every precaution has been taken in order to make them as comparable as possible. One issue that arose when running one of the scenarios was that a J2EE-based client couldn't read from a file and store in an array more then around 1 million strings compared to a .NET-based client which could easily read all 4 million strings from the file and save them in an array. The J2EE application was redeveloped by simply just reading 1 million strings and reusing this data four times. This is comparable to have an array with 4 million strings but to keep the applications as similar and comparable as possible the .NET-based clients were also updated accordingly.

3.4 Experiment execution

Before each scenario was executed the proper Web Service was deployed. When using a .NET-based Web Service no work had to be done since web server was integrated into Windows XP and, as long as the Web Service had been deployed once it was developed, would start the Web Service automatically when a request came for it. In J2EE on the other hand the server had to be started.

When the Web Service was running the clients were started using the application launcher. The application launcher was located on the server so that both clients would get the start-signal at the same time. The clients start by reading all the strings from a file and storing them in an array. Since file-management can take a different amount of time depending on the platform used, and since the purpose of this experiment is to measure the platforms performance when handling Web Services and not when managing files, this work is done before the actual experiment (the studied part of it) is started. When the clients are finished with reading the files they wait until they receive a socket-connection from the application starter before they start sending data to the Web Service. Fig. 3.1 below gives an overview of the distributed components involved in a typical experiment, (some experiment scenarios only contain 1 client).

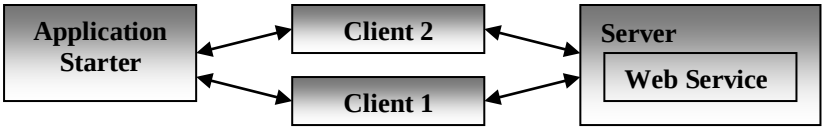


Fig 3.1 Experiment overview.

After every call the client makes to the Web Service, the elapsed time is recorded in an array. The values in the array are stored in a file when the client has completed all its calls in order to minimize the disturbances file-management can have on the measured performance. To make the measured values more reliable and to exclude strange results each scenario is executed twice. A scenario, in the context of the experiment, means a test with a certain configuration such as a J2EE Web Service with 2 J2EE-based clients sending a certain amount of data for a certain number of times. Each scenario configuration along with the results is described in the next chapter about the experiment results.

3.5 Measurements

Measured in the experiment is the time it takes for a client to send an amount of data to a Web Service and to receive the result sorted. Each client stores all recorded times, (one time is recorded for every call the client makes to the Web Service), in a file which is summarised to the values which can seen in the example in Table 3.3.

Table 3.3 Example of the values calculated from the measurements done by a client.

Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
734774	5358	120	183,6935	176799240	102047897

All measurements recorded by the client have been copied into a spreadsheet in Excel. The **Total**, **Max**, **Min, and Average** values have then been calculated automatically by Excel. All these values for every experiment scenario can be found in the Result section (next chapter).
When some of the experiment scenarios had been finished, it was discovered that J2EE stored the values with an accuracy of 1ms as planned but .NET stored the values with a much higher accuracy (one millisecond with four decimals). All values recorded by the .NET-based client have therefore been rounded off to the nearest millisecond to be comparable to the measurements taken by the J2EE-based client.

The **Sent Bytes** and **Received Bytes** values are taken from the network monitor tool which is a part of Windows XP. Before every run the network monitor was reset to cut out network traffic which occurred before the experiment is started. The purpose of these values is to see how much network traffic each type of client (J2EE- or .NET-based ones) creates.

Since most scenarios used two clients which connects to a Web Service an application starter has been used as previously described. In order to make the measurement recorded by the client even more comparable one client is called first during the first run of the scenario and the other client is called first during the second run.

4 RESULT

4.1 Introduction

This chapter contains the result from the experiment. Each Scenario with its particular configuration is described followed by the values calculated from the results. Each section is then ended with a brief explanation of the result.

The next chapter containing the conclusion of the experiment gives are more general overview of the numbers and what the general conclusions obtained from the results are.

4.2 Scenario result

4.2.1 Scenario 1

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.1 and the second run in Table 4.2 below.

Table 4.1 Result from Scenario 1 Run 1

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	1603556	971	230	400,88895	102149238	101404259
Client 2	1605699	1011	230	401,4247204	102152597	101404081

Table 4.2 Result from Scenario 1 Run 2

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	1602855	1482	230	400,713698	102146776	101396433
Client 2	1600692	951	230	400,1729204	102146685	101395931

Since this is the first Scenario the numbers give a general view of .NETs performance when a .NET-based Web Service sorting the data is used. Another Web Service will of course give a different set of values as can be seen in Scenario 11 which is similar to this Scenario with the only difference that the Web Service doesn't sort anything.

The client's values in this Scenario are pretty close to each other considering clients in the same run as well as clients in different runs. This means the values are stable and therefore can be considered to be trustworthy.

4.2.2 Scenario 2

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.3 and the second run in Table 4.4 below.

Table 4.3 Result from Scenario 2 Run 1

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	734774	5358	120	183,6935	176799240	102047897
Client 2	723849	5358	120	183,6935	176800373	102049821

Table 4.4 Result from Scenario 2 Run 2

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	705672	3645	120	176,418	176803282	102050035
Client 2	748728	3565	130	187,182	176802451	102050691

This if the first Scenario based on the J2EE-platform which means the numbers only give a general overview of J2EEs performance.

The numbers aren't as close to each other as in the previous Scenario, but they are still fairly close and therefore should also be considered as relatively reliable.

By comparing these number to the ones found in the previous Scenario it can be concluded that J2EE is more than twice as fast as .NET under the described circumstances of which a Web Service sorting its input data is most important. Another interesting observation is that the J2EE-based clients send a lot more data than they receive as opposite to the .NET-based clients in the previous Scenario which send approximately the same amount of data as they receive. This should be a drawback of J2EE (since more data should take a longer time to transmit as well as increasing the burden on the network which usually handles other traffic too) but this can't be concluded since J2EE is so much faster than .NET. The reason behind this phenomenon is unclear but one possibility is that the array of strings sent by the client has to be wrapped in a class called ArrayOfString as described in section 3.3 about the application developed. This could make the data that have to be sent in J2EE larger than in .NET but why only the data sent by a client is larger and not the data received is somewhat of a mystery.

4.2.3 Scenario 3

In this Scenario one J2EE-based client (Client 1) and one .NET-based client (Client 2) have issued requests to a J2EE-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.5 and the second run in Table 4.6 below.

Table 4.5 Result from Scenario 3 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	635094	4166	120	158,7735	176805575	102049414
Client 2	1323753	2794	200	330,938366	214082243	102792246

Table 4.6 Result from Scenario 3 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	642636	3646	120	160,659	176802507	102046270
Client 2	1319848	871	190	329,961962	214083602	102785697

This is the first Scenario in which clients based on both platforms are communicating with the same Web Service. Here it can be concluded that a J2EE-based client connecting to a J2EE-based Web Service is around twice as fast as a .NET-based client connecting to a J2EE-based Web Service. This isn't really a surprise since a J2EE-based client should know how to communicate with a J2EE-based Web Service more effectively than a .NET-based client. Another observation made when looking at this Scenario's amount of data sent and received is that the J2EE-based clients send and receive the same amount as in the previous Scenarios but the .NET-based now sends a lot more data than before. This indicates that the amount of data sent by a client has more to do with the Web Service (or the web server the Web Service is running on) than on the client sending the data. Since this is a .NET-based client communicating with a J2EE-based Web Service there ought to be some performance loss when integrating different platforms. This could explain why a .NET-based client sends a bit more data than a J2EE-based one when the Web Service is J2EE-based. The last observation is that both the J2EE and .NET-based clients are faster compared to Scenario 1 and 2. In J2EEs case the reason could simply be that .NET is much slower which means the J2EE-based client is free to use the server more. In the case of the .NET-based client this could depend on the J2EE server being faster than its .NET counterpart.

4.2.4 Scenario 4

In this Scenario one J2EE-based client (Client 1) and one .NET-based client (Client 2) have issued requests to a .NET-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.7 and the second run in Table 4.8 below.

Table 4.7 Result from Scenario 4 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	1612851	4346	270	403,21275	135500663	101766713
Client 2	1602744	981	230	400,6861584	102147101	101396951

Table 4.8 Result from Scenario 4 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	1625177	5268	260	406,29425	135501258	101768310
Client 2	1614872	4326	240	403,718018	102147685	101399808

This is the last Scenario containing a mix of .NET- and J2EE-based clients. The most obvious observation here is that both clients are as slow as the .NET-based client in Scenario 1. The reason that the J2EE client is as slow as the .NET-based one indicates that the web server used to host .NET-based Web Services is working as fast as it can and therefore can't handle the requests any faster. Secondly the amount of data sent by a .NET-based client is the same amount as before in Scenario 1. But the amount sent by the J2EE-based client is lower than before. This indicates that the web server used to host .NET-based Web Services doesn't require as much data as its J2EE counterpart. The amount of data received by the clients are the same as in all previous Scenarios which indicates that both web servers sends the same amount of data back to the client.

4.2.5 Scenario 5

In this Scenario a single J2EE-based client has issued requests to a J2EE-based Web Service. The client has called the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.9 and the second run in Table 4.10 below.

Table 4.9 Result from Scenario 5 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	683574	3565	120	170,8935	176803999	102041565

Table 4.10 Result from Scenario 5 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	687451	3625	130	171,86275	176799207	102037475

The purpose of this Scenario was to see how fast a single J2EE-based client could complete its task. As can be expected the result shows that a single client performs better than 2 simultaneous clients as in Scenario 2. But strangely the J2EE-based client in Scenario 3, where the client has to compete with a .NET-based client, performs even better. Logically a single client should perform best and not a client competing with other clients (whether they are based on the same platform or not). Therefore either this value or the value in Scenario 3 is not completely reliable and more tests to see if the numbers are correct or not and especially to find out the reasons behind the numbers should be done.

4.2.6 Scenario 6

In this Scenario a single .NET-based client has issued requests to a .NET-based Web Service. The client has called the Web Service 4000 times, attaching an array of 1000 strings to be sorted in each request.

The result from the first run is shown in Table 4.11 and the second run in Table 4.12 below.

Table 4.11 Result from Scenario 6 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	1003583	671	220	250,8957704	102144010	101358000

Table 4.12 Result from Scenario 6 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	1003603	901	220	250,9007776	102141025	101359312

The purpose of this Scenario was to see how a single .NET-based client would perform. The values show that a single client is much faster than two .NET-based clients (as in Scenario 1). What's interesting is that this difference is much bigger than the difference between a single J2EE-based client (Scenario 5) and two J2EE-based clients (Scenario 2). In the first two Scenarios, J2EE was much faster than .NET but now with only one client this difference is much smaller. Therefore a conclusion can be that the difference in performance between the clients seems to decrease as the number of clients decrease (although more tests with more simultaneous clients should be done to verify this observation).

4.2.7 Scenario 7

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 2000 times, attaching an array of 2000 strings to be sorted in each request.

The result from the first run is shown in Table 4.13 and the second run in Table 4.14 below.

Table 4.13 Result from Scenario 7 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	3271664	2263	1011	1635,832211	57452243	56901207
Client 2	3270833	2243	1041	1635,416614	57552528	57002345

Table 4.14 Result from Scenario 7 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	3284883	2423	1021	1642,441715	56979044	56251416
Client 2	3285715	3064	1031	1642,857313	56746361	56202577

This Scenario is intended to see how an increased amount of data sent per call affects the performance. The amount of data has been doubled (2000 strings instead of 1000) and the result is that an average call takes 4 times longer compared to Scenario 1. Worth noticing is that most of the increase in time is because of the sorting algorithm since doubling the amount of data will result in much more than just a doubling in time when the data has to be sorted.

4.2.8 Scenario 8

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 2000 times, attaching an array of 2000 strings to be sorted in each request.

The result from the first run is shown in Table 4.15 and the second run in Table 4.16 below.

Table 4.15 Result from Scenario 8 Run 1

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	1166368	5498	320	583,184	175384188	100808647
Client 2	1002943	5418	310	501,4715	175381837	100805829

Table 4.16 Result from Scenario 8 Run 2

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	1098260	5278	320	549,13	175382648	100803906
Client 2	1064740	5287	310	532,37	175381791	100803727

Just as in the previous Scenario the purpose with this Scenario is to observe how J2EE handles an increased amount of data per call. The result is that an average call takes around 3 times longer than before (Scenario 2). Comparing this with the result from the previous Scenario reveals that the difference in performance between the platforms has increased since .NET increased 4 times and J2EE only 3 times. Now J2EE is 3 times as fast as .NET (around 550ms for J2EE compared to 1640ms for .NET) compared to being 2 times as fast when comparing Scenario 1 and 2. It can therefore be concluded that increasing the amount of data sent per call will increase the difference in performance between J2EE and .NET (in J2EEs favour).

4.2.9 Scenario 9

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 8000 times, attaching an array of 500 strings to be sorted in each request.

The result from the first run is shown in Table 4.17 and the second run in Table 4.18 below.

Table 4.17 Result from Scenario 9 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	807982	601	70	100,9977276	104933140	104247020
Client 2	807040	651	70	100,8800584	104933433	104247948

Table 4.18 Result from Scenario 9 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	810145	1242	70	101,2681164	104931525	104245275
Client 2	809214	1172	70	101,151699	104931808	104245703

As opposite to the previous two Scenarios, which investigated how the platforms handled an increased amount of data per call, this Scenario observes what happens when the amount of data per call is decreased instead. The result is that .NET is around 4 times faster compared to Scenario 1 where the amount of data sent per call was twice as much (1000 compared to 500 in this Scenario).

4.2.10 Scenario 10

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 8000 times, attaching an array of 500 strings to be sorted in each request.

The result from the first run is shown in Table 4.19 and the second run in Table 4.20 below.

Table 4.19 Result from Scenario 10 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	700614	4206	50	87,57675	179402541	104640316
Client 2	684805	4226	50	85,600625	179402559	104623188

Table 4.20 Result from Scenario 10 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	688080	4026	60	86,01	179401698	104640771
Client 2	673909	4146	50	84,238625	179403346	104635353

The purpose of this Scenario is to see how J2EE handles a decreased amount of data per call. The result is that J2EE is around 2 times faster compared to Scenario 2.

The conclusion drawn from decreasing the amount of data sent per call is that the difference between the platforms decreases. The average time for J2EE is around 86ms compared to 101ms for .NET. This should be compared to J2EE being more than twice as fast when comparing Scenario 1 with 2 and being 3 times as fast when comparing Scenario 8 with 9. This is just as expected since the difference increased when the amount of data sent per call increased.

4.2.11 Scenario 11

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings in each request. No sorting is used which means the Web Service just returns the same array as it receives as input from the clients.

The result from the first run is shown in Table 4.21 and the second run in Table 4.22 below.

Table 4.21 Result from Scenario 11 Run 1

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	351135	1452	50	87,7837268	102133768	101198162
Client 2	348932	2413	50	87,2329348	102137831	101192950

Table 4.22 Result from Scenario 11 Run 2

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	352937	671	50	88,2343748	102139429	101201425
Client 2	343033	681	50	85,7583144	102138368	101206469

The purpose of this Scenario is to see what happens when the sorting algorithm is removed from the Web Service. Since there was a big difference between the platforms (in J2EEs favour) when they were sorting the data, will this difference still remain without any sorting? The result is that .NET is more than 4 times as fast as in Scenario 1 when the Web Service sorted the data.

4.2.12 Scenario 12

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 4000 times, attaching an array of 1000 strings in each request. No sorting is used which means the Web Service just returns the same array as it receives as input from the clients.

The result from the first run is shown in Table 4.23 and the second run in Table 4.24 below.

Table 4.23 Result from Scenario 12 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	548639	4396	90	137,15975	176801081	102053440
Client 2	566856	4387	90	141,714	176804832	102055142

Table 4.24 Result from Scenario 12 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	584859	3455	90	146,21475	176804037	102080650
Client 2	511056	3475	90	127,764	176804595	102080810

The purpose of this Scenario is to see how the J2EE platform handles a Web Service without any sorting and comparing the result to the previous Scenario. The result from this Scenario is that J2EE is a bit faster than in Scenario 2 (138ms compared to 183ms). Comparing this result to .NETs result in Scenario 11 shows that .NET is now faster than J2EE (86ms compared to 138ms). The conclusion is that J2EE handled the actual sorting of the data better than .NET but .NET handles raw transmission of data faster (under the current circumstances that is).

4.2.13 Scenario 13

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 2000 times, attaching an array of 2000 strings in each request. No sorting is used which means the Web Service just returns the same array as it receives as input from the clients.

The result from the first run is shown in Table 4.25 and the second run in Table 4.26 below.

Table 4.25 Result from Scenario 13 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	289246	1382	100	144,6229576	100785400	99771176
Client 2	363953	851	100	181,9766696	100772252	99775665

Table 4.26 Result from Scenario 13 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	242108	1302	100	121,0540672	100768058	99769041
Client 2	373657	1092	100	186,8286464	100764438	99770273

The purpose of this Scenario is to build upon the result obtained in the previous Scenario by increasing the amount of data sent in each call to 2000 strings. The result shows that the average call is around twice as long as in Scenario 11. The values in the result vary a bit more than usual and should therefore not be considered completely trustworthy. A little scepticism here can be a good thing.

4.2.14 Scenario 14

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 2000 times, attaching an array of 2000 strings in each request. No sorting is used which means the Web Service just returns the same array as it receives as input from the clients.

The result from the first run is shown in Table 4.27 and the second run in Table 4.28 below.

Table 4.27 Result from Scenario 14 Run 1

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	625992	4887	180	312,996	175377247	100749414
Client 2	631379	4857	180	315,6895	175377629	100744101

Table 4.28 Result from Scenario 14 Run 2

	Total (ms)	**Max(ms)**	**Min(ms)**	**Average(ms)**	**Sent Bytes**	**Received Bytes**
Client 1	541720	4527	180	270,86	175378613	100753929
Client 2	695598	4527	180	347,799	175380844	100762255

This Scenario checks how J2EE handles an increased amount of data per call. The result is similar to the previous Scenario since an average calls now takes around twice as long time as in Scenario 12. The conclusion here is that .NET is still around twice as fast as .NET (average 311ms for Scenario 14 compared to 158ms in Scenario 13) when the amount of data sent per call is doubled. Since there is no sorting going on continuing to increase the amount of data per call should give similar results.

4.2.15 Scenario 15

In this Scenario two J2EE-based clients have issued requests to a J2EE-based Web Service. Each client calls the Web Service 16 000 times, attaching an array of 100 strings to be sorted in each request.

The result from the first run is shown in Table 4.29 and the second run in Table 4.30 below.

Table 4.29 Result from Scenario 15 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	319818	3495	10	19,988625	81100762	50131232
Client 2	315585	3565	10	19,7240625	81101666	50131232

Table 4.30 Result from Scenario 15 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	315114	2474	10	19,694625	81101160	50131672
Client 2	313311	2403	10	19,5819375	81101310	50131672

The purpose of this Scenario is to see how J2EE handles a decreased amount of data to be sorted per call. The amount of data in each call has been decreased as much as possible (but still high enough to be measurable) to see if the difference between the platforms decreases or maybe even vanish completely. The result from this test should therefore be compared to .NETs result which can be found in the next Scenario.

4.2.16 Scenario 16

In this Scenario two .NET-based clients have issued requests to a .NET-based Web Service. Each client calls the Web Service 16 000 times, attaching an array of 100 strings to be sorted in each request.

The result from the first run is shown in Table 4.31 and the second run in Table 4.32 below.

Table 4.31 Result from Scenario 16 Run 1

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	207939	13412	10	12,9961876	50286289	51213791
Client 2	209531	2984	10	13,0957057	50286627	51214077

Table 4.32 Result from Scenario 16 Run 2

	Total (ms)	Max(ms)	Min(ms)	Average(ms)	Sent Bytes	Received Bytes
Client 1	207108	381	10	12,9442379	50288131	51215620
Client 2	206797	351	10	12,924835	50288154	51215834

The purpose of this last Scenario is to see how .NET handles a small amount of data per call and to compare this with the result from the previous Scenario to see which platform performs ‘best’. The result is that .NET takes around 13ms per call in average while J2EE takes just under 20ms. This means that .NET is still faster although .NET is not twice as fast as before.

5 CONCLUSION

This chapter will summaries and conclude the results obtained from the experiment.

The original questions that formed the basis of the experiment were:

- What is the performance of Web Services in J2EE compared to .NET when the clients are a mix of J2EE and .NET based ones?

If it is possible a bonus questions to answer would be:

- Which platform, .NET or J2EE, should be selected to host Web Services when the clients are a mix of J2EE- and .NET-based ones and only performance is considered?

After carefully going through all obtained results, described and analyzed in the previous chapter, it is obvious that the bonus question can't be answered. Answering such a question requires results that clearly point in a certain direction, either towards .NET or J2EE. When taking a decision to invest in a particular platform the environment (type of Web Service, number of clients, client types, network capacity, amount of data sent per call etc.) has to be carefully analysed and compared with the platform characteristics of which some has been found and analyzed in this thesis.

The general conclusion obtained when performing the experiment is that there are a lot of aspects that should be taken into account when evaluating the platforms and that one of the two evaluated platforms are not obviously better than the other when only looking at the performance aspect. The evaluation of the platforms was done in an experiment where aspects such as generated amount of network traffic and performance were analysed and evaluated. Of these aspects only performance was determined beforehand. The others were determined after some initial experimentation, were some interesting observations was made that had to be further analyzed in order to find the cause of it. The experiments were limited both in number (because of the time aspect), in software (other server software to host Web Services as well as other client-side software could have been used), in computers (more computers could have been used to generate higher network traffic, more requests / s from the clients to the Web Service, etc) and so on. Therefore all interesting observations couldn't be evaluated completely and more work has to be done here. Additionally the platforms can also be evaluated in other ways than through experiments. Theoretical evaluations is also important were aspects such as Java being platform-independent compared to C# and its limitation to the Microsoft Windows platform could also be done to give a more complete picture when comparing the platforms. This is something that could/should be addressed by future works in the area.

The first conclusion made when performing the experiment was that J2EE is much faster than .NET when the Web Service sorted its input data. When 1000 strings were sent per call J2EE was around twice as fast as .NET and this difference increased to J2EE being 3 times as fast as .NET when the number of strings per call increased to 2000. When decreasing the number of strings per call the difference decreases as would be expected. These values are illustrated in Fig. 5.1 below. In Scenario 1&2 1000 strings were sent per call, in Scenario 7&8 2000 strings per call were sent and in Scenario 9&10 500 strings were sent. A client in a 'real-world' application would probably send much less data per call than what is observed here which means the difference in performance would be minimal. An example of a 'real-world' application is order systems were clients can order items from a Web Service. A call would contain the items ordered as well as personal information about the person ordering.

This amount of information is much less than the data sent in the experiment. The experiment was intended to load the Web Service as much as possible to see how much it could handle, and since it was hard to increase the number of computers used, the amount of data per call was increased instead. A better way to do this in future experiments is to have much better time measuring than 1 ms. (which was the limit in the experiment) to be able to measure smaller time intervals. This would enable decreasing the amount of data per call and still be able to measure it relatively accurate (which is one of the reasons behind the large amount of data sent per call in the experiment).

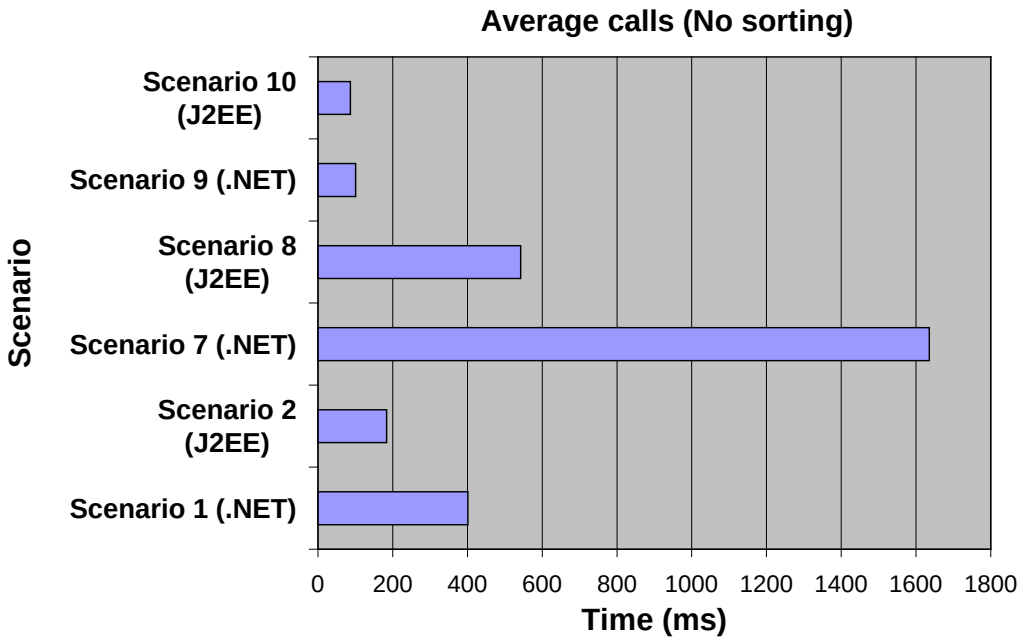


Fig 5.1 Average calls for some Scenarios.

The second conclusion was that J2EE clients send more data than its .NET counterpart (often around 70% more). The reasons behind this extra amount of data have not been found, but one of the reasons could be the wrapping of data to the data-type ArrayOfStrings done in J2EE. The only time a .NET client generated more data than a J2EE client was in Scenario 3 where a J2EE-based Web Service was used. Since this was the only time a .NET client communicated with a J2EE-based Web Service (and therefore had to wrap the data to an ArrayOfStrings) this is a strong indication that the wrapping highly affects the amount of data sent. The only conclusion that can be drawn here is that a J2EE client sends more data to a J2EE-based Web Service than a .NET client does to a .NET-based Web Service. This should be negative for J2EE (since the load on the network increase) but since this can't be seen in the measurements any conclusions is just pure speculations. Future works should investigate this phenomenon further to find the reasons as well as too see if the performance suffers when a lot of clients are used simultaneous, swamping the network with traffic. Maybe .NET will have better performance since it doesn't generate the same amount of network traffic and the network won't be swamped as quickly.

A third conclusion is that .NET is faster than J2EE when sending data that is not sorted by the Web Service. This is just opposite to the first conclusion where J2EE was faster when the data was sorted. The conclusion drawn from this observation is therefore that the J2EE-based Web Service handles the sorting faster than .NET since .NET handles the transmission of data back and forth between a client and a Web Service faster. The main reason behind this phenomenon is probably the software used for hosting the J2EE and .NET-based Web Services. A future work is therefore to change the software to see if the difference decrease or increase between the platforms.

A forth conclusion is that J2EE seems to do better when the platforms are integrated. When a J2EE-based Web Service is used (Scenario 3) the J2EE-based client is twice as fast as the .NET-based one. When a .NET-based Web Service is used the J2EE-based client is as fast as the .NET-based one. Of course these numbers only show the performance when the Web Service is sorting its input data and any general conclusion covering other configurations can't be made. But still there is a strong indication that J2EE seems to handle the integration better. More experiments has to be done to see how the integration works when just sending a very small amount of data per call as well as using another type of Web Service (non-sorting, database accessing etc).

6 FUTURE WORK

There are of course a lot of future works that can be done in this rapidly expanding area of distributed computer communication that Web Services belong to.

One way to continue the work of this thesis is to use other software both on the client as well as on the server. More commercially software is available for the J2EE platform than NetBeans 5.0 and Sun Java System Application Server Platform Edition 8.2. Software such as Borland JBuilder could have been used as development environment and Borland's Enterprise Server, IBM's WebSphere or BEA's WebLogic as server software. A server version such as Windows 2000 Server could be used to host the .NET-based Web Services instead of the built in web server in Windows XP Professional used.

The equipment used in the experiment was just as the software very limited. Much better computers (faster, more memory etc.) as well as a faster network can be used in future experiments. What is more important is the number of computers. Increasing the number of computers would generate a higher burden on both the network and the actual Web Service trying to respond to the requests. Maybe one of the platforms can handle a growing number of clients better than the other? By increasing the number of computers the experiment will become more realistic since Web Services are intended to handle a large number of simultaneous requests.

Another way to crate a more realistic experiment is to base the developed application on a 'real-world' application, making the results more useful. An example of such a 'real-world' application is an ordering system where clients orders items from the Web Service. The Web Service will then have to communicate with a database to store the orders in as well as other Web Services such as a credit checking Web Service verifying payment and an e-mail Web Service sending out confirmation messages. This would also test how several Web Services can work together to complete a task as well as how the platform can handle Web Service combined with databases.

One important aspect mentioned before in this thesis is that the platforms can be compared in both theoretical and practical ways. This thesis concentrated on explaining the Web Service concept and to do an experiment to compare the platforms in a practical way. The platforms could also be compared in a theoretical way. An example is Java's platform independence compared with .NET's dependence on the Windows operating systems. If the clients are running on a wide variety of operating systems then this aspect would be very important to consider.

The time measuring used in the experiment was very simple. The measuring was simply based on the time measuring that is a part of both the Java and C# programming languages. The problem with this approach is that only time intervals down to 1 ms can be measured. This limitation affected the experiment a lot since smaller time intervals would enable the measuring of a smaller amount of data to be sent per call. A dedicated time measuring tools can be used instead to give more precise values. But this is a very problematic area since an external tool could affect the experiment in unpredictable ways and the basic purpose of an experiment is to observe and not to affect its execution in any way.

During the experiment it was observed that the J2EE-based clients send a lot more data than they received. A way to find out why in the future can be to capture the SOAP-based messages exchanged by the client and the Web Service in both .NET and J2EE and compare them to see why the J2EE messages are larger than the .NET messages.

In the end the experiment investigated as many observations as the time allowed. The aspect of integration between the platforms therefore didn't become as investigated as initially intended. More integration-based scenarios could therefore be investigated. Examples are applications consisting of several Web Services based on both platforms communicating with each other or a mix of clients communicating with a mix of Web Service.

Web Services should also be compared with other techniques for distributed computing such as CORBA, Java RMI etc. Web Services will only become fully adopted if it provides services the other techniques lack such as enabling a higher degree of interoperability than the other techniques or enabling another type of design where one Web Service can be completely made up of other Web Services.

The above ideas are some suggested ways in which future works can continue on the work presented in this thesis and to give more usable results. It is very hard to create a general experiment which will deliver results usable in all situations especially since none of the two platforms are obviously better than the other one. An experiment is always limited, especially in equipment and time. Web Services is such a new area and is constantly being developed and improved which results in better specifications and ultimately better software. This means that experiments has to be done continuously to keep up with the development.

7 REFERENCES

Allamaraju et al., (2001), “Professional Java Server Programming – J2EE 1.3 Edition”, Wrox

Muschamp P., (2004), “An introduction to Web Services”, Vol 22:1 pp 9-18, BT Technology Journal

Nagappan R., Skoczylas R. & Sriganesh R. P., (2003), “Developing Java Web Services – Architecting and developing secure Web Services using Java”, Wiley Computer Publishing

Persson J., (2003), “Comparison of Enterprise Java Beans and .NET from a component point of view”, Master Thesis, Blekinge Institute of Technology, Ronneby, Sweden

Petrovic A. & Andersson J, (2002), “Web Services – aspects on new business possibilities”, Master Thesis, Blekinge Institute of Technology, Ronneby, Sweden

Siegel J., (2000), “CORBA 3 Fundamentals and programming – Second Edition”, Wiley Computer Publishing

Sommerville I., (2001), “Software Engineering – 6th edition”, Addison-Wesley

Szyperski C., Gruntz D. & Murer S., (2002),”Component Software – Beyond Object-Oriented Programming 2nd edition”, Addison-Wesley

wwwBorland (2002), http://bdn.borland.com/article/images/28818/webservices.pdf, Last accessed: 2005-11-29

wwwDalmatianGroup (2001), http://www.dalmatian.com/com_dcom.htm , Last accessed 2006-04-14

wwwMicrosoft(2005), http://msdn.microsoft.com/webservices/default.aspx?pull=/library/en-us/dnwebsrv/html/asmxremotesperf.asp, Last accessed 2006-05-20

wwwMicrosoft(2004),http://msdn.microsoft.com/architecture/soa/default.aspx?pull=/library/en-us/dnmaj/html/aj1soa.asp, Last accessed 2006-09-01

wwwMicrosoft(2003a), http://msdn.microsoft.com/library/default.asp?url=/library/en-us/dnsoap/html/understandsoap.asp, Last accessed 2005-11-24

wwwMicrosoft(2003b), http://msdn.microsoft.com/webservices/webservices/understanding/webservicebasics/default.aspx?pull=/library/en-us/dnwebsrv/html/understandwsdl.asp, Last accessed 2006-01-29

wwwMicrosoft (2001), http://msdn.microsoft.com/archive/default.asp?url=/archive/en-us/dnarxml/html/websvcs_platform.asp, Last accessed 2005-11-01

wwwNetBeans(2006), www.netbeans.org, Last accessed 2006-05-25

wwwOMG(2006a), http://www.omg.org/gettingstarted/corbafaq.htm ,
Last accessed 2006-03-15

wwwOMG(2006b), http://www.omg.org/gettingstarted/orb_basics.htm,
Last accessed 2006-03-15

wwwSunMicrosystems (2001), http://java.sun.com/developer/technicalArticles/J2EE/
j2ee_ws/index.html, Last accessed 2005-11-29

wwwUDDI (2006), OASIS Frequently Asked Questions, http://www.oasis-
open.org/who/faqs.php, Last accessed 2006-02-24

wwwUDDI (2004a), Introduction to UDDI - Important Features and Functional
Concepts, http://uddi.org/pubs/uddi-tech-wp.pdf, Last accessed 2006-02-24

wwwUDDI (2004b), UDDI Version 3.0.2 Spec, http://uddi.org/pubs/uddi_v3.htm,
Last accessed 2006-02-24

wwwUDDI (2002), The Evolution of UDDI,
http://uddi.org/pubs/the_evolution_of_uddi_20020719.pdf, Last accessed 2006-02-24

wwwW3C (2006), http://www.w3.org/TR/2006/CR-wsdl20-primer-20060106/,
Last accessed 2006-01-29

wwwW3C (2004a), http://www.w3.org/TR/ws-arch/, Last accessed 2005-11-29

wwwW3C (2004b), http://www.w3.org/TR/2004/REC-xml-infoset-20040204/,
Last accessed 2006-02-14

wwwW3C (2003a), http://www.w3.org/TR/2003/REC-soap12-part0-20030624/,
Last accessed 2005-11-24

wwwW3C (2003b), http://www.w3.org/TR/2003/REC-soap12-part1-20030624/,
Last accessed 2005-11-24

wwwWebServices.org (2003),
http://www.webservices.org/vendors/systinet/web_services_architecture
/(go)/Files# , Last accessed: 2005-11-29

8 APPENDIX A

This is the WSDL-file generated by WSDL.exe which is a tool in the .NET Framework v. 2.0 Redistributable Package for the .NET-based Web Service.

```
<?xml version="1.0" encoding="utf-8" ?>
- <wsdl:definitions xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tm="http://microsoft.com/wsdl/mime/textMatching/"
    xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:mime="http://schemas.xmlsoap.org/wsdl/mime/"
    xmlns:tns="http://tempuri.org/"
    xmlns:s="http://www.w3.org/2001/XMLSchema"
    xmlns:soap12="http://schemas.xmlsoap.org/wsdl/soap12/"
    xmlns:http="http://schemas.xmlsoap.org/wsdl/http/"
    targetNamespace="http://tempuri.org/"
    xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
- <wsdl:types>
- <s:schema elementFormDefault="qualified"
    targetNamespace="http://tempuri.org/">
- <s:element name="Sort">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="data"
    type="tns:ArrayOfString" />
    </s:sequence>
    </s:complexType>
    </s:element>
- <s:complexType name="ArrayOfString">
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="unbounded" name="string"
    nillable="true" type="s:string" />
    </s:sequence>
    </s:complexType>
- <s:element name="SortResponse">
- <s:complexType>
- <s:sequence>
  <s:element minOccurs="0" maxOccurs="1" name="SortResult"
    type="tns:ArrayOfString" />
    </s:sequence>
    </s:complexType>
    </s:element>
    </s:schema>
    </wsdl:types>
- <wsdl:message name="SortSoapIn">
  <wsdl:part name="parameters" element="tns:Sort" />
    </wsdl:message>
- <wsdl:message name="SortSoapOut">
  <wsdl:part name="parameters" element="tns:SortResponse" />
    </wsdl:message>
- <wsdl:portType name="SortWSSoap">
```

```
- <wsdl:operation name="Sort">
  <wsdl:input message="tns:SortSoapIn" />
  <wsdl:output message="tns:SortSoapOut" />
    </wsdl:operation>
    </wsdl:portType>
- <wsdl:binding name="SortWSSoap" type="tns:SortWSSoap">
  <soap:binding transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="Sort">
  <soap:operation soapAction="http://tempuri.org/Sort" style="document" />
- <wsdl:input>
  <soap:body use="literal" />
    </wsdl:input>
- <wsdl:output>
  <soap:body use="literal" />
    </wsdl:output>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:binding name="SortWSSoap12" type="tns:SortWSSoap">
  <soap12:binding transport="http://schemas.xmlsoap.org/soap/http" />
- <wsdl:operation name="Sort">
  <soap12:operation soapAction="http://tempuri.org/Sort" style="document"
    />
- <wsdl:input>
  <soap12:body use="literal" />
    </wsdl:input>
- <wsdl:output>
  <soap12:body use="literal" />
    </wsdl:output>
    </wsdl:operation>
    </wsdl:binding>
- <wsdl:service name="SortWS">
- <wsdl:port name="SortWSSoap" binding="tns:SortWSSoap">
  <soap:address location="http://localhost/SortWS/Service.asmx" />
    </wsdl:port>
- <wsdl:port name="SortWSSoap12" binding="tns:SortWSSoap12">
  <soap12:address location="http://localhost/SortWS/Service.asmx" />
    </wsdl:port>
    </wsdl:service>
    </wsdl:definitions>
```