A Peer-To-Peer Intermediary for Building Enterprise-Class Web Services

A Thesis

Submitted to the Faculty

of

Drexel University

by

Omar Hasan

in partial fulfillment of the

requirements for the degree

of

Master of Science

in

Computer Science

September 2002

Dedications

To Ami, Abba and Ammar for their endless love and support

Acknowledgements

I am most thankful to my Advisor, Dr. Bruce Char, for giving me the privilege to work with him. His encouragement and guidance were most valuable for me in completing this work. All credit goes to Dr. Char for so much that I learned.

I am thankful to the members of my Thesis Committee, Dr. Bruce Char, Dr. Jeremy Johnson and Dr. Spiros Mancoridis, for their valuable reviews.

I am thankful to Dr. Ali Shokoufandeh for always helping me and giving me insightful advice.

I am thankful to my Undergraduate Advisor, Dr. Sarmad Abbasi, for always helping me so generously.

I am thankful to my Graduate Advisors, Dr. Jeremy Johnson and Dr. Robert Boyer, for patiently listening to my problems and guiding me in the right direction.

I am thankful to all my teachers due to whom I have been able to learn so much over the years. My teachers will always remain a source of inspiration for me.

I am thankful to all my family and friends who have always encouraged and supported me.

Most of all, I am thankful to my parents and my brother, whose love and encouragement was the number one reason I was able to accomplish so much.

Table of Contents

List of Figures ..... **viii**

Abstract ..... **x**

Chapter 1: Introduction **1**

1.1 Background 1

1.2 Problem Description 3

1.3 Our Solution 5

1.4 Outline of Thesis 6

Chapter 2: Background – Web Services and Enterprise-Class Web Services **8**

2.1 Web Services 8

2.1.1 What Problems do Web Services Address and why are they a Good Solution? 8

2.1.2 Web Services Architecture 12

2.1.3 The Building Blocks of Web Services 14

2.1.4 Vendor Support for Web Services 20

2.1.5 Example Web Services 22

2.2 Enterprise-Class Web Services 23

2.2.1 Characteristics of Enterprise-Class Web Services 23

Chapter 3: Problem – Inadequacy of Web Services to Provide Quality-of-Service Guarantees **25**

3.1 What is Quality-of-Service in Web Services? 25

3.1.1 Security 25

3.1.2 Reliability 26

3.1.3 Manageability 27

3.2 Why is Quality-of-Service in Web Services Important to Enterprises? 28

3.3 Why Web Services are Unable to Guarantee Quality-of-Service? 29

3.3.1 HTTP is Unreliable 29

3.3.2 Public Internet is not Secure 30

3.4 Some Views and Opinions about Lack of Quality-of-Service Guarantees in Web Services 30

3.5 Review 32

Chapter 4: Existing Solutions and their Drawbacks 34

4.1 Upgrade Web Services Protocols 34

4.2 Hard Code Desired Quality-of-Service into each Web Service Implementation 35

4.3 Use a Web Service Intermediary 35

4.3.1 What is an Intermediary? 35

4.3.2 What is a Web Service Intermediary? 36

4.3.3 How does a Web Service Intermediary Solve the Problem? 38

4.3.4 Some Implementations Based on the Concept of Web Service Intermediary 40

4.4 A Review of the Discussed Approaches 47

Chapter 5: Our Solution – Peer-to-Peer Intermediary 49

5.1 Objectives 49

5.1.1 Requirement of Customization through Additional Coding 49

5.1.2 Enterprise's Control of the Flow and Management of its Information 50

5.1.3 Ease and Speed of Deployment 51

5.1.4 Scalability 51

5.2 Design 52

5.2.1 Architecture 52

5.2.2 Message Exchange Model 56

5.3 How does PI Meet the Objectives? 57

5.3.1 Requirement of Customization through Additional Coding 58

5.3.2 Enterprise's Control of the Flow and Management of its Information 59

5.3.3 Ease and Speed of Deployment 61

5.3.4 Scalability 63

5.4 Quality-of-Service Features Provided to Web Services by PI 69

5.4.1 Security 69

5.4.2 Reliability 75

5.4.3 Manageability 79

Chapter 6: Implementation and Evaluation 81

6.1 The Bank Web Service 81

6.1.1 Description of Functionality 81

6.1.2 How can the Bank Web Service Benefit from PI? 82

6.1.3 Design and Implementation 84

6.1.4 Sample Messages Exchanged between the Bank Web Service Server and Client 90

6.2 Implementation of PI Proxy 91

6.2.1 Some Basics of the Current Implementation of PI Proxy 92

6.2.2 Limitations of PI-v1 93

6.2.3 Implementation of Basic Message Relay 93

6.2.4 Implementation of Quality-of-Service Features 95

6.2.5 Classes 98

6.2.6 PI Proxy Administration Tool 102

6.3 Evaluation of PI Proxy Version 1.0 103

6.3.1 Does PI Proxy Version 1.0 Achieve the Design Objectives of PI? 103

6.3.2 Test of Quality-of-Service Features Provided by PI Proxy Version 1.0 108

Chapter 7: Conclusion 113

7.1 Review 113

7.2 Contributions 115

7.3 Future Research Possibilities 116

Bibliography 119

Appendix A: Glossary of Acronyms 131

Appendix B: Platforms and Tools Used for the Implementation and Evaluation of PI Proxy Version 1.0 132

List of Figures

1.1 General architecture of existing Web Service intermediaries 4

1.2 Architecture of our Web Service intermediary 5

2.1 Web Services Architecture 12

2.2 A Web Service built from several other Web Services 13

2.3 Web Services standards and protocols 15

2.4 Anatomy of a SOAP message 18

3.1 Web Services insecurity 32

4.1 An intermediary [Barrett1999intermediaries] 36

4.2 A standard Web Service provider and Web Service client [Irani2001intermediaries] 37

4.3 A Web Service provider and Web Service client with an intermediary [Irani2001intermediaries] 37

4.4 A Web Service intermediary composed of more than one component 38

4.5 General architecture of a Web Service intermediary 39

4.6 Architecture of Grand Central Communications' Web Services Network 42

4.7 Architecture of Flamenco Networks' Web Services Network 45

5.1 PI architecture 52

5.2 Multiple clients – configuration 1 55

5.3 Multiple clients – configuration 2 56

5.4 A Web Service application server and its n clients using a central-server-based Web Service intermediary 63

5.5 m Web Service application servers and n clients of each one of them using a central-server-based Web Service intermediary 64

5.6 A Web Service application server and its n clients using PI 66

5.7 `m` Web Service application servers and `n` clients of each one of them using PI 67

5.8 A Web Service application client using PI to communicate with more than one Web Service application server 68

5.9 PI authentication model 71

5.10 An example of PI access control 73

5.11 The PI process of delivering confidential messages 74

5.12 PI guaranteed message delivery 78

6.1 The Bank Web Service architecture 85

6.2 UML class diagram of the Bank Web Service server 86

6.3 The Bank Web Service client's main dialog 88

6.4 UML class diagram of the Bank Web Service client 89

6.5 Basic message relay 95

6.6 Format of the message ID 97

6.7 UML class diagram of the current implementation of PI proxy 99

6.8 UML class diagram showing the generalization/specialization relationship of `system`, `local_relay` and `remote_relay` with `HttpSerlvet` 100

6.9 GUI of the PI proxy administration tool 103

Abstract

A Peer-To-Peer Intermediary for Building Enterprise-Class Web Services

Omar Hasan

Bruce Char

Web Services is a new technology for building distributed systems. The Web Services technology provides the means to make the functionality of an application available over the Internet such that it can be used by remote applications. This is not a new concept; however the novelty of Web Services is that it enables interaction between heterogeneous applications regardless of their development and operational platforms. The Web Services technology also offers some other significant benefits: client applications can access server applications without knowing their location in advance, and Web Service applications can be used as software components to build new applications. Web Services with all their benefits, however, lack quality-of-service features such as security, reliability and manageability. These features are essential for enterprise-class applications.

One recently introduced solution to this problem is that of a Web Service intermediary. In this solution an intermediary is placed between two communicating Web Service applications. The intermediary takes the responsibility of implementing the quality-of-service requirements. Although this is a suitable solution, its current implementations have some undesirable constraints. These constraints include: the requirement of additional coding to enable a Web Service application to interact with the intermediary, and the need for a central server in the intermediary architecture. A central

server is not desirable because it centralizes control in a single authority, and limits the scalability of a Web Service application.

We present a peer-to-peer Web Service intermediary which does not suffer from these problems. Due to the peer-to-peer architecture, it does not require a central server. The solution also does not require any additional coding in a Web Service application, which makes it quickly deployable. Our solution is convenient and practical for building enterprise-class Web Services.

Chapter 1: Introduction

1.1 Background

Distributed systems have proven their usefulness over the past several years. Email and World Wide Web are very popular examples of distributed systems. Dedicated distributed systems are used by banks, airlines, universities and almost every other enterprise for its operation. BannerWeb [Drexel2002bannerweb] (Drexel University's system for student and employee records) is an example of such a system.

Many technologies have been developed for building new distributed systems. These technologies include CORBA [OMG2002corba], Java RMI [Sun2002javarmi] and DCOM [Microsoft2002dcom]. Although these technologies have been useful to some extent, they have some problems which keep them from being largely successful. The key problem is as follows:

These technologies communicate information in formats that are not universally recognized or accepted. Agreement on a single universal standard has not been possible because the vendor of each technology is unwilling to give it up for a technology by another vendor.

The result is that all these technologies lack heterogeneity. Lack of heterogeneity means that a technology is limited to a few computing platforms. A distributed system built on one of these technologies cannot function unless each of its components is built with that same technology. The following example illustrates the drawback of lack of heterogeneity in these technologies:

A company has several suppliers. The suppliers would like to be connected to the company's information system for acquiring information such as inventory and sales. The company uses one of the distributed systems technologies available for its computing platform for example DCOM, to allow access to the suppliers. It is possible that some or many of the suppliers have a platform that does not support that technology. In that case those suppliers would not be able to connect to the company.

This problem prompted the development of the Web Services technology. Web Services technology is based on XML [W3C2002xml]. XML is a universally accepted standard for information representation. XML is also platform independent, primarily because it is text based. The components of a distributed system built on the Web Services technology interact using XML. XML acts as a bridge between dissimilar platforms. The Web Services technology can therefore be used to build distributed systems from heterogeneous components.

In addition to enabling distributed systems to be built over heterogeneous platforms, Web Services have several other novel features which include the following:

- Web Services can be published on the Internet with special registries based on a protocol called UDDI [UDDI2002uddi]. Clients can search and locate suitable Web Services from these registries. For example a client could be looking for a Web Service that performs a difficult mathematical computation for the lowest price. It can find such a Web Service and use it without knowing its location in advance. All this can be done programmatically without human involvement.
- Web Services can be used as components of an application. A Web Service in turn can also use other Web Services as its components. For example an e-commerce

application can be built that uses a credit card processing Web Service and a shipping information processing Web Service as its components. The credit card processing Web Service can itself use other Web Services such as a customer credit rating Web Service.

1.2 Problem Description

In the previous section we introduced Web Services and stated their benefits. The Web Services technology unfortunately also suffers from a significant problem; there is no provision of quality-of-service features such as security, reliability and manageability in the Web Services technology.

Quality-of-service is very important in enterprise-class distributed systems (distributed systems used by enterprises for their operation). For example a bank transferring money over a distributed system must be absolutely certain that the system is secure and reliable.

Some solutions have already been proposed to this problem. One of these solutions is that of a Web Service intermediary. In this solution an intermediary is placed between the communicating Web Service applications. The Web Service applications then communicate through this intermediary instead of communicating directly. The intermediary takes the responsibility of enforcing the quality-of-service requirements. This is a suitable solution to the problem. However, the existing implementations of this solution place many constraints on the Web Services that use them. These constraints include the following:

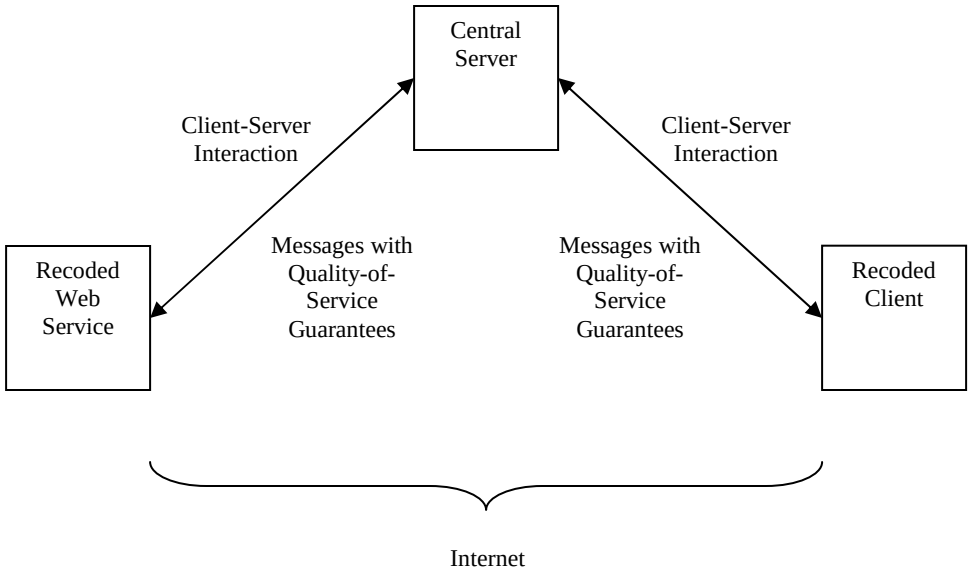

Figure 1.1: General architecture of existing Web Service intermediaries

- Some implementations of the Web Service intermediary solution require that the Web Service applications be recoded to be able to communicate with the intermediary. This is clearly undesirable since recoding requires substantial time and effort.
- The existing implementations of the Web Service intermediary solution are based on central-server architecture. This leads to more constraints on the Web Services that uses these implementations. It places control of the information flow and the configuration of quality-of-service requirements with a single party. The other parties involved in the communication are left deprived of this control. A central server also limits the scalability of a Web Service.

A Web Service intermediary should be developed that does not suffer from these problems, yet fulfills the Web Services' quality-of-service requirements.

1.3 Our Solution

In this thesis, we present a novel Web Service intermediary that does not suffer from the problems described in the previous section. Our Web Service intermediary is based on peer-to-peer architecture.

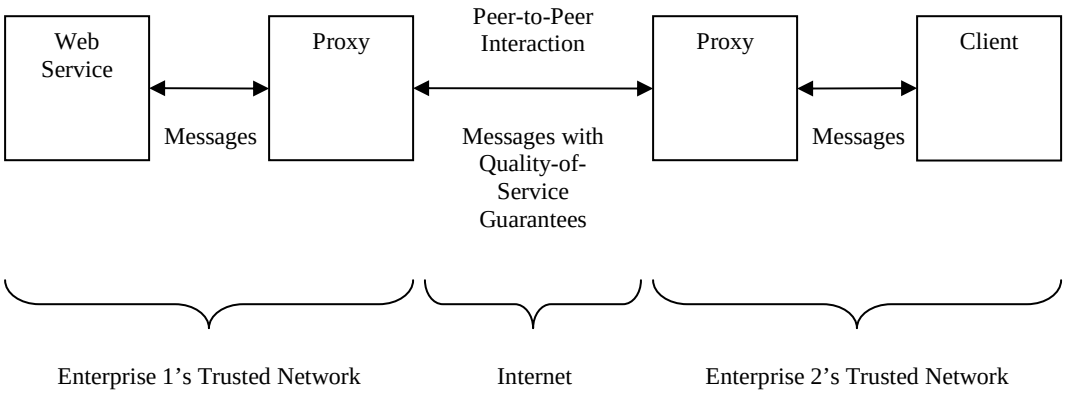

Figure 1.2: Architecture of our Web Service intermediary

A proxy is installed for each Web Service application on its trusted network. The Web Service application sends and receives messages through its proxy. The communication between the Web Service application and its proxy is presumed to meet the quality-of-service requirements since it takes place over a trusted network. Two Web Service applications that wish to communicate do so through their proxies. The proxies

communicate with each other in peer-to-peer fashion over the Internet. Since the Internet is unreliable and insecure, the proxies take various measures to implement the quality-of-service requirements in their communication.

Our Web Service intermediary is designed such that there is no need for recoding of Web Service applications. Since our Web Service intermediary is based on the peer-to-peer architecture, there is also no need for a central server.

1.4 Outline of Thesis

Chapter 2 is an introduction to Web Services. We discuss the rationale for Web Services, their architectural and functional elements, and the current Web Services offerings by major vendors. We also describe and characterize enterprise-class Web Services in this chapter.

In Chapter 3, we identify the problem that we have addressed in this thesis, that is, the inadequacy of Web Services to provide quality-of-service guarantees. We discuss what is meant by quality-of-service guarantees, what is their importance and why Web Services are unable to provide them.

Chapter 4 is a description of the solutions that have already been proposed and some of their implementations. We discuss the benefits and drawbacks associated with each of these solutions. We reason that there is a need for a better solution.

In Chapter 5, we present our solution to the problem. We present its architecture and functionality. We discuss how it overcomes the problems in the existing solutions. Throughout this chapter we discuss our solution from a design perspective.

In Chapter 6, we describe the implementation of our solution and its evaluation. We also describe a sample Web Service that we have developed for exercising and evaluating our solution.

In Chapter 7, we present conclusions and identify future research possibilities.

Chapter 2: Background – Web Services and Enterprise-Class Web Services

In this chapter we introduce the Web Services technology. We discuss the rationale for Web Services, their architectural and functional elements, and the current Web Services offerings by major vendors. We also describe and characterize enterprise-class Web Services in this chapter.

2.1 Web Services

Web Services is an emerging technology for building distributed systems. More specifically, the Web Services technology can be used to make the functionality of an application accessible over the Internet. This functionality can then be used by remote applications. For example a company's inventory application can use the Web Services technology to access a supplier's sales application and place an order for depleted supplies.

2.1.1 What Problems do Web Services Address and why are they a Good Solution?

A number of technologies already exist for building distributed systems by enabling application-to-application interaction, such as: CORBA [OMG2002corba], Java RMI [Sun2002javarmi] and DCOM [Microsoft2002dcom]. However, these technologies have some drawbacks [Flamenco2002wsnetworks], which include the following:

- These technologies communicate information in formats that are not universally recognized or accepted. For example CORBA and Java RMI are two equally popular

technologies but an object serialized under CORBA is not deserializable under Java RMI and vice versa. The drawback of this limitation is that two enterprises that support different technologies are unable to communicate.

- These technologies communicate information in binary formats. Almost all enterprise networks are protected by firewalls. Binary information is subjected to very strict scrutiny by firewalls since binary information can contain executable code which may have been programmed by an attacker for malicious purposes. Firewalls do not allow suspicious binary information to pass through. This can result in the disruption of communication.
- These technologies provide access to an application's functionality at the object method level. This means that the application is viewed as a bunch of objects and not as one single module. This makes the development of a client application complex since it is easier to understand and use one coherent module than several objects. There is also no single widely accepted standard for describing the interface of an application under these technologies.
- A client application must know the exact location (for example URL, IP address etc.) of a server application in advance to be able to access it.
- Each of these technologies places some other restrictions on the communicating applications. CORBA requires each communicating application to use an Object Request Broker (ORB) software from the same vendor [Gisolfi2001corba]. Interoperability between ORBs from different vendors is available in some cases but limited to very basic services. Java RMI is available to applications written only in Java. DCOM is available primarily on the Windows platform.

The Internet promotes open standards and interaction between diverse platforms. The above mentioned drawbacks make these technologies particularly unsuitable for wide use over the Internet.

These problems in the existing technologies called for a new technology, which should have the following characteristics:

- The technology should communicate information in a universally recognized and accepted format.
- The technology should be “firewall-friendly”.
- The technology should be able to represent an application’s programming interface as a single module and not just as a bunch of objects.
- Client applications should be able to access server applications even if they do not know their location in advance.
- The technology should not be restricted to specific platforms or vendors. It should provide interoperation between applications on disparate platforms.

Web Services technology has been developed as a response to these needs. The key strengths of Web Services are as follows:

- Web Services technology is based on the universally accepted standard of XML [W3C2002xml]. XML is a standard for information representation. XML is understood on all platforms and accepted by all enterprises. XML acts as a bridge between dissimilar platforms and enterprises.

- All Web Services information representation and exchange protocols such as XML, SOAP [W3C2002soap], HTTP [W3C2002http], UDDI [UDDI2002uddi] and WSDL [W3C2002wsdl] are text-based. This means that data meaningful to a Web Service entity on one platform is also readable by another Web Service entity on a completely dissimilar platform. Text-based information representation and exchange also make Web Services firewall friendly. Moreover, text is human-readable, which results in easier debugging.
- A Web Service can be viewed as a self-contained module. New applications can be developed by using one or several Web Services as their sub-modules. For example an e-commerce application can be built that uses an existing credit card processing Web Service as its sub-module. This software architecture is also known as the service-oriented architecture [Burbeck2000wstao].
- Web Services can be published on the Internet with special registries based on a protocol called UDDI [UDDI2002uddi]. Clients can search and locate suitable Web Services from these registries. For example a client could be looking for a Web Service that performs a difficult mathematical computation for the lowest price. It can find such a Web Service and use it without knowing its location in advance. All this can be done programmatically without human involvement. This kind of distributed system is also called a Semantic Web [Klein2001semantic].

It is obvious from this discussion that Web Services technology is a good solution to the problems that earlier technologies suffered from.

2.1.2 Web Services Architecture

As described by [Tsur2001revolution], the basic Web Services architecture consists of three entities:

- A service provider, which publishes the availability and the nature of its services in a registry.
- A service broker, which provides support via its registry, for the publication and the location of services offered by providers. In its simplest form this role can be thought of as that of the yellow pages directory.
- A service requestor, which finds services of interest via the service broker and once found, binds to the services via the service provider.

This architecture is illustrated in the following figure:

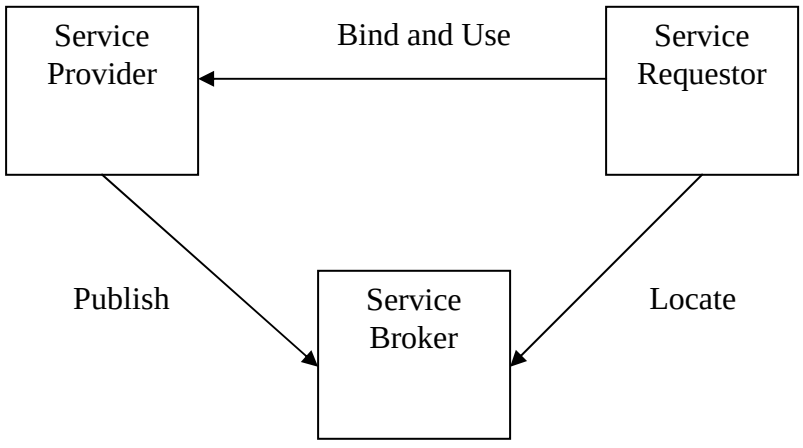

Figure 2.1: Web Services Architecture

A service provider is usually referred to simply as a Web Service. A service requester is usually referred to as a client.

Another perspective of the Web Services architecture is a single Web Service formed by the composition of several other Web Services. As mentioned earlier, this architecture is known as the service-oriented architecture. This architecture is also described by [Tauber2001webservices] as a Business Web. The following figure illustrates this architecture:

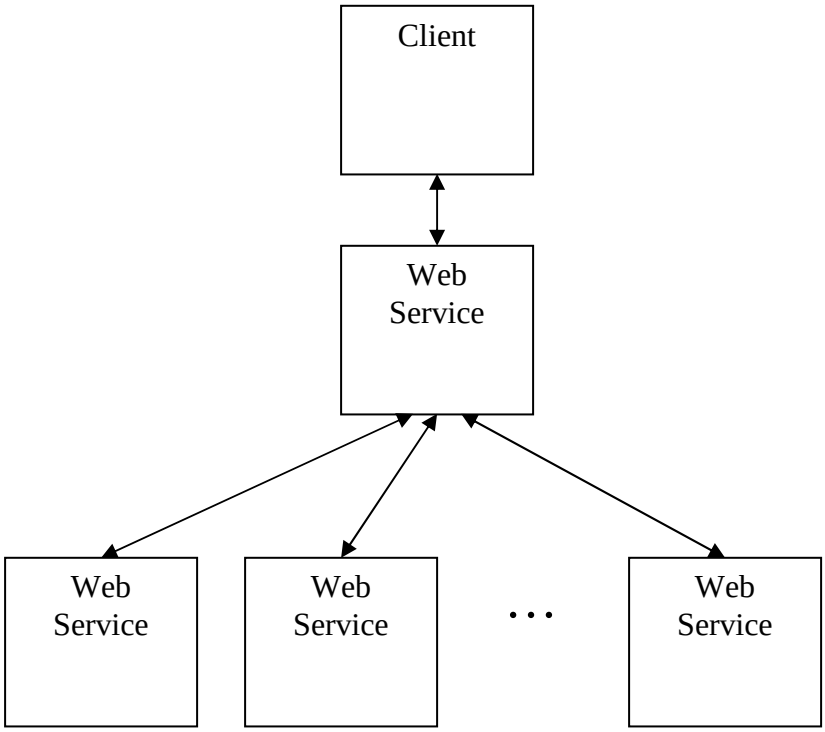

Figure 2.2: A Web Service built from several other Web Services

2.1.3 The Building Blocks of Web Services

The service provider, service requester and the service broker, all interact by sending and receiving eXtensible Markup Language (XML) text messages. This exchange takes place using the Simple Object Access Protocol (SOAP). SOAP needs an Internet protocol such as: HTTP, SMTP or FTP to function. The most popular combination is SOAP over HTTP.

The service broker is implemented using the Universal Description, Discovery and Integration (UDDI) protocol. Web Services Description Language (WSDL) is an XML based language that is used to describe a Web Service. The UDDI contains in its database, a WSDL document for each published Web Service.

Clients can search the UDDI for suitable Web Services. The search results are based on the information provided in the WSDL documents published in the UDDI. Once the client locates a suitable Web Service, it retrieves its complete WSDL document. The information provided in the document is sufficient for the client to start interacting with the Web Service on a one-on-one basis.

The following subsections provide a brief overview of these building blocks of the Web Services technology.

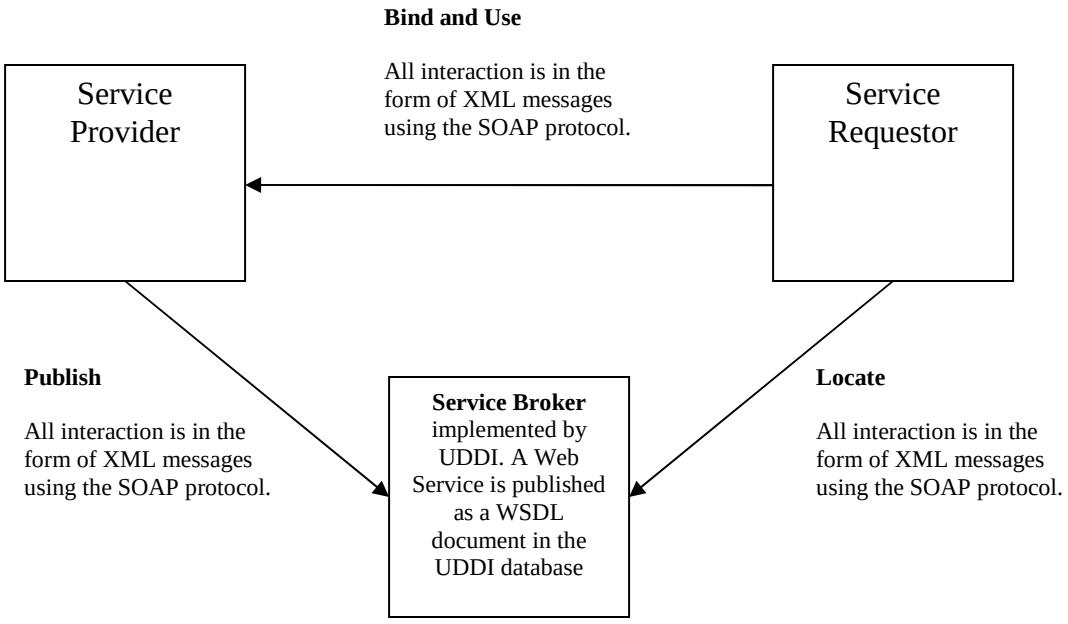

Figure 2.3: Web Services standards and protocols

2.1.3.1 XML

eXtensible Markup Language (XML) [W3C2002xml] is a text-based markup language which uses tags to identify data. Unlike HTML, an earlier text-based markup language which specifies how to display data, XML specifies the structure of data. The benefit of this approach is that having identified the structure of some given data, it is up to each individual user, as to how to use it. XML has been universally accepted as the standard language for cross-platform information exchange. XML has several novel characteristics [Armstrong2002introxml]:

- It is text-based which makes it platform independent and simple to use.
- It can be used to describe structured data.
- It is easy for computer programs to parse XML documents.

- XML can be used to define new application specific languages. These languages inherit the novel characteristics of XML. SOAP (described in the next section) is an example of such a language.

The most significant reason for XML's popularity is its platform neutrality.

Following is an example of how information is represented in XML. This instance of XML represents the date August 2, 2002.

```
<date>
   <month>
      August
   </month>
   <day>
      2
   </day>
   <year>
      2002
   </year>
</date>
```

2.1.3.2 SOAP

Simple Object Access Protocol (SOAP) [W3C2002soap] is a protocol to exchange XML messages between applications. The key strength of SOAP is that it is platform neutral. Applications different from each other in terms of development language and operating platform are able to interact using SOAP. This platform neutrality is due to its use of XML for information exchange and an Internet protocol, such as HTTP, for transport. SOAP messages can be used to exchange commands and data. SOAP also provides a mechanism for Remote Procedure Calls (RPC). SOAP supports one-way and two-way messaging. A one-way message is a notification or an event from one application to the other. A two-way message is a request-response pair.

2.1.3.2.1 Anatomy of a SOAP Message

A SOAP message has three parts: a mandatory envelope, an optional header, and a mandatory body. The envelope encapsulates the header and the body. The header may contain additional information about the message such as instructions on how to process the message. The body of a SOAP message contains the actual XML data intended for the destination application.

The following figure illustrates the anatomy of a SOAP message with the aid of a simple SOAP message. The SOAP message in the figure contains the XML representation of the date August 2, 2002, given in the previous section.

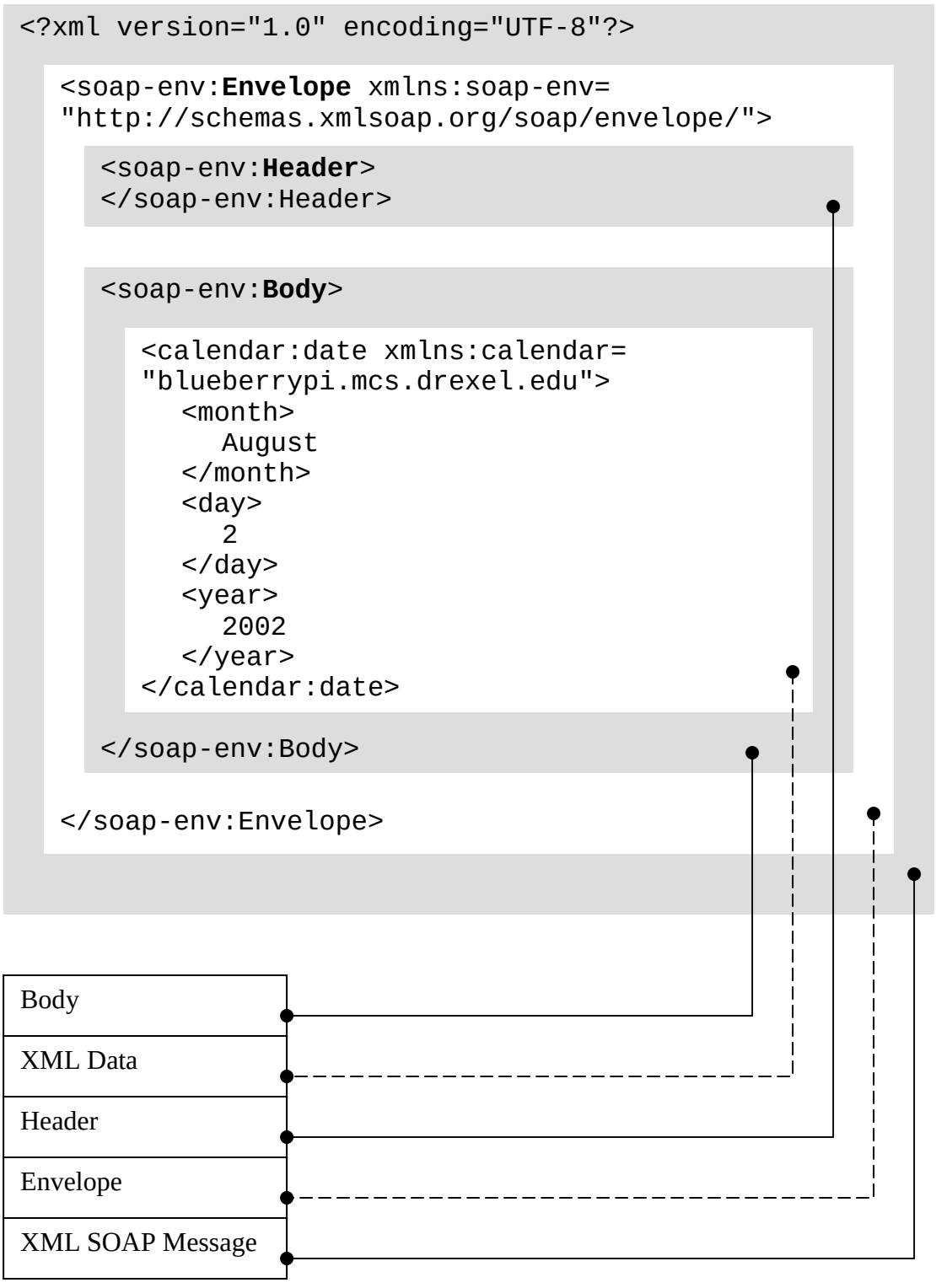

Figure 2.4: Anatomy of a SOAP message

2.1.3.3 WSDL

Web Services Description Language (WSDL) [W3C2002wsdl] is an XML based language that can be used to describe the programmatic interface of a Web Service in a standardized manner. WSDL describes a Web Service as a single entity. This is in contrast to other description languages, such as Java Interface Description Language (IDL) [Sun2002javaidl], which describe the interface simply as a list of methods for each object. A WSDL description is able to provide a range of useful information about a Web Service. This includes textual descriptions of the interface, which can be helpful for the developers of an application planning to use that Web Service. Detailed information on WSDL can be found at [W3C2002wsdl].

2.1.3.4 UDDI

The idea known as the Semantic Web [Klein2001semantic] has recently gained a lot of interest. The goal of the Semantic Web is to give resources well-defined, universally-understandable meaning. One key benefit of doing this is that the right resources can be found on the Internet with very high accuracy. It also enables computer programs to understand and process information about resources available on the Internet.

The Universal Description, Discovery and Integration (UDDI) [UDDI2002uddi] mechanism in conjunction with WSDL implements this idea for Web Services. A UDDI registry is a searchable repository of Web Services. Like other Web Services protocols, UDDI is also XML-based. Web Services can register themselves with a UDDI registry. Clients can search and locate the service they are looking for in the UDDI registry. Since the UDDI registry contains semantic descriptions of Web Services in the form of WSDL, it is easier for clients to identify the exact service they require. A UDDI registry can be

compared to a telephone book's white, yellow and green pages [Seebeyond2001webservices]. It allows Web Services to be listed by name, location, service provided etc. For example Web Services that provide local traffic conditions could be listed by their localities. A detailed look at the discovery process can be found in [UDDI2002tech].

2.1.4 Vendor Support for Web Services

Web Services have gained significant support from all major software vendors, including Sun Microsystems, Microsoft and IBM. All vendors agree upon the basic Web Services standards, such as XML, SOAP, WSDL and UDDI. Following is a brief overview of the Web Services support being offered by major vendors.

2.1.4.1 Sun Microsystems

Sun has recently released the Java Web Services Developer Pack (JWSDP) [Sun2002jwsdp], which includes several useful APIs and tools for developing and deploying Web Services. JWSDP is part of Sun's SunONE Web Services Architecture [Sun2002sunone]. JWSDP can be used with Sun's J2SE [Sun2002j2se] and J2EE [Sun2002j2ee] platforms. JWSDP includes the Java API for XML Messaging (JAXM), Java API for XML Processing (JAXP), Java API for XML Registries (JAXR) and the Java API for XML-based RPC (JAX-RPC). It also includes the Tomcat Java Servlet container, which can be used to host Web Services. JWSDP is a powerful and comprehensive package for Web Services implementation. After releasing two Early Access (Beta) versions, Sun released the first production versions of JWSDP in June

2002. The best thing about JWSDP is that it is free of charge. However, from our experience with JWSDP, we feel that it currently lacks comprehensive documentation.

2.1.4.2 Microsoft

The heavily advertised Microsoft .NET [Microsoft2002dotnet] is Microsoft's platform for Web Services. Microsoft has updated its Visual Studio 6.0 Integrated Development Environment (IDE) to Visual Studio .NET, which adds support for the development of Web Services. Visual Studio, as always is a relatively easy and simple yet powerful development environment. However, Web Services developed on the .NET platform can run only on the Windows Operating System. An integral part of Microsoft's .NET platform is the Common Runtime Language (CLR) which provides services to help simplify Web Services development and deployment.

2.1.4.3 IBM

IBM's WebSphere [IBM2002websphere] has been a popular application server. IBM has recently evolved WebSphere into a full-fledged Web Services platform. WebSphere offers to "build, deploy and integrate high-performance Web sites with advanced e-business features using open standards" [IBM2002websphere]. IBM is also planning to include Web Services support in its other major products, including: DB2, Lotus and Tivoli [WSorg2002wsoverview].

2.1.5 Example Web Services

2.1.5.1 Microsoft Passport

Microsoft Passport [Microsoft2002passport] is a Web Service which centralizes users' authentication information. An application that needs to authenticate a user can simply connect to Microsoft Passport and do so. This is opposed to the application having to implement its own authentication mechanism. The benefit that is brought to the users is that they can register with Passport once, and use the same authentication information to log in to any online service which uses Passport. This saves users from remembering a different login and password for each online service that they use.

Microsoft .NET My Services [Microsoft2002myservices] is a set of proposed Web Services, which will use Microsoft Passport to authenticate its users. This is an example of a Web Service application using another Web Service application as one of its components. Microsoft .NET My Services include personal organization tools such as an address book, a scheduler etc.

2.1.5.2 California Traffic-Reporting Web Service by InterKnowlogy

InterKnowlogy LLC [InterKnowlogy2002ik] is a consulting and software engineering firm based in the San Diego area. InterKnowlogy has developed a Web Service which exposes California Department of Transportation's real time traffic information. Applications can connect to this Web Service and retrieve customized traffic information in real time. The specific use of this information is up to the client applications. They can use it for trip planning, reporting or any other purpose. This would not be possible if this

information were available only on a simple web site. The information on a web site is intended for human consumption and is generally not reusable.

An example application would be a vehicle's computer retrieving the current traffic conditions from the traffic Web Service and using them to constantly calculate the fastest route to the destination as the vehicle is being driven.

2.2 Enterprise-Class Web Services

Web Services technology is currently in its initial stages and has therefore some issues that are being worked out. A simple Web Service implementation currently lacks the quality-of-service that enterprises demand from their software. The reasons for the lack of quality-of-service in Web Services are discussed in detail in the next chapter. A Web Service that is able to meet an enterprise's quality-of-service requirements can be termed as an enterprise-class Web Service.

2.2.1 Characteristics of Enterprise-Class Web Services

As identified by [Jones2002enterpriseclassws], characteristics of enterprise-class Web Services include:

- Security – The ability to determine variable access rights based on a user identity or class of identity. Security should address data transport to ensure data integrity, confidentiality, authentication and non-repudiation.
- Reliability – Once a business process or object is placed into production, mechanisms should exist to ensure reliable access to that resource. In the case of data transport, reliability should address guaranteed and once-and-once-only delivery of data.

- Manageability – This requires robust, precise monitoring tools that allow prediction, identification and resolution of a large number of deployed interfaces, objects or services.
- Configurable Usage Policies – These govern everything from supported vendor software, to required hours of availability, naming policies, error-handling techniques, and security requirements.
- Physical scalability – This refers to the ability to handle increasing volumes of usage without requiring drastic reconfiguration, testing and new hardware.
- Developmental scalability – This addresses the ability to spread development projects among multiple teams and locations without compromising source integrity, reuse and quality assurance.

Chapter 3: Problem – Inadequacy of Web Services to Provide Quality-of-Service Guarantees

In the previous chapter, we discussed the emerging Web Services technology. We saw that the Web Services technology has several advantages over earlier technologies in building distributed systems. We also pointed out that although the Web Services technology has its benefits, its implementations lack quality-of-service guarantees, which are important for enterprise software. In this chapter we will discuss this problem in detail. We start by clarifying what is actually meant by quality-of-service in Web Services.

3.1 What is Quality-of-Service in Web Services?

Quality-of-Service (QoS) in Web Services refers to their non-functional properties, such as reliability, security and manageability [Mani2002qos]. In the previous chapter, we listed some quality-of-service characteristics of Web Services. Here we discuss in detail the three quality-of-service characteristics that are of greater significance.

3.1.1 Security

There are many aspects to security in a Web Service, such as: authentication, authorization/access control, confidentiality and non-repudiation.

- *Authentication* is the process of verifying that the sender of a message actually is who he claims to be. An authenticated client is one whose identity has been verified as a

valid user of the service and is therefore allowed access to the service. Authentication is a defense mechanism against unauthorized access.

- *Authorization/access control* works closely with authentication. Once a client is authenticated, he is given access to the service. However, it may not be desirable to allow all the users access to every function and data that the service offers. The manager of the service should be able to assign each user a different level of access. Authorization/access control in a Web Service enforces the level of access granted to a particular client.
- Maintaining *confidentiality* in a Web Service is making sure that a message meant for a particular party is not readable by unauthorized third parties. This is to safeguard in-transit confidential information from falling in the wrong hands.
- *Non-repudiation* is a guarantee that once a particular transaction has been made between two parties (i.e. the Web Service and the client), neither one of them is able to deny it.

3.1.2 Reliability

Reliability of a Web Service refers to the quality of message delivery between the Web Service and the client. When a message is sent from one party to the other, there should be a level of assurance that the message will be delivered, delivered in order and that it will be delivered exactly once.

- It is important for most applications that messages are always delivered to the destination. Minor glitches such as temporary network outages should not stop the delivery of messages. *Guaranteed message delivery* provides the sender and receiver

a certain level of assurance that messages will get delivered unless the destination is permanently unreachable.

- *Ordered delivery* of messages is important for some applications; real-time multimedia streams are an example.
- *Exactly-once delivery* of a message guarantees that even if duplicate copies of a message are received, only a single copy of the message is delivered to the destination. This is important if the application has operations that are not idempotent. An idempotent operation is an operation that can be performed repeatedly with the same effect as it had been performed exactly once [Coulouris2001distributed].

3.1.3 Manageability

A Web Service requires a degree of management to keep it operating smoothly. Management includes monitoring the Web Service for performance and exceptions, maintaining logs of activity for future reference, and client management.

- The manager of a Web Service should be able to *monitor* its various aspects in real-time. This provides the manager the opportunity to detect any problems in the Web Service early on. The problems can then be resolved before they make a significant impact.
- *Logging* is saving the monitoring information for permanent record. Logs are helpful to compare the activity of a Web Service over a period of time. The logs themselves should be manageable.
- A Web Service should have easy but effective *client management*. For example it would be very inconvenient if the addition of each new client requires coding and recompilation of the service.

3.2 Why is Quality-of-Service in Web Services Important to Enterprises?

Quality-of-service guarantees in Web Services are very important for enterprises because without them a Web Service is prone to many problems, which are unacceptable for proper business function. We can illustrate this with the help of an example from [Flamenco2002wsnetworks].

Imagine that a manufacturer has implemented a Web service so that a customer's application can dispatch orders directly into the manufacturing control application of the manufacturer. In absence of the above-described quality-of-service guarantees, a few things could go wrong:

- An unauthorized user/application could connect to the Web service and invoke an order by pretending to be a certain customer. *Reason: Lack of authentication.*
- A competitor of the manufacturer could intercept in-transit messages, find out the production schedule of the manufacturer and have complete knowledge of the orders that the manufacturer receives. *Reason: Lack of confidentiality*.
- A glitch in the information system of the customer would not record the dispatch of the order to the manufacturer's system and since their would be no trail that showed that the order came from the customer, the customer would be within their rights to refuse the order. *Reason: Lack of non-repudiation*.
- Due to the slowness of the connection, an application can time-out, retry the order and unintentionally order double the quantity that the customer required. *Reason: Lack of exactly-once delivery*.

- During a tax audit of the manufacturer, there would be no way to ascertain that a customer had indeed made an order at a certain date and at a certain price because there would be no logs. *Reason: Lack of logging*.

It is obvious from this example that unless a Web Service has the quality-of-service guarantees that we discussed earlier, it is not practical for an enterprise to use that Web Service.

Although quality-of-service guarantees in Web Services are important for all enterprises, they are critical for some enterprises, such as:

- E-commerce based enterprises, for example e-commerce web-sites
- Financial institutions, for example banks
- Healthcare organizations, for example hospitals and pharmacies
- Transportation enterprises, for example airports

3.3 Why Web Services are Unable to Guarantee Quality-of-Service?

There are several reasons why the current Web Services infrastructure is unable to provide quality-of-service guarantees. We discuss a few of them here.

3.3.1 HTTP is Unreliable

The Web Services message delivery protocol, SOAP, relies on an Internet protocol for transport of its messages. The most popular choice is the Hyper Text Transfer Protocol (HTTP). The reason for this choice is the common availability of HTTP infrastructure due to the popularity of the World Wide Web (WWW), which uses this protocol.

HTTP is a best-effort delivery service [Mani2002qos]. It suffers from the following problems:

- It does not guarantee the delivery of a message to the destination.
- It does not guarantee the delivery of messages in the correct order.
- It does not guarantee that a message will be delivered only once.

These shortcomings are generally not a problem for the WWW, for which HTTP is intended. However, for enterprise-class application-to-application interaction, it creates the problems that we noted earlier. It is also noteworthy that HTTP is not the only protocol with these shortcomings; most other Internet protocols such as FTP and SMTP have similar characteristics.

3.3.2 Public Internet is not Secure

Web Services communicate with clients over the public Internet. This means that all communication is visible to the public. Moreover, Web Services protocols such as SOAP do not include any built-in security mechanisms. This makes Web Services vulnerable to security attacks. The problem of lack of security in Web Services is discussed in detail in [Powell2001security].

3.4 Some Views and Opinions about Lack of Quality-of-Service Guarantees in Web Services

This section lists some views and opinions of industry professionals about the lack of quality-of-service guarantees in Web Services.

Tim Hilgenberg, Chief Technology Strategist for Hewitt Associates, says: “If you want to do real business processes, the keys are security, non-repudiation and reliable messaging” [Fontana2002security].

Matt Powell of Microsoft Corporation says: “I must confess that in reality, there is almost no mention at all of security in the SOAP specification. You could say that SOAP, as it exists today, is not secure” [Powell2001security].

Concerned about the lack of quality-of-service guarantees in Web Services, John Studdard, Chief Technology Officer at VirtualBank in Florida, says: “It is foolish now to build Web services that run outside the firewall” [Fontana2002security].

A poll conducted by Evans Data Corporation [EvansData2002evansdata], notes security and authentication issues as the biggest obstacle to Web Services implementation. The results of this poll are illustrated in the following graph:

Which do you think will be the biggest obstacle to Web Services implementation?

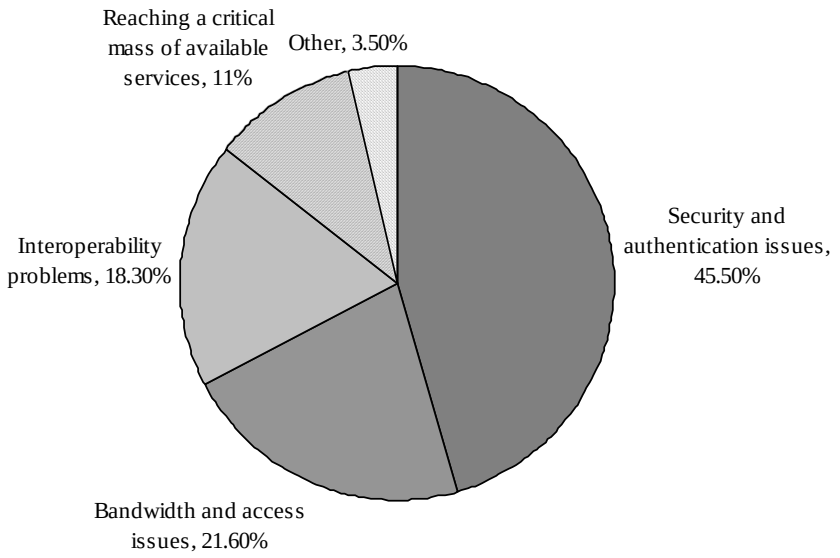

Figure 3.1: Web Services insecurity

3.5 Review

In this chapter, we first defined what is meant by quality-of-service in Web Services. Then with the help of an example we established why it is important for enterprises to have quality-of-service guarantees in Web Services. We also explained some reasons which contribute to the inability of Web Services to guarantee quality-of-service. Some views and opinions of industry professionals were then presented, which highlight the concern for the lack of quality-of-service guarantees in Web Services.

From discussions in this and the previous chapter, it is clear that although Web Services technology is quite advantageous, the lack of quality-of-service guarantees makes it impractical for enterprise-class applications. This is a problem that needs to be addressed. The following chapter discusses some solutions that try to fix this problem.

Chapter 4: Existing Solutions and their Drawbacks

The Web Services community is well aware of the problems discussed in the previous chapter. Several solutions are being considered and new solutions are also being proposed. Here we discuss three approaches to add enterprise-class quality-of-service to Web Services.

4.1 Upgrade Web Services Protocols

One of the obvious solutions is to upgrade the Web Services protocols, so that they include support for enterprise-class quality-of-service. The positive aspect of this solution is that it would be integrated and long-term. However, this approach has some drawbacks. First of all, it would take considerable time and effort to develop and standardize the new protocols. Standardization activities are usually very time-consuming. Second of all, it may not even be desirable to add extra burden on the existing Web Services protocols. One of the reasons of Web Services' practicality is the simplicity and minimality of its protocols. Adding more functionality to the protocols could make them inefficient or cause interoperability and integration problems, which would defeat the purpose of Web Services [Fontana2002security].

Anyhow, this approach does have potential and it has yet to be fully explored. Several groups, including many supported by major companies, are currently working in this area. Examples of work in progress include: Reliable HTTP (HTTPR) [Todd2002httpr] and Web Services Security Language (WS-Security) [Atkinson2002wssecurity].

4.2 Hard Code Desired Quality-of-Service into Each Web Service Implementation

Another solution is to hard code the desired quality-of-service into each Web Service implementation as it is developed. The advantage of this solution is that it is immediately available. However, this approach also has several significant drawbacks. Manually implementing quality-of-service features into a Web Service is an arduous task. It requires considerable time, effort and skill on the developers' part. Moreover, custom code for each Web Service means that there is a greater risk of errors. Making sure that a Web Service has the desired level of quality-of-service using this approach, would require thorough and time-consuming testing. The worst aspect of this approach however, is that non-standard mechanisms to achieve quality-of-service would severely limit the interoperability and integration capabilities of a Web Service.

4.3 Use a Web Service Intermediary

4.3.1 What is an Intermediary?

As defined by [Barrett1999intermediaries]:

> "An intermediary is a computational element that lies between an information producer and an information consumer on an information stream."

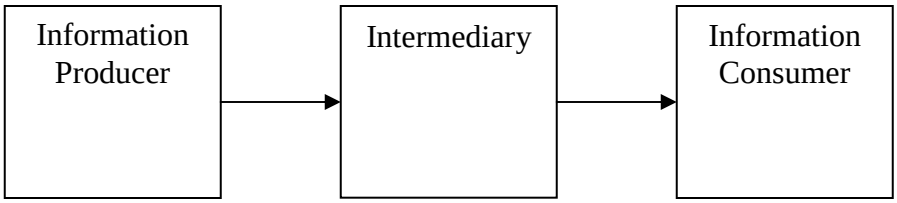

Figure 4.1: An intermediary [Barrett1999intermediaries]

The concept of intermediaries is well-established. According to [Barrett1999intermediaries], “Intermediaries can turn ordinary information streams into smart streams that enhance the quality of communication”. Intermediaries are usually used to monitor traffic, bridge dissimilar information streams or add value to a simple data flow. A common intermediary is a web proxy between a web server and a web browser.

4.3.2 What is a Web Service Intermediary?

A Web Service intermediary is an intermediary placed between the Web Service provider and the Web Service client [Irani2001intermediaries]. The concept of Web Service intermediaries is a recent one.

A Web Service provider and Web Service client without an intermediary are illustrated in the following figure:

Figure 4.2: A standard Web Service provider and Web Service client
[Irani2001intermediaries]

In this case, SOAP messages are exchanged over an Internet protocol such as HTTP, directly between the Web Service and the client.

A Web Service provider and Web Service client with an intermediary are illustrated in the following figure:

Figure 4.3: A Web Service provider and Web Service client with an intermediary
[Irani2001intermediaries]

In this case, the intermediary lies between the Web Service and the client. All messages are directed to the intermediary, which in turn forwards them to their destination.

It is noteworthy that an intermediary may not be a single entity. An intermediary may be composed of a system of entities. It is also possible for the Web Service and the client to use more than one intermediary. An intermediary composed of several distributed components is also called a Web Services Network. The Web Service intermediary shown in the following figure is an example of a Web Services Network:

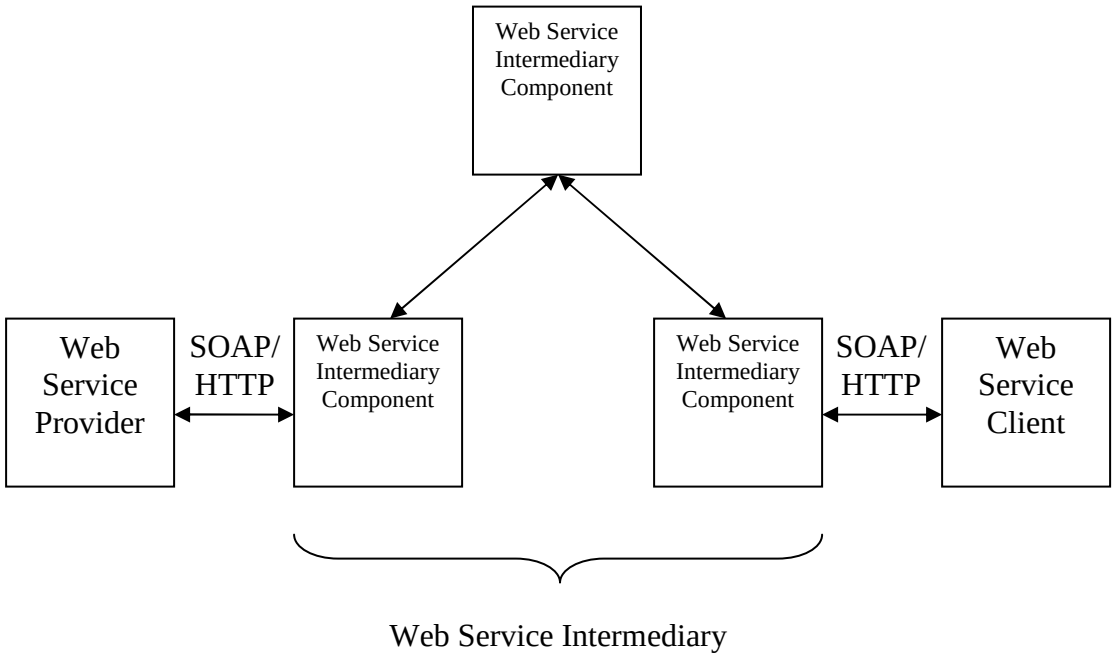

Figure 4.4: A Web Service intermediary composed of more than one component

4.3.3 How does a Web Service Intermediary Solve the Problem?

A Web Service intermediary takes the responsibility of implementing the quality-of-service requirements in the communication between a Web Service provider and a client. This is accomplished as follows:

The only channel that the provider and the client can directly communicate on is the Internet which does not meet the quality-of-service requirements. Instead of communicating directly they communicate through a Web Service intermediary. The provider and the client of a Web Service each interact with a local component of the Web Service intermediary. This interaction takes place over a local communication channel that does meet the quality-of-service requirements.

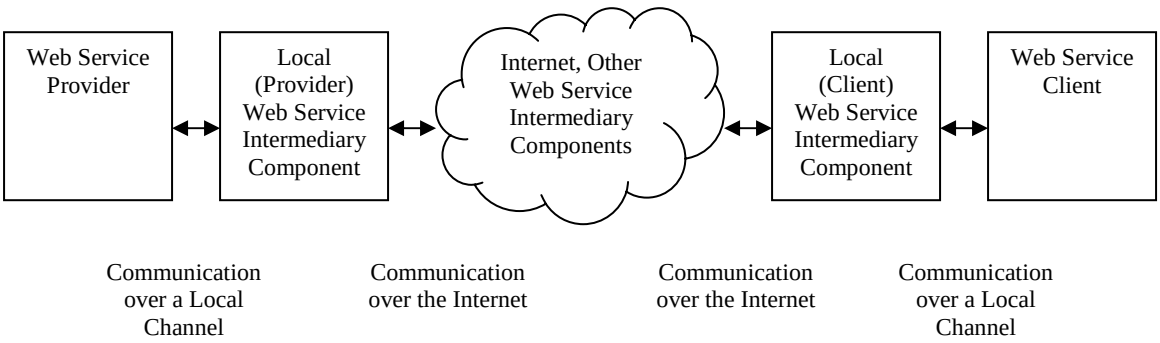

Figure 4.5: General architecture of a Web Service intermediary

The component of the Web Service intermediary local to the provider and the component local to the client interact over the Internet to relay the communication between the Web Service provider and the client. Their interaction may go through other components of the Web Service intermediary. The interaction takes place over the Internet which as we know does not generally provide quality-of-service guarantees. However, the components of the Web Service intermediary take measures to make their interaction over the Internet meet the quality-of-service requirements. These measures

include using public key cryptography for security, message queues for reliable delivery, and use of ordered message delivery and exactly-once message delivery algorithms etc.

This way the Web Service provider and the client do not have to be concerned about implementing the quality-of-service requirements. They can rather concentrate on the functionality of the Web Service. The Web Service intermediary enforces the required quality-of-service for them.

This process will become more vivid when we discuss actual Web Service intermediary implementations in the upcoming sections of this and the next chapter.

4.3.4 Some Implementations Based on the Concept of Web Service Intermediary

Grand Central Communications [GrandCentral2002gc] and Flamenco Networks [Flamenco2002flamenco] are two notable start-up companies, each of which offers its own implementation of a Web Service intermediary [Tominaga2002gcflamenco]. Although, both products are based on the same concept, there are some differences in the internal architecture of the intermediaries. Here is an overview of the two products.

4.3.4.1 Grand Central Communications' Web Services Network

4.3.4.1.1 Architecture and Operation

Grand Central's Web Services Network is based on a hub-and-spoke model [Jones2002enterpriseclassws]. Grand Central hosts a central server at its site. All communicating parties must go through this server to exchange messages. The Web Service and the client each need a custom wrapper to interact with the Grand Central

server. The wrappers can be written using a Software Development Kit (SDK) provided by Grand Central. Communication between the wrappers and the Grand Central server takes place over the Internet. The wrappers collaborate with the server to implement the required quality-of-service.

An enterprise network administrator's interface to the Grand Central Web Services Network is through a web site hosted by Grand Central. The web site allows the administrator to configure the level of quality-of-service desired. The web site is also responsible for providing activity reports and other management facilities.

To use Grand Central's Web Services Network, each enterprise must register with Grand Central. Registration requires an initial setup fee and then a monthly fee for as long as the service is used.

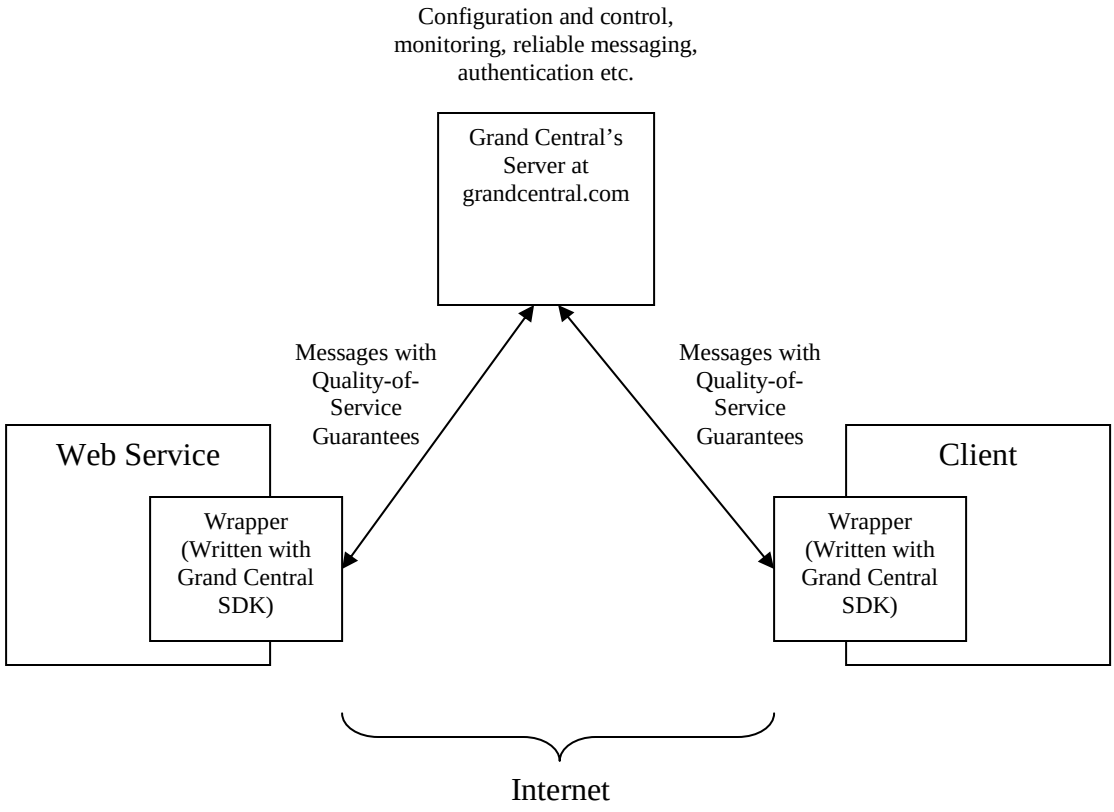

Figure 4.6: Architecture of Grand Central Communications' Web Services Network

4.3.4.1.2 Services Provided

Grand Central's Web Services Network provides the following services [Grandcentral2002gcservice]:

- Flexible connection – Enterprises have the option to communicate with the Grand Central Server using SOAP over FTP, SMTP or HTTP.
- Security – Security services include authentication, encryption and access control. Authentication is provided using the VeriSign-based [VeriSign2002verisign] authentication system. Connections can be secured with up to 128-bit encryption. 128-bit encryption is considered satisfactory by today's standards.

- Reliable messaging – Messages are queued for robust delivery. Messages can also be delivered using a store-and-forward model.
- Partner management – Separate profiles are provided for each client of a Web Service.
- Monitoring – Activity reports and exception alerts are provided.

4.3.4.1.3 Pros and Cons

An advantage of Grand Central's Web Services Network is that, since all communication goes through a central server, it is able to provide very comprehensive monitoring services. However, this product also has some disadvantages:

1. An enterprise willing to use Grand Central's Web Services Network, has to first develop a wrapper for its Web Service or Client for it to be able to communicate with it. Developing this wrapper may not be too difficult, but it still poses a hurdle and prevents immediate deployment.
2. Another drawback is due to its central server architecture. The central server makes all communication completely dependent on Grand Central. If the server ever ceases to function, the whole communication would fail. For instance, Grand Central could stop offering this product in the future and shutdown the server. The central server places control of the flow and management of information in the hands of a third party which may not be desirable. A central server also limits the scalability of a Web Service application.

4.3.4.2 Flamenco Networks' Web Services Network

4.3.4.2.1 Architecture and Operation

Flamenco's Web Services Network is a hybrid of client-server and peer-to-peer models. A key component of Flamenco's Web Services Network is the Flamenco proxy. A Flamenco proxy is installed on each communicating party's trusted network. The Web Service and the client communicate only with the proxy installed on their own trusted network. It is presumed that this internal communication already meets the enterprise's quality-of-service requirements. This is generally true because the proxy runs either on the same machine as the Web Service/Client, or on a machine connected by a reliable Local Area Network (LAN). The Web Service and the client do not directly interact with any other entity.

When a proxy receives a message from the local application inside the trusted network, it forwards it over the Internet to the proxy of the destination application. That proxy on receiving this message then delivers it to the destination application residing on its trusted network. The proxies interact in a peer-to-peer manner to exchange messages. The proxies are responsible for implementing the required quality-of-service in their communication.

The communication however also involves a central server hosted at Flamenco Networks' site. Each of the peer proxies is managed and controlled from the central server. The proxies send activity reports to the server, which are then compiled and made available to the user from the central server. The central server also functions as a digital

certificate authority for authentication purposes. The server plays an indispensable role in Flamenco's Web Services Network.

Like Grand Central, an enterprise network administrator's interface to the Flamenco Web Services Network is through a web site hosted by Flamenco Networks. The Flamenco proxies can be managed only through the Flamenco Networks web site. Flamenco also requires registration with an initial setup fee and a continuous monthly fee thereafter.

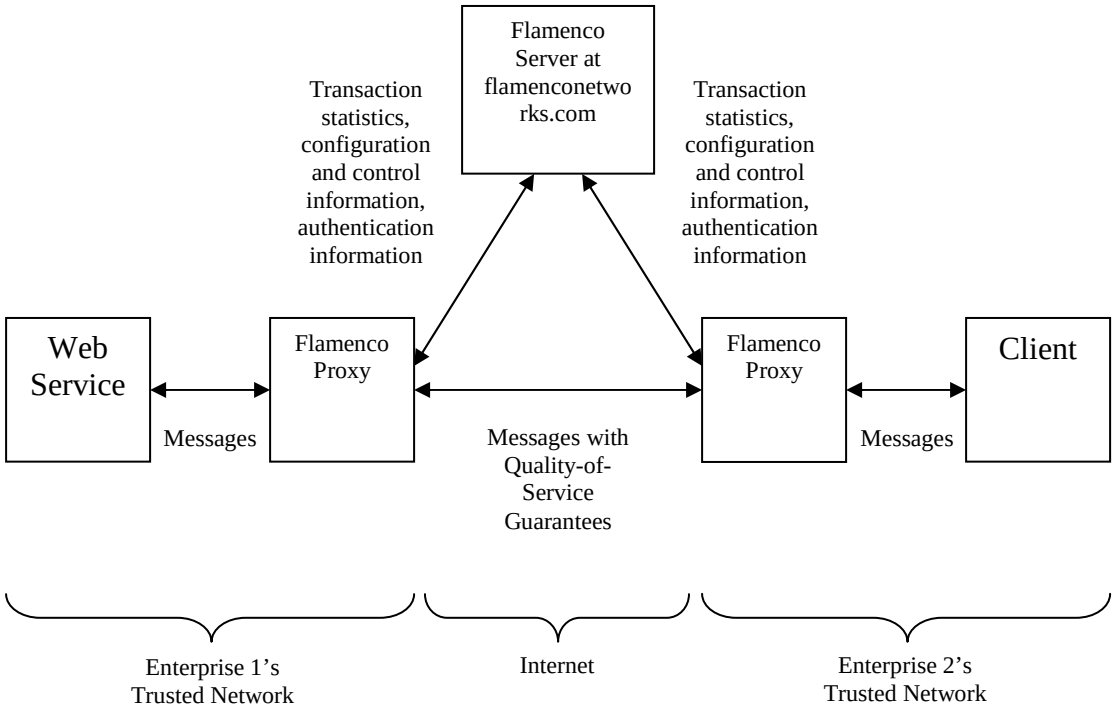

Figure 4.7: Architecture of Flamenco Networks' Web Services Network

4.3.4.2.2 Services Provided

Flamenco's Web Services Network provides the following services [Flamenco2002wsnetworks]:

- Security – Flamenco's security services include authentication and confidentiality. Flamenco acts as its own digital certificate authority for authentication purposes. The proxy-to-proxy and proxy-to-server communication can be encrypted for confidentiality.
- Reliability – The messaging is made reliable by providing guaranteed delivery, non-repudiation and once-and-once only delivery.
- Manageability – Message tracking, activity monitoring and logging features are available for improved manageability.
- Access management – Enterprises can control the level of a Web Service's functionality that is accessible by clients.

4.3.4.2.3 Pros and Cons

Flamenco Networks' Web Services Network has the advantage of being immediately deployable. An enterprise needs only to install the Flamenco proxy, configure it for the desired quality-of-service and start exchanging messages. There is no development required. However, this product is not free of all problems either. It shares Grand Central's drawback number 2 (section 4.3.4.1.3) due to its use of a central server. Third-party dependence is not desirable for an enterprise.

One could argue that the problem is not in the architecture and that the server may as well be hosted by one of the two communicating enterprises instead of a third-party. However, this argument does not hold completely valid due to the following reasons:

- This new configuration would only partially solve the problem. It is true that the enterprise that gets to host the server would benefit and would be in total control. The other enterprise though, would still remain deprived of the ability to independently control its quality-of-service requirements. It will also have to depend on the host enterprise for all of its activity monitoring and other management needs.
- Moreover in the Web Services architecture a Web Service application can be a client as well as a server. If two peer-to-peer Web Service applications want to communicate there is no basis to choose one of them over the other to host the central server.

4.4 A Review of the Discussed Approaches

The first approach, to update existing Web Services protocols, is currently under debate. It would take considerable research to establish if it can or cannot be a viable solution.

The second approach, to hard code the desired quality-of-service, has obvious defects. It is time and effort consuming and limits the integration and interoperability capabilities of a Web Service implementation.

The third approach, to use a Web Service intermediary, offers many benefits and doesn't have any significant disadvantages intrinsically. However, how this approach is implemented determines the actual advantages and disadvantages that can be derived from it. We have discussed two implementations of this approach. They both offer important benefits but unfortunately also suffer from serious drawbacks. This is partly due to the central server architecture that they both use.

We can conclude from this discussion that although the Web Service intermediary approach is a good solution to our problem, current implementations of this concept have

drawbacks associated with them. An implementation of the Web Service intermediary approach that exploits its benefits but does not have the disadvantageous side effects is desired.

Chapter 5: Our Solution – Peer-to-Peer Intermediary

5.1 Objectives

In the previous chapter we established that the Web Service intermediary approach is an appropriate solution to the given problem. However, as discussed, the current implementations of this approach suffer from several drawbacks. Our goal is to design an implementation of the Web Service intermediary approach which is free from these drawbacks. The characteristics that we aim for in our solution are as follows:

- It should not require additional coding for each deployment
- It should not place control of the information flow and its management in the hands of a single party (especially not a third-party)
- It should be easily and quickly deployable
- It should not limit the scalability of the Web Service

The following subsections explain why we aim for these characteristics in our solution.

5.1.1 Requirement of Customization through Additional Coding

Customization/Modification of software is a time and effort consuming process that requires skilled software engineers who thoroughly understand the original design of the software. David L. Parnas, one of the most renowned experts in the field of Software Engineering, highlights the risks involved in modification of software in general and especially by people who do not thoroughly understand the original design of the software as follows [Hoffman2001parnas] – section 29.3.2:

> "Changes made by people who do not understand the original design concept almost always cause the structure of the program to degrade. Under those circumstances, changes will be inconsistent with the original concept; in fact, they will invalidate the original concept. Sometimes the damage is small, but often it is quite severe. After those changes, one must know both the original design rules, and the newly

> introduced exceptions to the rules, to understand the product. Those who made the changes, never did. In other words, nobody understands the modified product. Software that has been repeatedly modified (maintained) in this way becomes very expensive to update. Changes take longer and are more likely to introduce new "bugs"."

We can infer that unless modification to software is necessary, it should be avoided.

5.1.2 Enterprise's Control of the Flow and Management of its Information

In some cases enterprises do not wish to rely on third-parties for their operation. Reasons include concerns about confidentiality and dependability.

The following quote from a recent article titled "Third-Party Dependence – A Potential Disaster Domino" highlights a risk involved in third-party dependence [Keating2002thirdparty]:

> "As suppliers play a more critical role in business continuity planning (BCP), many companies employ single and sole-source supplier agreements because of the operational and financial benefits of dealing with one company. Unfortunately these agreements bring the risk that suppliers' contingent business interruptions could close internal operations. Two recent incidents resulted in losses of $175 and $400 million, according to The Wall Street Journal."

Although the article refers to the manufacturing industry and sole dependence on a single third-party, the same is also true for any enterprise depending on one or more than one third-parties for the flow and management of its information.

We can infer that third-party dependence should not be a mandatory requirement in a Web Service intermediary.

Moreover consider Web Service applications that are based on a peer-to-peer architecture. Giving one of the applications (or owner enterprise of that application) more

control of the flow and management of the communication is not conformant with the peer-to-peer model.

5.1.3 Ease and Speed of Deployment

[Schwartz2002deployability] makes the following observation about deployability of business systems:

> "In order for businesses to be competitive, the systems that facilitate operations and create value need to be deployed quickly and predictably into many different environments, over several channels, at very low cost – all while maintaining service levels and security requirements.
> These requirements make business system deployability a key factor in the overall success of a business system."

Easy and quick deployment of business systems saves money and time for businesses. The saved money results in higher profits. Saved time is valuable for getting ahead or at par with the competition.

5.1.4 Scalability

[Coulouris2001distributed], one of the most definitive texts on Distributed Systems, describes scalability as one of the key characteristics that the designers of a distributed system should aim for. [Coulouris2001distributed] can be quoted as follows:

> "A distributed system is scalable if the cost of adding a user is a constant amount in terms of the resources that must be added. The algorithms used to access shared data should avoid performance bottlenecks and data should be structured hierarchically to get the best access times."

The demand on the resources of a distributed system generally grows with time. Scalability ensures that the distributed system is able to meet the growth in demand.

An example of a scalable distributed system is the Internet. The Internet started out with a few computers but it gradually scaled to include millions of computers. Scalability is one of the primary reasons for the success of the Internet.

5.2 Design

With these objectives in mind, we have designed a peer-to-peer implementation of the Web Service intermediary approach. This solution is termed **P**eer-to-peer **I**ntermediary, abbreviated as PI (pronounced as π). In this chapter we discuss a high level design of the solution. Implementation level details are discussed in the next chapter.

5.2.1 Architecture

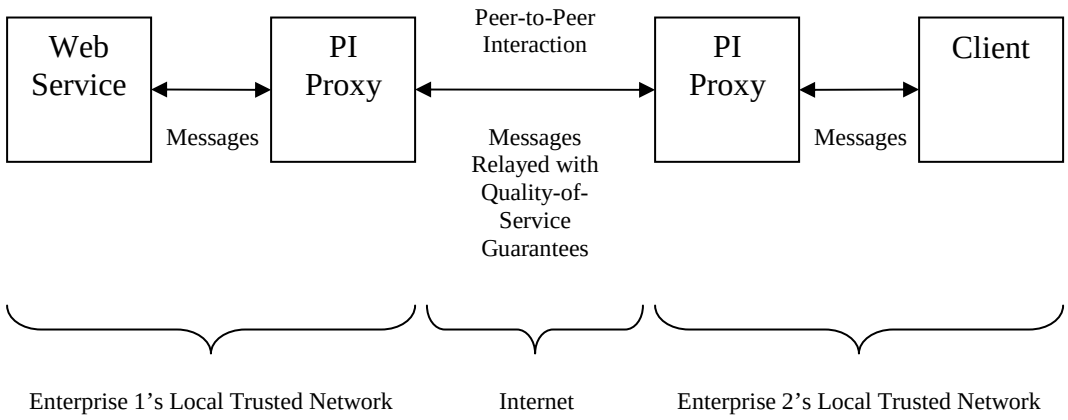

Figure 5.1: PI architecture

5.2.1.1 Components

The components of the PI architecture are the Web Service, the client and a pair of PI proxies. The Web Service and the client are standard components from the Web Services architecture. The PI proxies are the additional components in the PI architecture.

A PI proxy is a software program whose basic function is to relay messages between its local application (Web Service/client) and its peer PI proxy. For each Web

Services component (Web Service/client), a copy of the PI proxy is installed on its local trusted network. The Web Service and the client each interact only with its local PI proxy. It is presumed that this internal interaction already meets the enterprise's quality-of-service requirements. This is generally true because the PI proxy runs either on the same machine as the Web Service/client, or on a machine connected by a reliable Local Area Network (LAN). Trusted networks are described more thoroughly in section 5.2.1.2.1. The two peer PI proxies interact over the unreliable and insecure Internet but take measures to make this interaction conform to the quality-of-service requirements. These measures are described in section 5.4.

5.2.1.2 Connectors

Three connectors connect the components described above. These connectors are the Web Service's local trusted network, client's local trusted network and the Internet. The Web Service and the client interact with their respective PI proxies only over their own local trusted networks. As discussed, these local networks meet their enterprise's quality-of-service requirements. The PI proxies interact over the public Internet in peer-to-peer fashion. The Internet is unreliable and insecure therefore quality-of-service is not automatically guaranteed. The proxies, however, take measures to implement the quality-of-service requirements in their communication. An intranet can also be used instead of the Internet as the connecter between the PI proxies.

5.2.1.2.1 Trusted Network

We define a trusted network as a computer environment which provides a communication channel between a Web Service application and a PI proxy that meets the quality-of-service requirements of the owner enterprise. The quality-of-service requirements that we consider here are reliability and security from external threats. These quality-of-service requirements are discussed in section 3.1.

A trusted network in the simplest form is a standalone machine. A Web Service application and a PI proxy running on that machine can reliably and securely communicate with each other. All communication is internal to the machine therefore there is no risk of reliability problems that are generally introduced by communication

over a network. The communication is secure since attackers have no access to the machine because it is not connected to any network. We do not consider physical threats.

Standalone machines are however not useful in the PI architecture. The PI architecture requires that trusted networks be connected to the Internet for the PI proxies to be able to communicate with each other. Connecting a machine to the Internet does not affect the reliability of its internal communication but does render it vulnerable to external security threats. To make the machine secure, a standard solution such as a firewall can be used.

A trusted network can also be composed of multiple machines connected to each other by any kind of network as long as the quality-of-service requirements are met. In this case the Web Service application and the PI proxy can be running on separate machines. Popular LANs such as Ethernet 100BaseT, 100BaseF, 1000BaseT and FDDI provide reliable communication if designed and administrated properly. Detailed discussion of these and other LAN standards can be found in [LeonGarcia2000networks]. The network can be secured from external security threats by using a firewall at all points of Internet access.

Communication on a trusted network is not safe from internal security threats including physical threats. The strength of security from external threats depends on the effectiveness of the firewalls used.

5.2.1.3 Multiple Clients Scenario

The PI architecture allows a Web Service to serve multiple clients. For each client there is an exclusive PI proxy on the client's trusted network. The PI proxy for the Web Service can be configured to serve multiple clients. Each client's PI proxy interacts with the Web Service's PI proxy in the same manner as it would in a single client scenario.

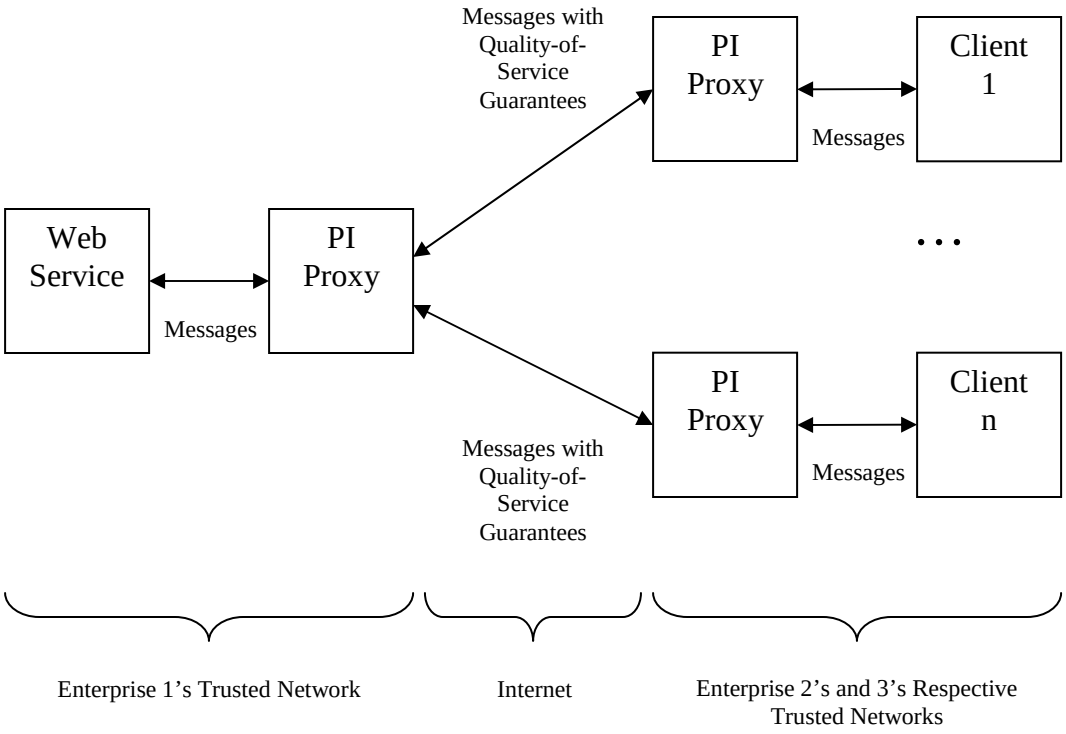

Figure 5.2: Multiple clients – configuration 1

Another possibility is to have multiple PI proxies for the Web Service. Each of the Web Service's PI proxies can be configured to serve a single or multiple clients.

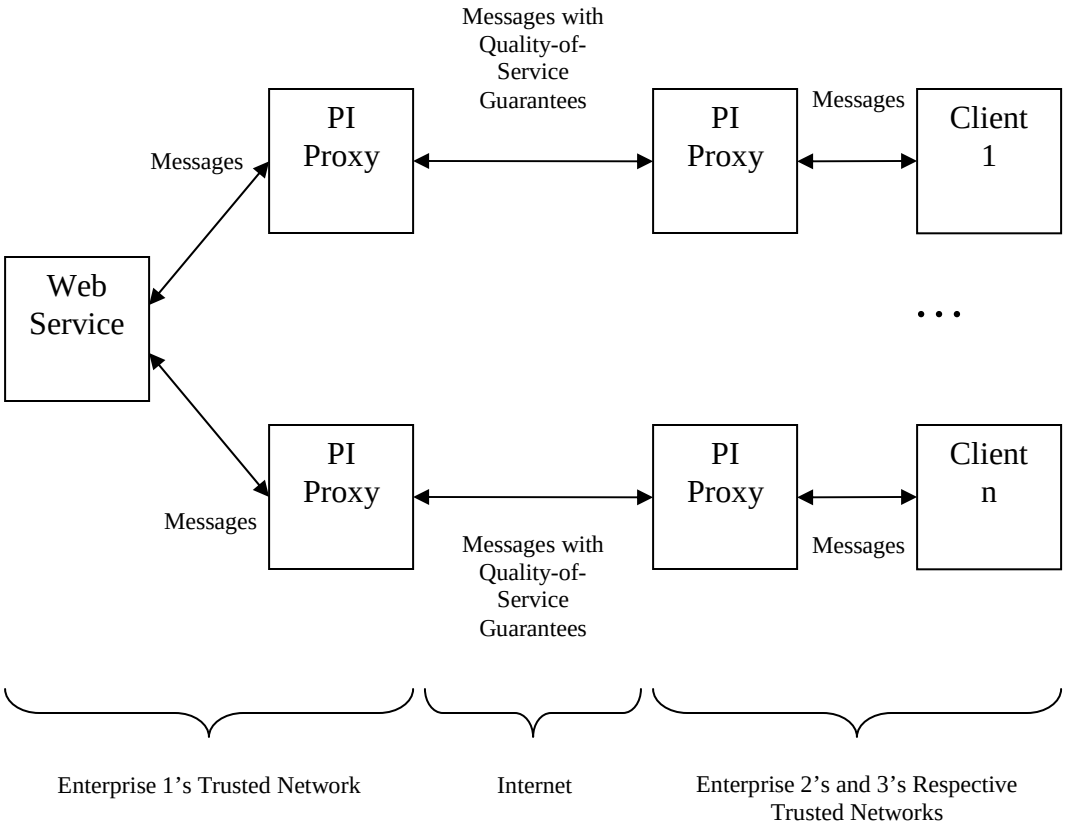

Figure 5.3: Multiple clients – configuration 2

5.2.2 Message Exchange Model

The basic function of a PI proxy is to relay messages between its local application (Web Service/client) and its peer PI proxy. To perform this function, each PI proxy is provided with the location of the local application and the location of each peer PI proxy that the PI proxy will interact with. A unique ID is assigned to each of these peer PI proxies.

The PI proxy relays messages in two directions. It receives incoming messages from peer PI proxies which are relayed to the local application. It receives outgoing messages from the local application which are forwarded to the appropriate peer PI proxy.

Since there is only one local application, all incoming messages are simply delivered to the location of the local application as configured in the PI proxy. However,

there can be multiple peer PI proxies that a PI proxy interacts with, so the process of relaying outgoing messages can be more involved. We can illustrate this process with the help of an example. In the case where there is no Web Service intermediary, a Web Service sends a message directly to client's location/address for example, http://clientsaddress/. In the case where PI is being used, the Web Service would send the message to its local PI proxy, for example, on the address http://piproxyaddress/. However, to be able to forward this message to the appropriate peer PI proxy, the PI proxy would need the local application to identify that peer PI proxy. For this purpose the unique ID assigned to each peer PI proxy is used. The local application includes this ID as an argument in the address, for example if the message is intended for a peer PI proxy with the unique ID 'blue', it would send the message to the address http://piproxyaddress/blue. The PI proxy upon receiving the message would forward it to the peer PI proxy identified by the ID 'blue'.

The steps involved in delivering a message from one application (Web Service/client) to the other using PI are as follows:

1. The sender application sends the message to the local PI proxy.
2. The PI proxy receives the message, determines the recipient peer PI proxy and forwards the message to it.
3. The recipient peer PI proxy upon receipt of the message delivers it to its local application which is the final destination of the message.

The communication between the peer PI proxies is in peer-to-peer fashion. The peer PI proxies can take measures to communicate this message in a manner that conforms to the quality-of-service requirements. The measures taken to achieve this are described in section 5.4. Since the two PI proxies are independently configured, the stricter of the two quality-of-service requirements is selected.

5.3 How does PI Meet the Objectives?

The following comparison between existing implementations of the Web Service intermediary approach (based on central-server architecture) and PI (based on peer-to-

peer architecture) illustrates that PI does not have the problems associated with the former. It is also demonstrated that PI meets the objectives that we stated in section 5.1.

5.3.1 Requirement of Customization through Additional Coding

5.3.1.1 Grand Central (Central-Server Architecture)

Grand Central requires each Web Service application to be customized by hard coding for it to be able to use its Web Service intermediary. For this purpose, Grand Central provides SDKs for Java, Microsoft .NET and Perl. Grand Central can be quoted as follows [GrandCentral2002gc]:

> "Using the methods offered by the Grand Central Communications SDK, a client application needs to be built or an existing one extended to exchange information with the other Web service over Grand Central Communications."

Grand Central's SDKs are proprietary and not openly available. The SDKs are available only to the customers of Grand Central. We therefore could not acquire:

- Any experience as to how easy or hard it is to customize a Web Service application with these SDKs.
- An estimate of how much time the customization process takes.

The following question and answer from the Grand Central Communications Frequently Asked Questions (FAQ) [GrandCentral2002faq] indicates that the process may take a few days:

> "Is Grand Central Communications a software solution? – No, Grand Central is delivered as a subscription service that has a low implementation cost and can be deployed in days rather than months."

There are several disadvantages of customization through reprogramming, which include the following:

- Reprogramming of the application can have negative side effects on it such as introduction of new bugs.
- A programming project is time and effort consuming and also expensive.
- Many enterprises do not have the expertise to undertake a programming project.

5.3.1.2 PI (Peer-to-Peer Architecture)

PI does not require a Web Service application to be customized through hard coding. The only change PI requires is that the Web Service application communicate with the local PI proxy instead of the partner Web Service application. This should be a matter of changing a field in the configuration or in an invocation file if the application has been developed using typical good programming practices.

The PI architecture separates the Web Service application and the local part of the Web Service intermediary into two separate components. Instead of being joined together, they run separately and communicate with each other over a local trusted network.

This design eliminates the need for customization through hard coding.

5.3.2 Enterprise's Control of the Flow and Management of its Information

5.3.2.1 Flamenco, Grand Central (Central-Server Architecture)

In case of both Flamenco and Grand Central Web Service intermediaries, the central server for the intermediary is hosted by its vendor. This means that enterprises using their products have to completely rely on them for the flow and management of their information. If problems occur, an enterprise can do absolutely nothing but wait for the vendor to fix them. An extreme scenario would be the vendor going out of business and shutting down its services, in which case an enterprise's information infrastructure would completely break down. Moreover, the clients of a Web Service may not even want to relay their information through a third party due to confidentiality concerns. For example the third party would have knowledge such as the business partners of each enterprise, its terms of communication with them, and time and type of transactions with those partners.

Much of this knowledge is gained by the third party even if all communication is encrypted.

Even if the server of a central-server based Web Service intermediary is hosted by one of the communicating enterprises, it places all control with that one enterprise and the other enterprise still lacks equivalent control. For example, since one of the enterprises is in control, it can dictate that messages be sent unencrypted, the other enterprise has to comply. If a reverse situation arises, the enterprise still has to accept the terms set by the enterprise in control. The only other option is to cease communication with that enterprise.

Moreover in the Web Services architecture a Web Service application can be a client as well as a server. If two peer-to-peer Web Service applications want to communicate there is no basis to choose one of them over the other to host the central server.

One could reason that the benefit of a third party providing the Web Service intermediary is that the third party would completely take care of deployment and maintenance of the Web Service intermediary for the enterprises. We agree that this would definitely be a major benefit perhaps worthy of the fee the third party would charge for doing so. However, all vendors that we know of including Flamenco and Grand Central require the enterprises to setup and maintain the enterprise side components of the Web Service intermediary by themselves.

The vendors do provide some help and technical support to the enterprises in setting up and maintaining their side of the Web Service intermediary infrastructure. This is not as advantageous as a complete solution, but it still is a benefit.

5.3.2.2 PI (Peer-to-Peer Architecture)

This problem is resolved in PI primarily due to its peer-to-peer architecture. Each enterprise has a PI proxy hosted on its own trusted network for each of its Web Service applications. Each enterprise is in charge of its information infrastructure since all its components are in its own possession.

Since each enterprise is in full control of its PI proxy, it can define its own quality-of-service requirements. When two PI proxies communicate, they negotiate as to

the strictness of quality-of-service that they will implement. Therefore both enterprises are given fair treatment. For example we can take the scenario in which a Web Service wants to receive a message in unencrypted form but the client wants to send it in encrypted form. Their two peer PI proxies will negotiate with each other and since the client's requirement is stricter, that will be selected. If a reverse situation ever arises, the Web Service's stricter requirement would then be selected. This shows that the system is not biased towards any single party and everyone is treated equally.

It is to be noted that even though our approach provides fair treatment to each party, it is possible that an agreement between two parties may not be reached. For example, if a PI proxy is configured to require encryption but the peer PI proxy does not have the resources to support encryption, the negotiation will fail and the message will not be exchanged.

It is not mandatory to limit the condition of agreement to the strictest quality-of-service requirements of one of the two proxies. It is also possible to have a compromise between the quality-of-service requirements of the two proxies if both proxies are willing to accept it. This possibility is further discussed in section 7.3.

The PI infrastructure has to be maintained by an enterprise completely on its own. However, we believe that this should not be a major issue. The enterprise would already be administering a Web Service application and a firewall so it would have enough expertise to administer a PI proxy. Provision of good documentation by the implementers of PI proxy is essential for this purpose.

5.3.3 Ease and Speed of Deployment

5.3.3.1 Grand Central (Central-Server Architecture)

As discussed in section 5.3.1.1, Grand Central requires each Web Service application to be customized by extra coding for it to be able to use its Web Service intermediary. This is a time and effort consuming process. Moreover, programming skills are needed to undertake this process. This requirement restricts this kind of Web Service intermediaries from being quickly and easily deployed.

Following are the steps required from an enterprise that wants to use the Grand Central intermediary for its Web Service:

1. Development – The enterprise will have to assemble a team of developers who have the programming skills and understanding of the Web Service to customize it according to Grand Central's requirements. The team will develop and test the customized Web Service.
2. Installation – The administrator of the Web Service will then host the Web Service on the enterprise's network and will establish a connection with the Grand Central intermediary over the Internet.
3. Configuration – The administrator will then configure the intermediary for each partner that the enterprise wishes to communicate with.

The total time required to complete these steps can expressed as follows:

Total time = Development time + Installation time + Configuration time

5.3.3.2 PI (Peer-to-Peer Architecture)

The PI deployment process does not require coding. Following are the steps required from an enterprise that wants to use PI for its Web Service:

1. Installation – The administrator of the Web Service will install a PI proxy on the enterprise's network and will make Internet access available.
2. Configuration – The administrator will then configure the intermediary for each partner that the enterprise wishes to communicate with.

The total time required to complete these steps can expressed as follows:

Total time = Installation time + Configuration time

Since installation and configuration procedures of the Grand Central Web Service Intermediary and PI are similar in complexity, they should both take approximately the same amount of time.

PI eliminates the development step therefore its deployment is quicker and easier.

5.3.4 Scalability

5.3.4.1 Flamenco, Grand Central (Central-Server Architecture)

We analyze the scalability of a Web Service intermediary in terms of the number of connections to Web Service applications that it has to handle at any given time.

A Web Service application can be both a server and a client at the same time. However, to simplify our analysis we assume that a Web Service application can be either a server or a client but not both. This assumption does not affect our final inference.

Consider that a Web Service application server has n clients that communicate with it through a central-server-based Web Service intermediary. This configuration is depicted in the following figure.

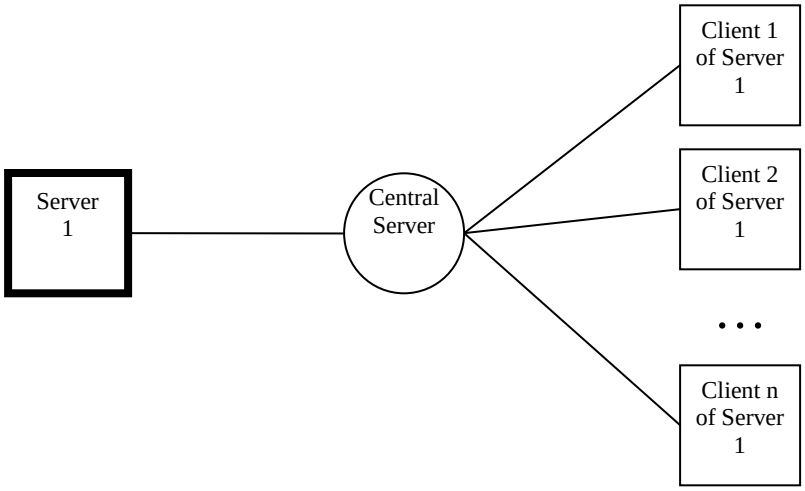

Figure 5.4: A Web Service application server and its n clients using a central-server-based Web Service intermediary

According to central-server fundamentals and as is done in practice, the central-server-based Web Service intermediary serves multiple Web Service application servers. This scenario is shown in the following figure where the central-server-based Web Service intermediary serves m Web Service application servers, each of which has n clients.

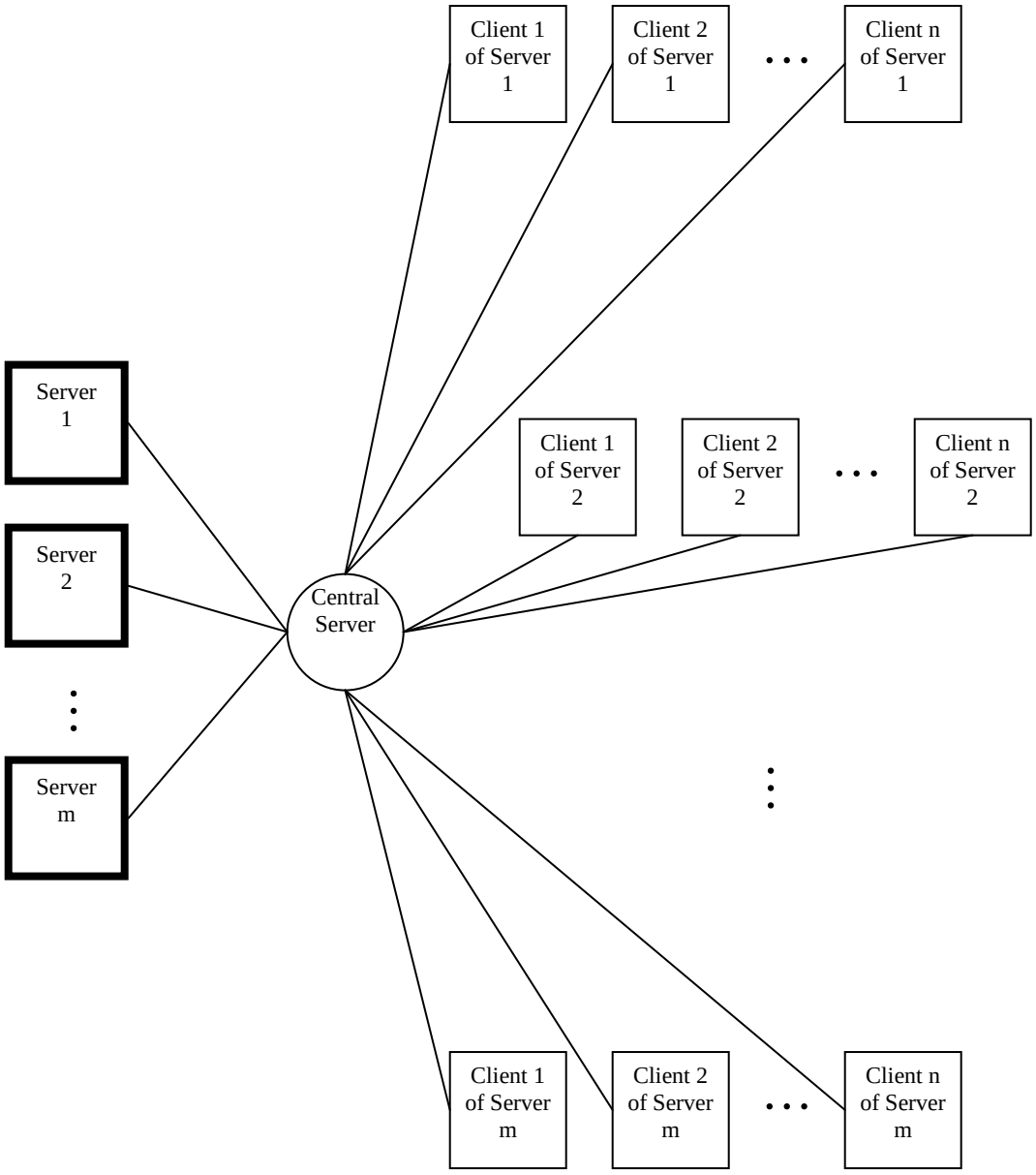

Figure 5.5: m Web Service application servers and n clients of each one of them using a central-server-based Web Service intermediary

We consider n as the maximum number of clients that a Web Service application server communicates with. A Web Service application server may have fewer than n clients.

The maximum number of connections that the central-server-based Web Service intermediary would have to handle in the given scenario is $m \times (n + 1)$, which is $O(n^2)$.

It should be noted that the scenario discussed above is a very simplified case. We do not consider many factors including: the possibility of a Web Service application being both a server and a client at the same time, a Web Service application being a client of more than one Web Service application server, use of multiple servers in the central-server-based Web Service intermediary, the number/size of the actual messages passed over the connections etc. Each of these factors can affect the scalability of the Web Service intermediary. Nevertheless the scenario presented is realistic and can occur frequently in practice.

5.3.4.2 PI (Peer-to-Peer Architecture)

We consider the same scenario for PI as we did for central-server-based Web Service intermediaries in the previous section.

A Web Service application server has n clients that communicate with it through PI proxies. The configuration is shown in the following figure.

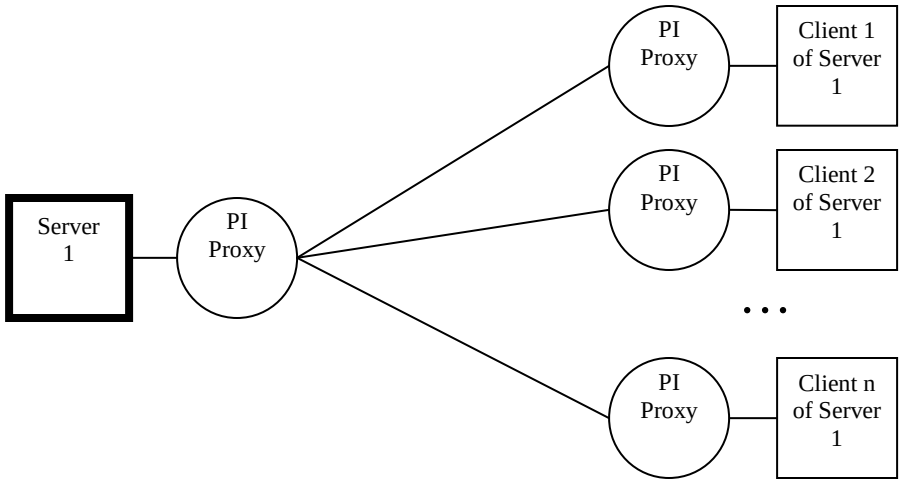

Figure 5.6: A Web Service application server and its n clients using PI

Now consider the scenario in which there are m Web Service application servers and each one of them has n clients. A Web Service application server from its PI proxy's perspective is the local Web Service application. One PI proxy exclusively serves only one local Web Service application. Therefore each of the m Web Service application servers would have a separate PI proxy of its own. The scenario is illustrated in the following figure.

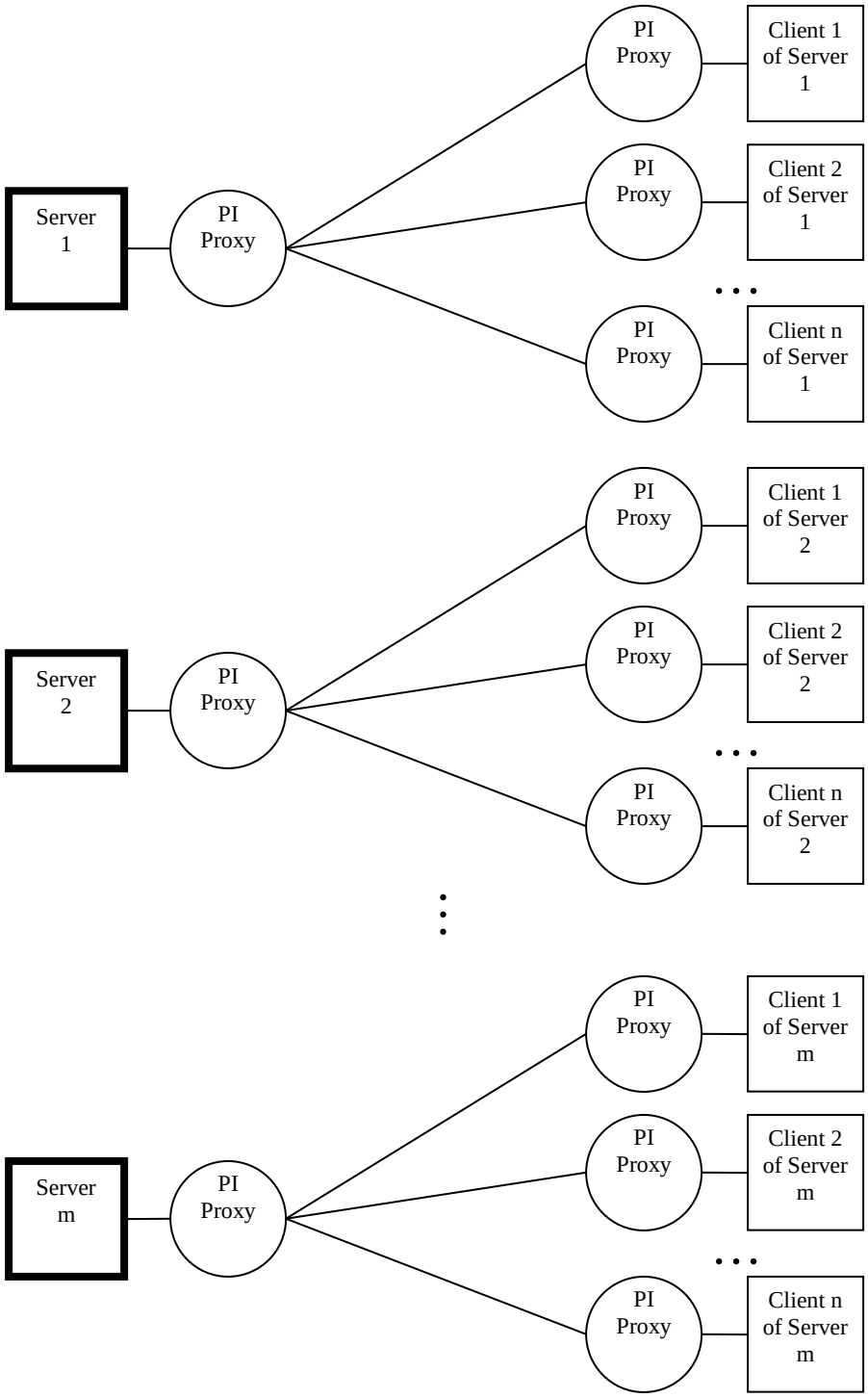

Figure 5.7: m Web Service application servers and n clients of each one of them using PI

The maximum number of connections that any PI proxy has to handle is $n + 1$, which is $O(n)$.

$O(n)$ connections are far less resource-consuming than $O(n^2)$ connections and therefore PI is more scalable than the central-server-based intermediaries.

It should be noted that the $O(n)$ bound also applies to the scenario where a Web Service application can be the client of more than one Web Service application server. This is not true for a central-server-based Web Service intermediary's $O(n^2)$ bound, which would actually increase in this scenario. The following figure shows PI in such a scenario.

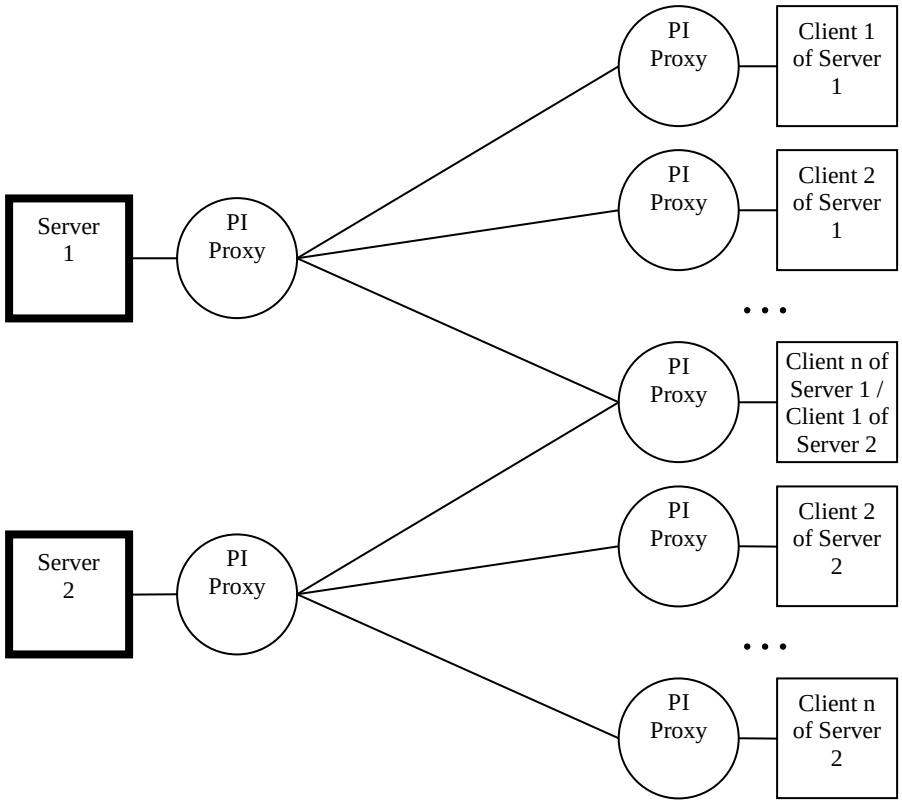

Figure 5.8: A Web Service application client using PI to communicate with more than one Web Service application server

We have presented a very simplistic analysis of the scalability of central-server-based intermediaries and PI. As we mention in section 7.3, an in-depth analysis should be conducted for more thorough results.

5.4 Quality-of-Service Features Provided to Web Services by PI

PI is designed to accommodate all the quality-of-service requirements mentioned in section 3.1. How each of these requirements is met is discussed from a design perspective in the following subsections. All these services are customizable both at the Web Service and the client end. The measures taken by the PI proxies to achieve the quality-of-service requirements are invisible to the Web Service and the client.

5.4.1 Security

PI uses public key cryptography for providing authentication, confidentiality and non-repudiation. The use of public key cryptography for providing security is a universally accepted practice [iDTwo2002pki]. A short introduction to public key cryptography is as follows:

The public key cryptography system has the following six elements:

Plain text – `P`, which is the message to be encrypted.

Cipher text – `C`, which is the encrypted message.

Public key – `X`, of an enterprise `E`.
Private key – `Y`, of an enterprise `E`. } Key Pair

Encryption function, `C = e(P, K)`, where `K = X` or `K = Y`

Decryption function, `P = d(C, K)`, where `K = X` or `K = Y`

The keys and the functions are such that `C = e(P, X)` can only be decrypted as `P = d(C, Y)`. Conversely `C = e(P, Y)` can only be decrypted as `P = d(C, X)`. This means that a message encrypted with one key in the key pair can only be decrypted by the other key in the pair.

Public key cryptography is discussed in detail in [Buchmann2001crypt].

The algorithms given in the following subsections are standard algorithms for providing security. Further detail on these algorithms can be found in standard texts on computer and network security such as: [Buchmann2001crypt], [Coulouris2001distributed] – chapter 7, and [LeonGarcia2000networks] – chapter 11.

5.4.1.1 Authentication

Authentication is provided by using public key cryptography and digital certificates. An enterprise E can be authenticated by receiving from it a message which is encrypted with its (E's) private key. If the message can be decrypted with E's public key, it is proof that the enterprise is whom it claims to be. However, since the public key is public, we cannot be sure if it is actually the public key of E and not a key planted by an attacker for malicious reasons. This is where digital certificates are useful. A digital certificate is a document which states the public key of an enterprise and the document itself is encrypted by the private key of an entity called a certificate authority. A certificate authority is an entity which is widely trusted and whose public key is well known. Therefore it is safe to use the public key of an enterprise provided in a digital certificate by a trusted certificate authority. Digital certificates are discussed in detail in [Buchmann2001crypt].

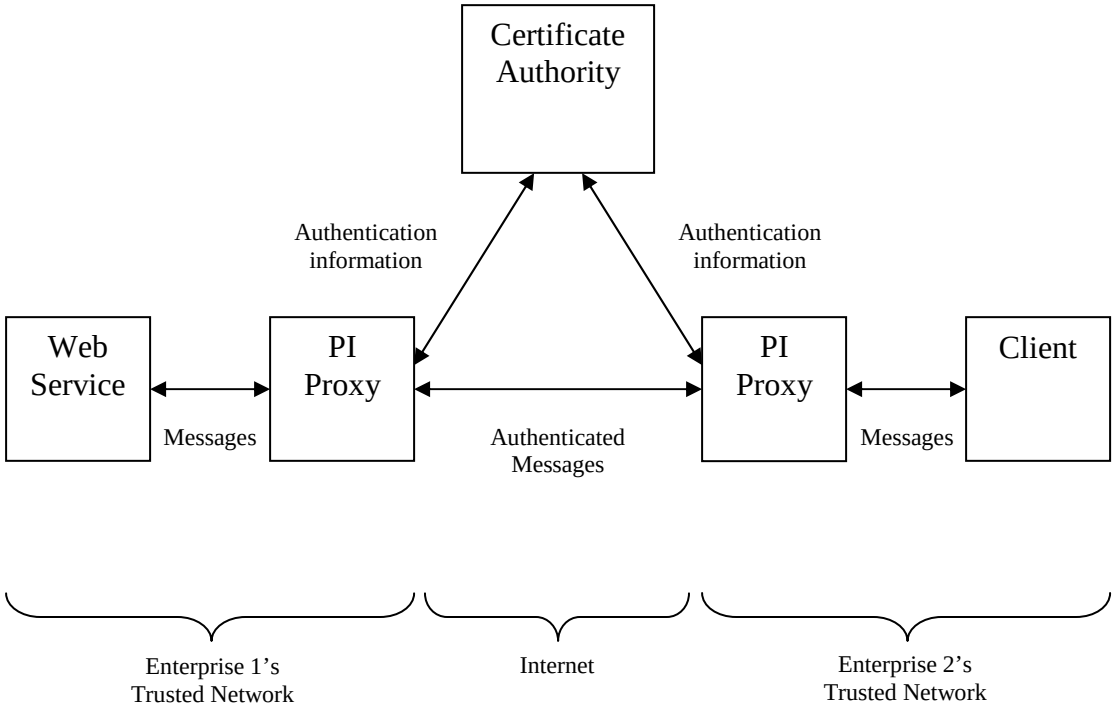

Figure 5.9: PI authentication model

PI requires that each enterprise which would need to be authenticated by another enterprise must have a digital certificate issued by a certificate authority which is trusted by the authenticating enterprise. There are only a few well known certificate authorities that are commonly used such as VeriSign [VeriSign2002verisign] and RSA Keon [RSA2002keon], therefore compatibility is not an issue. The PI authentication process is as follows:

1. A PI proxy receives a message from a peer PI proxy.
2. If the recipient PI proxy is configured to require authentication, it requests the sender PI proxy to send its identity, and some predetermined information encrypted with the sender PI proxy's private key.
3. The recipient PI proxy upon receipt of the sender's identity requests the trusted certificate authority for the digital certificate of the enterprise that has been claimed.

4. Upon receipt of the digital certificate, the recipient PI proxy attempts to decrypt the message with the claimed enterprise's public key.
5. If the message is successfully decrypted, the identity of the enterprise is proven and authentication is successful. Otherwise authentication fails.
6. In case of successful authentication, a session can be established for authenticated transfer of subsequent messages.

It appears that the certificate authority may pose as a bottleneck in the authentication process due to its central server nature. This however is not necessarily true. The main reason is that the information exchanged with the certificate authority is minimal and does not form the bulk of the traffic. The certificate authorities also take measures to ensure that their service is scalable. VeriSign [VeriSign2002verisign] is one of the best known and most widely trusted certificate authorities. VeriSign can be quoted on the scalability issue as follows [VeriSign2000services]:

> "VeriSign's software architecture is designed to distribute the processing and avoid performance bottlenecks. Additional servers can be added on as needed when transaction volume increases. The platform provides automatic failover, load-balancing, and threshold-monitoring on critical servers."

RSA Keon [RSA2002keon] by RSA Security [RSA2002rsa] is another major certificate authority. RSA Security claims that RSA Keon can be scaled to more than eight million certificates per instantiation, without compromising or losing integrity to performance and manageability [RSA2001scalability]. This claim has been independently evaluated and verified by Sun Microsystems' iForce Ready Center in Menlo Park, California [Sun2002iforce]. RSA Security does not state if eight million certificates per instantiation are adequate for current and future demand.

5.4.1.2 Authorization / Access Control

Each client can be given access to selected functions of the Web Service. An access list is maintained for each client by the Web Service's PI proxy. The access list contains a

matrix that indicates which functions of the Web Service are accessible to that client. Whenever that client requests access to some function, the PI proxy looks up its access rights in its list and allows it access only if it has the permission.

To determine which function the client is attempting to access, the PI proxy parses the requesting message and extracts the message name which is conventionally the same as the function name listed in the access list. The PI proxy can be configured to either let through or block messages that request functions not listed in the access list.

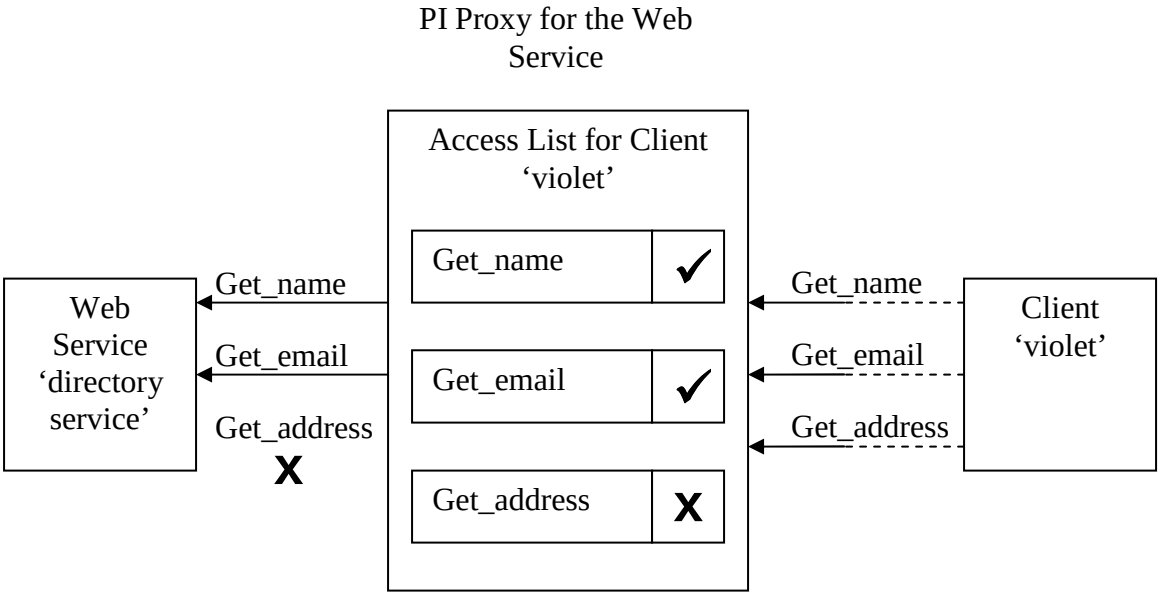

Figure 5.10: An example of PI access control

5.4.1.3 Confidentiality

Confidentiality is provided by using public key cryptography and digital certificates in a process somewhat similar to that of authentication. PI requires that the enterprise that would receive a confidential message must have a digital certificate issued by a certificate authority which is trusted by the sending enterprise. The PI process of delivering confidential messages is as follows:

1. A PI proxy receives from its local application a message which is meant to be relayed by confidential means.
2. The PI proxy requests the trusted certificate authority for the digital certificate of the recipient enterprise.
3. Upon receiving the digital certificate, the sender PI proxy encrypts the message with the recipient's public key.
4. It then sends the encrypted message over the regular communication channel that is the Internet. The message can be intercepted by a third party during transmission, however the message remains confidential since the message is in encrypted form and can only be decrypted by the intended recipient.
5. When the destination peer PI proxy receives the encrypted message, it decrypts it with its own private key.
6. The decrypted message is then delivered to the local application.

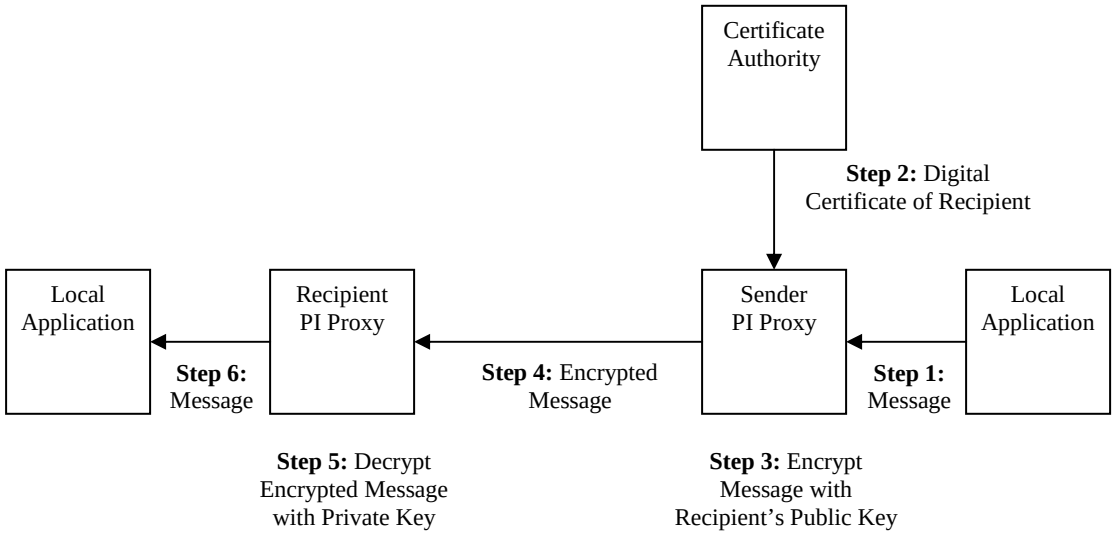

Figure 5.11: The PI process of delivering confidential messages

5.4.1.4 Non-Repudiation

Non-repudiation is provided by using digital signatures. The concept of digital signatures is based on public key cryptography. Digital Signatures are discussed in detail in [Buchmann2001crypt].

The process of providing non-repudiation is quite similar to that of authentication. However, unlike the processes of authentication and confidential delivery, the non-repudiation process requires permanent storage of each message which is guaranteed non-repudiation. PI requires that each enterprise that would send messages that cannot be repudiated, must have a digital certificate issued by a certificate authority which is trusted by the recipient enterprise. The PI process to guarantee non-repudiation of a message is as follows:

1. A PI proxy that is required to relay a message that cannot be repudiated, encrypts the message with its own enterprise's private key. This process is called digital signing of the message.
2. The digitally signed message is then sent to the peer PI proxy of the recipient enterprise.
3. The recipient PI proxy obtains the digital certificate of the sender enterprise and decrypts the message with the sender enterprise's public key.
4. If the encrypted message is decrypted successfully with the public key, it is proof that the message was encrypted by that enterprise's private key. Since only that enterprise is supposed to have the knowledge of its own private key, it is evidence that it is the only possible sender of the message.
5. The recipient PI proxy saves the digitally signed message in permanent storage as evidence to resolve any future repudiation claims by the sender.
6. The decrypted message is delivered to the local application.

5.4.2 Reliability

5.4.2.1 Guaranteed Delivery

Without PI or some other guaranteed delivery mechanism, a Web Service application's attempt to deliver a message to another Web Service application can fail in many ways. If

there is a temporary network outage at the time a send is attempted, the operation simply fails. Even if the send is successful, there is no guarantee that the message has been successfully received by the destination application.

PI provides guaranteed delivery of messages unless the destination is permanently unreachable. The following algorithm inspired by a similar algorithm given in [Coulouris2001distributed] – section 11.4.2, is used by PI to provide guaranteed delivery of messages:

1. When a PI proxy receives a message from the local application, it places it in the rear of a message queue called ‘ready to send’.
2. A ‘number of retries’ variable with the original value zero is associated with each message in this queue.
3. The PI proxy attempts to deliver the front most message in this queue to the destination peer PI proxy.
4. If the message is successfully sent, it is moved to another queue called the ‘awaiting acknowledgment’ queue.
    i. For each message in the ‘awaiting acknowledgement’ queue, the ‘number of retries’ variable is retained and a timer variable ‘time to wait for acknowledgment’ is also associated with it. The original value of this timer variable is some preconfigured value.
    ii. If the sender PI proxy receives an acknowledgement for the message before the timer expires, the message delivery is complete and it is removed from the queue.
    iii. If the timer expires and no acknowledgment has been received, the message is moved to the rear of the ‘ready to send’ queue along with its ‘number of retries’ value.
    iv. If an acknowledgment is received for a message that is not in the ‘awaiting acknowledgment’ queue, PI proxy looks for the message in the ‘ready to send’ queue. If it finds it there and it is not being sent, the message is removed and the delivery is considered complete. Otherwise the acknowledgment is disregarded. The reasons for an acknowledgment being disregarded include: a duplicate acknowledgment is received, the proxy has given up on a message and it is no longer in any queue, or that the message is already being sent again.

5. If the message is undeliverable at the moment, it is moved to the rear of the ‘ready to send’ queue and the message’s ‘number of retries’ variable is incremented by one.
6. The PI proxy keeps trying to redeliver a message in the ‘ready to send’ queue as long as the message’s ‘number of retries’ variable is less than the preconfigured ‘maximum number of retries’ variable. If the message is still undelivered, the destination is assumed to be permanently unreachable. In this case the message is removed from the queue and an entry is made into a permanent log indicating the failure of its delivery.
7. In place of or in conjunction with ‘number of retries’, it is possible to have a timer variable ‘interval’ associated with each message in the ‘ready to send’ queue. Each message’s redelivery can then be retried after its timer has expired after each attempt.

When a PI proxy receives a message from a sender PI proxy, its job is to relay this message to the local application. If it is successfully able to send the message to the application, it also sends an acknowledgment to the sender PI proxy for the delivery of that message. Each message is assigned a unique ID according to some unique naming/numbering scheme. Therefore acknowledgments are not mixed up for different messages. Please see section 6.2.4.2 for a scheme that assigns a universally unique ID to each message.

The whole process described above ensures that maximum effort is made to deliver a message. If a message is considered delivered, it is guaranteed that it has been delivered. If the message is impossible to deliver, the fact is noted in a permanent log for reference and help in resolution of the problem.

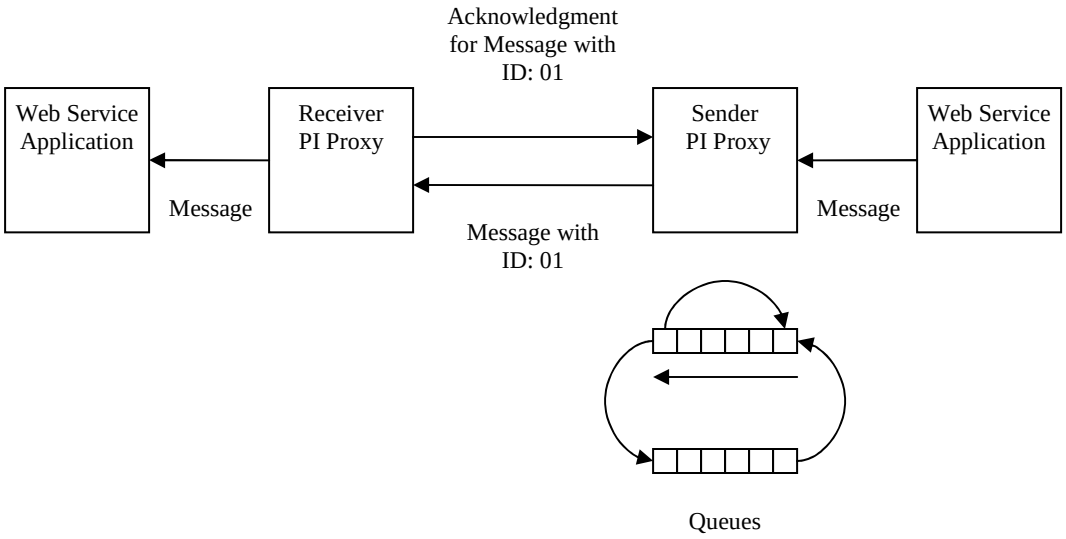

Figure 5.12: PI guaranteed message delivery

5.4.2.2 Ordered Delivery

PI uses a FIFO ordering algorithm for the ordered delivery of messages. For each peer PI proxy, a PI proxy maintains two variables: ‘messages sent’ and ‘messages received’. ‘messages sent’ is a count of the messages that have been sent to that peer PI proxy and ‘messages received’ is the count of messages that have been received from that peer PI proxy. The PI process for ordered delivery of messages is as follows:

1. When a message is sent to a peer PI proxy, the variable ‘messages sent’ for that peer PI proxy is piggy backed with the message. The variable ‘messages sent’ for that peer PI proxy is then incremented by one.
2. The recipient PI proxy upon receiving the message compares the ‘messages sent’ value received with the message with its own value of ‘messages received’ for the sender PI proxy.
3. If ‘messages sent’ = ‘messages received’ + 1, then the message is in the correct order and it is delivered to the local application. The ‘messages received’ variable for the sender peer PI proxy is then incremented by one by the recipient PI proxy.
4. If ‘messages sent’ > ‘messages received’ + 1, the message is placed in a waiting queue until all the intervening messages have been delivered. The waiting message is

then removed from the queue, delivered to the local application and the ‘messages received’ variable for the sender PI proxy appropriately incremented by one by the recipient PI proxy.

A similar algorithm is discussed in detail in [Coulouris2001distributed] – section 11.4.3.

5.4.2.3 Exactly-Once Delivery

The sender PI Proxy assigns each message a unique ID. The recipient PI proxy maintains record of the previously received messages for some set amount of time. Upon receipt of a message, the PI proxy compares the ID of that message with the IDs of the previously received messages. If the message has not been previously received, it is delivered to the local application. Otherwise it is discarded. This mechanism keeps duplicate messages from being delivered.

5.4.3 Manageability

5.4.3.1 Monitoring

The goal of monitoring is to keep the manager of a PI proxy informed of the events and activities that are taking place in the PI proxy. The manager can use this information to detect or predict problems and plan for future improvements.

The ongoing activity can be viewed from the PI proxy interface. For example the guaranteed delivery of a message may produce the following messages on the PI proxy interface:

- Message received from local application, ID: 01 assigned to message.
- Attempt 1 to send message ‘01’ failed, message placed in the rear of ‘ready to send’ queue.
- Attempt 2 to send message ‘01’ successful.
- Message ‘01’ placed in ‘awaiting acknowledgment’ queue.
- ‘time to wait for acknowledgment’ timer for message ‘01’ set to 60 seconds.
- Acknowledgment received for message ‘01’, delivery successful.

5.4.3.2 Logging

Monitoring information is logged in text files for long-term storage. Logs can be used for graphing trends of activities such as failed message sends, duplicate messages received etc. over long periods of time. This can give information about the performance of the communicating Web applications and PI proxy.

5.4.3.3 Client Management

A separate profile is maintained for each client (peer PI proxy). This profile contains the client's location, access list, state variables such as 'messages sent' and 'messages received' etc. The profile is configurable from the PI proxy interface.

Chapter 6: Implementation and Evaluation

We begin this chapter by describing a simple Bank Web Service that we have developed. We then continue to describe our implementation of PI and its evaluation. The Bank Web Service is described first because it was developed in that order and some concepts introduced in its description are used in the description of PI.

6.1 The Bank Web Service

The Bank Web Service is a simple Web Service that provides basic bank services such as viewing account balances, withdrawing money and depositing money into bank accounts. We have developed the Bank Web Service for the purpose of exercising and evaluating the various features of PI. The Bank Web Service is appropriate for this purpose because it can take meaningful advantage of most of the quality-of-service features provided by PI.

6.1.1 Description of Functionality

The Bank Web Service provides the following three functions:

Get Account Balance

This function allows a user to retrieve the current balance (amount of available money) of an account. This function takes two arguments: the account number of the account for which the request is made and the PIN (password) for that account number. The function

checks the given account number and PIN with its records and if they are valid it returns the current balance of the account. Otherwise it returns an appropriate error message.

Withdraw Money

This function allows a user to withdraw money from an account. This function takes three arguments: the account number of the account from which money is to be withdrawn, PIN for that account number and the amount of money requested. The function proceeds to return money if the given combination of account number and PIN is valid, otherwise it returns an appropriate error message. If the amount of money available in the account is greater or equal to the requested amount, the requested amount of money is deducted from the balance and returned to the user. If the available amount is less than the requested amount, the whole available amount of money is returned and the balance is set to zero. We perceive the amount being returned as virtual money.

Deposit Money

This function can be used by any user to deposit an amount of money into an account. The function takes two arguments: the account number and the amount of money to be deposited. The function adds the deposit amount to the balance of the requested account and returns a receipt to the user.

6.1.2 How can the Bank Web Service Benefit from PI?

The following subsections list the benefits that the Bank Web Service gains by using the quality-of-service features provided by PI. This also shows that the Bank Web Service is suitable for thorough exercise and evaluation of PI.

6.1.2.1 Security

- Authentication – Authentication would ensure that only authorized users are able to use the Bank Web Service.
- Confidentiality – Confidentiality would ensure that PINs and Money are not stolen by third-parties during transmission.
- Non-repudiation – Non-repudiation has separate but similar advantages for the user and the bank. When a user withdraws money from the bank, the bank would have evidence of that transaction and the user would not be able to deny it. Similarly, when the user deposits money into the bank, the user would have evidence of the transaction and the bank would not be able to deny it.

6.1.2.2 Reliability

- Guaranteed delivery – Guaranteed delivery would ensure that money sent by the bank to a user is received by the user and is not lost in the way.
- Ordered delivery – Ordered delivery would eliminate problems such as the following: The original balance of an account is $0. A user sends a deposit of $100 for the account. After that he sends a request for withdrawal of $50. If the second request is received out-of-order and before the first request, it would be denied by the bank (because there is yet no money in the account) even though from the user's point-of-view it is a legitimate request.
- Exactly-once delivery – This feature is vital for the bank since it would not want money intended to be sent once, erroneously get sent multiple times.

6.1.2.3 Manageability

- Monitoring – Monitoring would be essential for the customer service department to make sure that the system is running smoothly.
- Logging – Logs would be helpful for accounting and planning purposes.

6.1.3 Design and Implementation

The Bank Web Service is a client-server application written in Java. The Web Service client and the server use the JAXM API to create and read SOAP messages, and communicate them over an HTTP connection.

6.1.3.1 Server

The Bank Web Service server is implemented as a Web Application which is hosted by a Web Application Container, the Tomcat Container in this case.

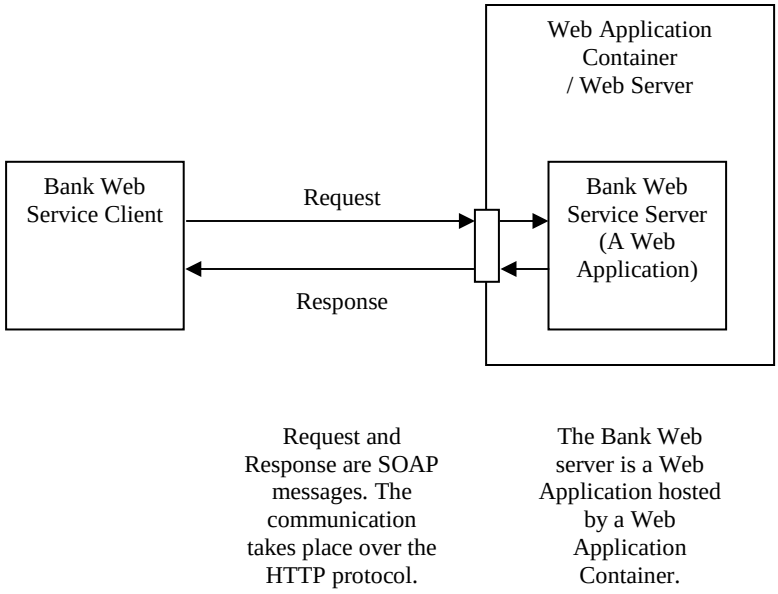

Figure 6.1: The Bank Web Service architecture

A Web Application Container is a software program that has the basic functionality of a Web Server but can also host Web Applications. Web Applications are applications that don't have their own web server capabilities but can use those of a host Web Application Container. Developers of Web Applications don't need to implement web server functionality in their applications. They can instead employ those already provided by a Web Application Container.

The Bank Web Service server receives request messages from the client. All messages are communicated in the form of SOAP messages using the HTTP protocol over the Internet. The Bank Web Service operates in a request-response manner. Each request message from the client is followed by a response message from the server. The server logs all messages in a local text file.

6.1.3.1.1 Classes

The Bank Web Service server is composed of three classes: `bank_server`, `account_list` and `account`. The following UML class diagram shows these classes and their relationships:

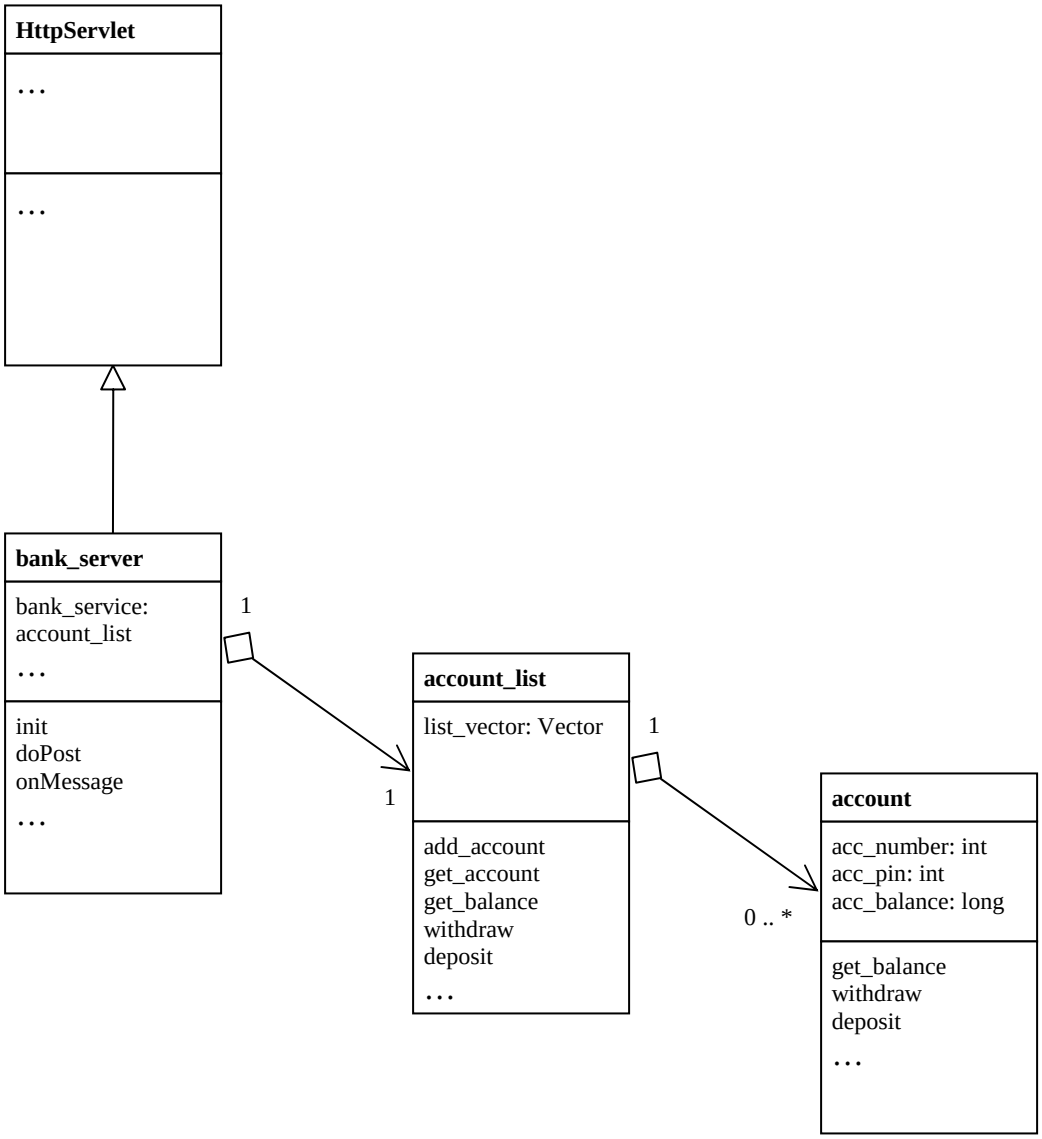

Figure 6.2: UML class diagram of the Bank Web Service server

An object of the `account` class represents a bank account. It contains the account number, PIN and the balance for that account. The class provides `get_balance`, `withdraw` and `deposit` methods.

An object of the `account_list` class contains a list of `account` objects. The class provides methods to add and retrieve `account` objects from the list. It also provides `get_balance`, `withdraw` and `deposit` methods that locate the requested account in the list and call its corresponding method.

The `bank_server` class extends the Java `HttpServlet` class. A `bank_server` object is a client's Web Services interface to the `account_list` object. The `bank_server` object upon receiving a SOAP message from the client extracts the request and then calls the requested method of the `account_list` with the provided arguments. It then creates a new SOAP message with the method's response in it and sends it back to the client.

6.1.3.2 Client

The Bank Web Service client is a self-contained application. It uses the JAXM API to create and read SOAP messages as well as create an HTTP connection to the server for sending and receiving the messages.

6.1.3.2.1 Client's Graphical User Interface

The Bank Web Service client has a Graphical User Interface (GUI) developed using the Java Swing API.

When the Bank Web Service client is started, it prompts the user for the *account number*, *PIN* and the *URL of the Bank Web Service server*. These values are then stored for the session. The values can also be changed later during the session.

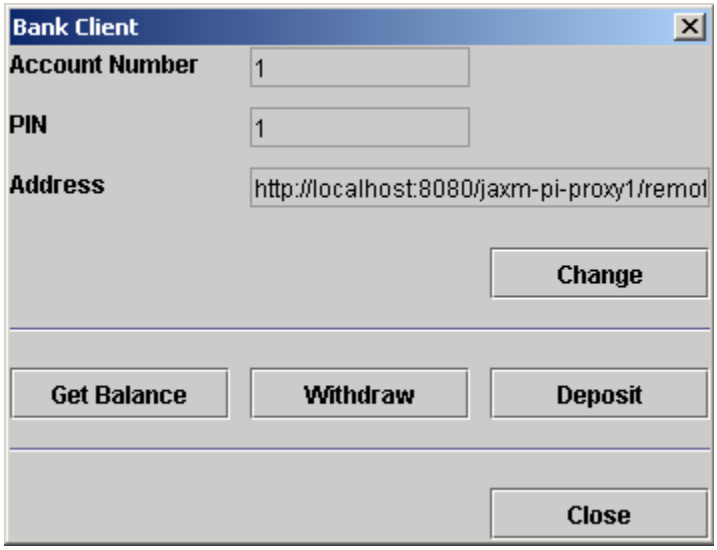

Figure 6.3: The Bank Web Service client's main dialog

The main dialog provides options to perform the *get account balance*, *withdraw* and *deposit* functions using the stored values. The user is prompted for the *amount* for the *withdraw* and *deposit* functions. The GUI displays the response received from the server or an appropriate error message in case of an exception.

6.1.3.2.2 Classes

The Bank Web Service client is composed of five classes: `bank_client`, `bank_client_gui_main`, `bank_client_gui_change`, `bank_client_gui_amount` and `bank_client_gui_response`. The later four classes compose

the Bank Web Service client's GUI. `bank_client_gui_main` is the class for the main dialog. It contains the rest of the GUI classes and instantiates them to receive input from the user or display output. The following UML class diagram shows the classes `bank_client` and `bank_client_gui_main` and their relationship:

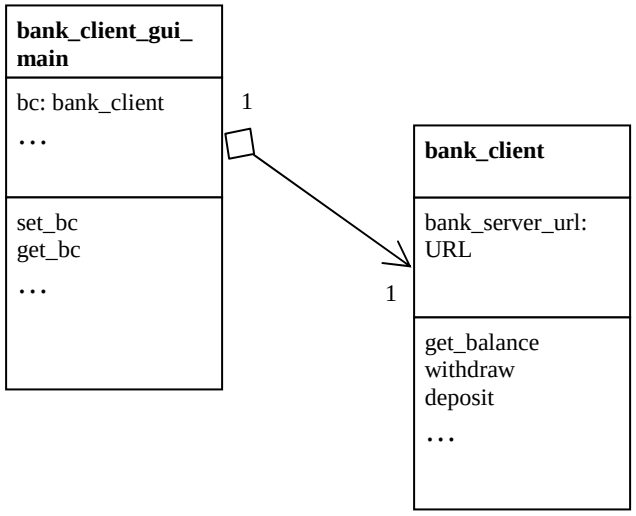

Figure 6.4: UML class diagram of the Bank Web Service client

The `bank_client` class provides the methods: `get_balance`, `withdraw` and `deposit`. Each of these methods takes the arguments required to invoke corresponding methods in the Bank Web Service server's `account_list` class. The method then creates a SOAP message that includes the name of the requested function and the given arguments. An HTTP connection is created to send the SOAP message to the server. The response is extracted from the SOAP message returned by the server. This response is the return value of the method.

A `bank_client_gui_main` object contains a `bank_client` object. It receives input from the user, calls the appropriate methods of the `bank_client` object and then displays the response back the user.

6.1.4 Sample Messages Exchanged between the Bank Web Service Server and Client

The following SOAP message is a request sent by a client to withdraw $444 from an account with the account number: 3, and PIN: 95.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-
env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Body>
      <bank:request xmlns:bank="blueberrypi">
         <name>withdraw</name>
         <account_number>3</account_number>
         <pin>95</pin>
         <amount>444</amount>
      </bank:request>
   </soap-env:Body>
</soap-env:Envelope>
```

The server after processing the request, sends the following message to the client. The message returns the requested $444 to the client.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-
env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Body>
      <bank:response xmlns:bank="blueberrypi">
         <description>Money Returned: 444</description>
      </bank:response>
   </soap-env:Body>
</soap-env:Envelope>
```

The following SOAP message is a request by a client to deposit $768 to an account with the account number: 5.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-
env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Body>
      <bank:request xmlns:bank="blueberrypi">
         <name>deposit</name>
         <account_number>5</account_number>
         <amount>768</amount>
      </bank:request>
   </soap-env:Body>
</soap-env:Envelope>
```

The server after processing the request, sends the following message to the client. The message is a receipt for the deposit of $768.

```
<?xml version="1.0" encoding="UTF-8"?>
<soap-env:Envelope xmlns:soap-
env="http://schemas.xmlsoap.org/soap/envelope/">
   <soap-env:Body>
      <bank:response xmlns:bank="blueberrypi">
         <description>Amount Deposited to Account Number 5:
         768</description>
      </bank:response>
   </soap-env:Body>
</soap-env:Envelope>
```

6.2 Implementation of PI Proxy

We have currently implemented a subset of the PI proxy design that we gave in chapter 5. Our implementation includes the basic infrastructure required to relay messages between

a local Web Service application and a peer PI proxy, and the following quality-of-service features: guaranteed delivery, exactly-once delivery and logging. Other quality-of-service features have not been implemented due to time constraint. The current implementation is however adequate for a basic evaluation of PI, which is given in section 6.3.1.

6.2.1 Some Basics of the Current Implementation of PI Proxy

We call the current implementation of PI proxy: PI Proxy Version 1.0, abbreviated as PI-v1. PI-v1 has been implemented as a Web Application. Web Applications were introduced in section 6.1.3.1. We have used the Tomcat container to host PI-v1.

PI-v1 is written in Java. Java was selected because of its platform independence. PI-v1 uses the JAXM API for reading and editing SOAP messages. It also uses the JAXM API to send SOAP messages over an HTTP connection. The use of Java or the JAXM API does not limit the interoperability of PI proxy. Although, we have not tested this claim, we believe that PI-v1 should also work perfectly with Web Services developed on other Web Services development platforms such as Microsoft .NET. The ground for this claim is that all Web Services communicate messages in the same standard SOAP format.

PI-v1 has three servlets: system, local_relay and remote_relay. A servlet is an individually addressable component of a Web Application. For example if the address of the PI proxy's Web Application is: http://piproxy/, then the servlets: system, local_relay and remote_relay can have the following respective addresses: http://piproxy/system, http://piproxy/local_relay and http://piproxy/remote_relay.

The servlet system contains variables such as the address of the local Web Service application and the address of the remote peer PI proxy. The other servlets retrieve the

values of these variables from this servlet when they need them. The servlet system also provides a connection to the PI proxy administration tool (discussed in section 6.2.6), which allows a user to view and edit the user-definable settings of a PI proxy.

The servlets local_relay and remote_relay provide the function of relaying messages between two Web Service applications. These servlets also implement the quality-of-service features provided by PI-v1. The function of these servlets is given in detail in sections 6.2.3 and 6.2.4.

6.2.2 Limitations of PI-v1

PI-v1 has the following limitations:

- PI-v1 can only handle request-response style messages. In this style of messaging each request message from a client to a server is always followed by a response message from the server to the client.
- PI-v1 can be configured for only one remote Web Service application. Provisions have been made in the code to modify PI-v1 easily to allow communication with multiple remote Web Service applications from a single PI proxy.

6.2.3 Implementation of Basic Message Relay

The following steps are taken when a Web Service application uses PI-v1 to send a request message to another Web Service application and receive a response from it.

For convenience, let us call the Web Service application that sends the request: WS-1, and the Web Service application that receives the request and then sends the response: WS-2.

1. WS-1 sends a request message destined for WS-2 on the address of its (WS-1's) local PI proxy's remote_relay servlet.
2. The remote_relay servlet receives the request message and transmits it to the local_relay servlet of WS-2's PI proxy. Meanwhile the remote_relay servlet keeps the connections open with both WS-1 and the local_relay servlet of WS-2's PI proxy.
3. The local_relay servlet of WS-2's PI proxy upon receipt of the request message forwards it to WS-2. Meanwhile the local_relay servlet keeps the connection open with WS-2 and the remote_relay servlet of WS-1's PI proxy.
4. WS-2 processes the request and sends a response message back to its local PI proxy's local_relay servlet on the open connection.
5. The local_relay servlet receives the response message, closes the connection with WS-2 and sends the message to the remote_relay servlet of WS-1's PI proxy on the open connection.
6. The remote_relay servlet upon receipt of the response message closes the connection with the local_relay servlet of WS-2's PI proxy. It then sends the response message to WS-1 on the open connection and closes the connection. This step completes the process.

These steps are illustrated in figure 6.5. If WS-2 sends a request message to WS-1, the exact same process occurs but in the opposite direction.

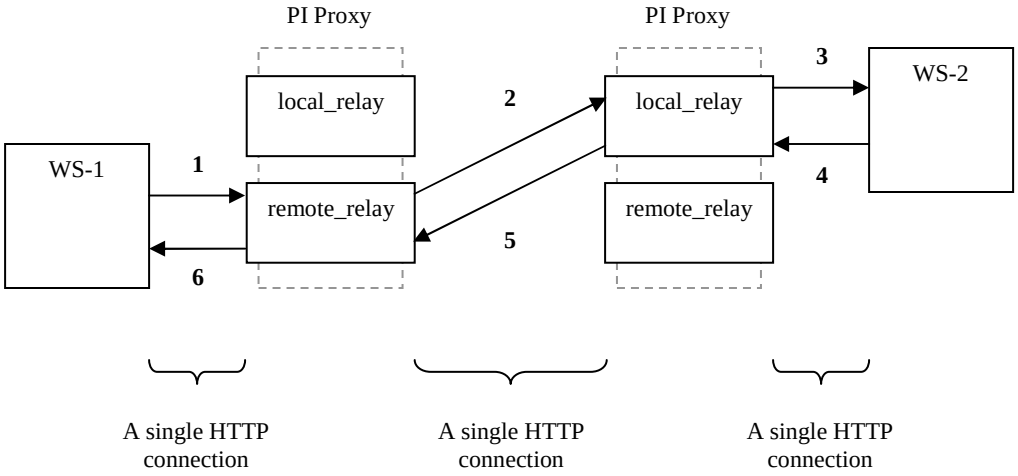

Figure 6.5: Basic message relay

Connection to a PI proxy's remote_relay servlet from outside the trusted network is blocked by a firewall. Only the local Web Service application is allowed to initiate an HTTP connection on the address of its PI proxy's remote_relay servlet. This step is taken so that an imposter cannot send messages through a PI proxy, posing as its local Web Service application.

6.2.4 Implementation of Quality-of-Service Features

6.2.4.1 Guaranteed Delivery

The local_relay and remote_relay servlets use the javax.xml.soap.SOAPConnection.call method to send a message and receive a response. The method throws an exception when the destination does not respond immediately.

There are two user-configurable variables in PI-v1 that govern guaranteed delivery: number of retries to send a message (let us call this variable: `r`), and time in seconds between retries (let us call this variable: `s`).

The local_relay and remote_relay servlets use the following algorithm to implement guaranteed delivery. The destination is considered permanently unreachable if send/receive does not succeed within the given number of tries.

```
int i = 0;
boolean exception = true;

while ((exception == true) && (i < r+1))
//while send/receive fails and retries are remaining
{
   try
   {
      response = connection.call(request, destination);
      exception = false;  // send/receive succeeded
   }
   catch (Exception error)
   {
      exception = true;   //send/receive failed
   }

   i++;

   if ((i < r+1) && (exception == true))
   //if there are retries remaining and send/receive failed
   {
      wait(s);  //wait for s seconds
   }
}
```

According to this algorithm, `r` number of retries are made to send a request and receive the response. A time interval of `s` seconds is given between each retry. The response message is considered the acknowledgement of the request message. This implementation does not provide acknowledgements for response messages.

6.2.4.2 Exactly-Once Delivery

Before the remote_relay servlet sends a message to the peer PI proxy's local_relay servlet, it adds a universally unique message ID to the message. The local_relay servlet upon receipt of the message extracts and removes the message ID. The same steps are taken in the opposite direction when a local_relay servlet sends a message to the peer PI proxy's remote_relay servlet. This process forms the basis for enforcing exactly-once delivery.

A message ID has the format shown in the following figure:

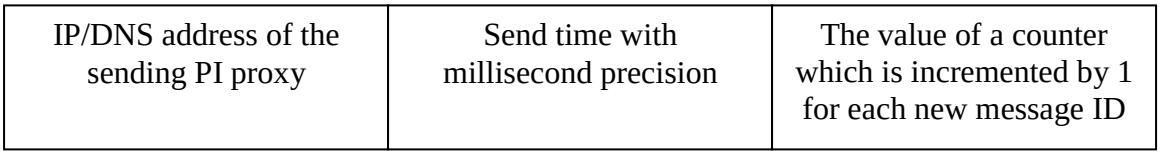

Figure 6.6: Format of the message ID

The incremented value of a counter is added to each message ID to ensure uniqueness of multiple messages sent within a single millisecond. PI-v1 uses a counter of type long. The messages sent within a millisecond will be unique as long as the counter does not complete a cycle during that millisecond.

The remote_relay and local_relay servlets each maintain a list of recently received message IDs. The number of IDs maintained is user-configurable. Whenever a message is received by one of these servlets, it checks its message ID with the message IDs in the list. If a match is found, it means that the newly received message is a duplicate that has

already been received. The message is discarded in this case, otherwise it is delivered to the local Web Service application.

6.2.4.3 Logging

The local_relay and remote_relay servlets each generate detailed logs of their activity. The information logged includes:

- Time a request message is received
- Time a response message is returned
- The complete content of the messages received
- The complete content of the messages sent
- The message ID added to a message
- The message ID extracted from a message
- The address on which a message is sent
- The event of receipt of a duplicate message
- The event of a retry to send a message
- The event of a failure to send a message
- All exceptions and errors generated by the classes

6.2.5 Classes

PI-v1 is composed of the following seven classes: `local_web_service`, `remote_web_service`, `remote_web_services_list`, `local_relay`, `remote_relay`, `system` and `log_machine`. The following UML class diagram shows these classes and their relationships:

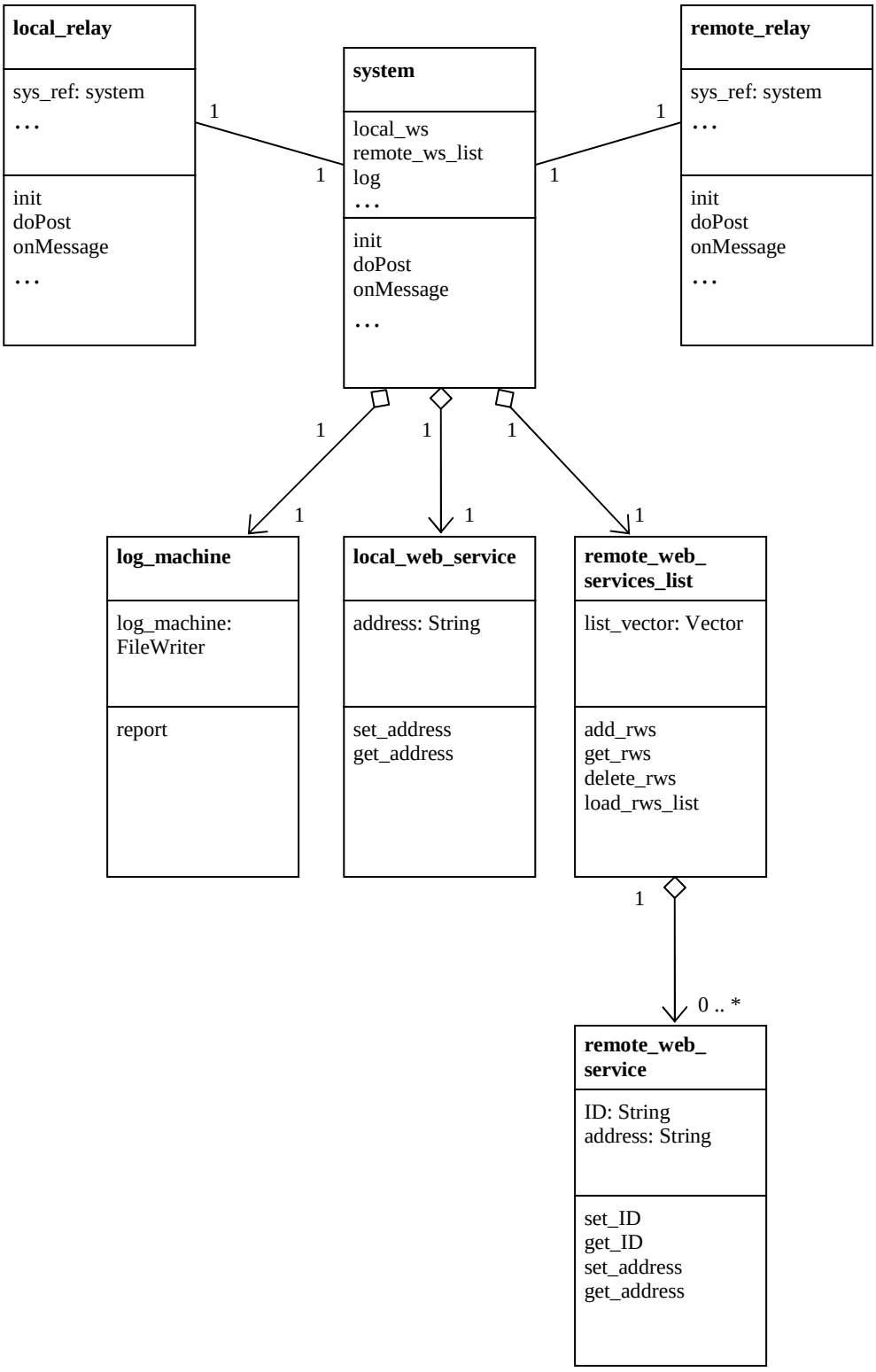

Figure 6.7: UML class diagram of the current implementation of PI proxy

The classes `system`, `local_relay` and `remote_relay` extend the Java `HttpServlet` class. The following UML class diagram shows this generalization/specialization relationship:

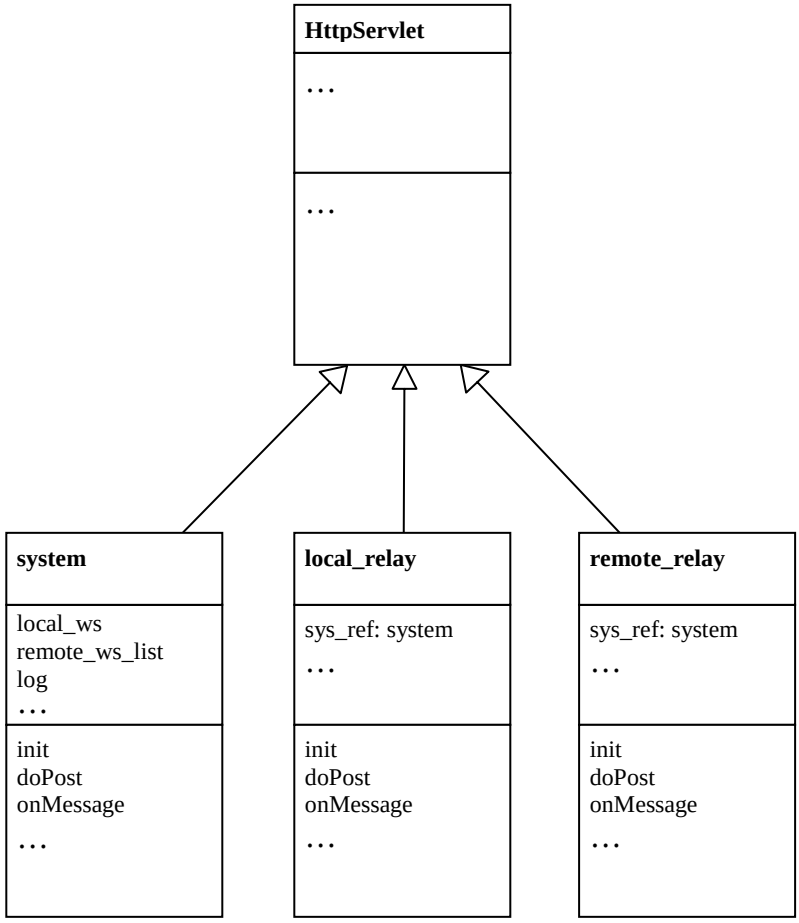

Figure 6.8: UML class diagram showing the generalization/specialization relationship of `system`, `local_relay` and `remote_relay` with `HttpSerlvet`

The `local_web_service` class represents the local Web Service application. It contains the address of the local Web Service application.

The `remote_web_service` class represents a remote Web Service application. It contains an ID and the address of that remote Web Service application. The ID of a remote Web Service application is unique within a single instantiation of a PI proxy.

The `remote_web_services_list` class contains a list of the `remote_web_service` class. It provides methods to add, retrieve and delete `remote_web_service` objects from the list. The add method enforces that the ID in each `remote_web_service` object is unique throughout the list. In the current implementation only a single `remote_web_service` object with the ID ‘default’ is added to the list.

The `log_machine` class provides logging facilities for the rest of the classes. Other classes may call upon `log_machine` to log various messages that they generate, for example messages generated due to exceptions.

The `system` class contains some global variables and constants, including a variable each of the classes: `local_web_service`, `remote_web_services_list`, and `log_machine`. The classes: `local_relay` and `remote_relay`, retrieve and use the values of these variables and constants. `system` is a servlet class. It provides a connection to the PI Proxy Administration Tool for updating user-definable values.

`local_relay` is a servlet class that receives request messages from a remote peer PI proxy and relays them to the local Web Service application. It relays the responses to those requests back to the peer PI proxy. The class logs the received and sent messages and other details about the message exchange such as time etc.

`remote_relay` is a servlet class that receives request messages from the local Web Service application and relays them to the remote peer PI proxy. It relays the responses to those requests back to the local Web Service application. This class is very similar to the class `local_relay`. It also logs the received and sent messages and other details about the message exchange.

Messages sent to a peer PI proxy by the class `remote_relay` are received by the class `local_relay` and vice versa. `local_relay` and `remote_relay` implement the following quality-of-service features in the message exchange: guaranteed delivery, ordered delivery and exactly-once delivery.

6.2.6 PI Proxy Administration Tool

The PI proxy administration tool is an application that can be used to view and change various settings of a PI proxy. These settings include:

- Address of the local Web Service application
- Address of the remote Web Service application
- Number of message IDs to retain in the ‘exactly-once delivery’ message ID list
- Number of retries to send a message
- Interval between retries
- Number of times to resend each message for testing ‘exactly-once delivery’

The tool interacts with the servlet `system` of a PI proxy. The interaction is in form of SOAP messages over an HTTP connection. The tool retrieves and displays the

current settings of the PI proxy. The user can change those values and submit them to the PI proxy which then updates its variables accordingly.

The tool has a GUI that has been developed using the Java Swing API. The GUI is shown in the following figure:

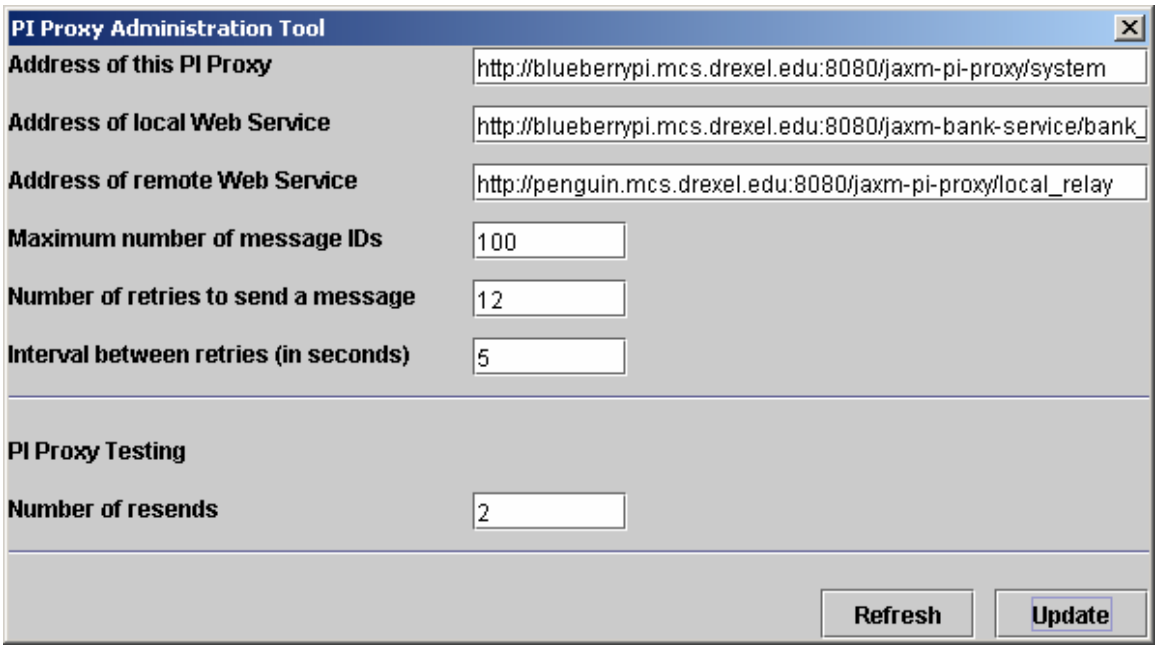

Figure 6.9: GUI of the PI proxy administration tool

6.3 Evaluation of PI Proxy Version 1.0

6.3.1 Does PI Proxy Version 1.0 Achieve the Design Objectives of PI?

In section 5.3 we theoretically demonstrated that PI meets the objectives set in section 5.1. In this section we give practical demonstrations to substantiate our claim. PI Proxy Version 1.0, an implementation of the PI design is used for this purpose.

6.3.1.1 Objective 1: The Solution should not Require Additional Coding for Each Deployment

PI-v1 is a Java Web Application, packaged in a single distributable Web Application Archive (WAR) file: `jaxm-pi-proxy.war`. WAR files are described in detail in [Bodoff2002war]. `jaxm-pi-proxy.war` contains the PI-v1 classes and deployment instructions for the host Web Application container.

The following steps are required to deploy PI-v1 on the Tomcat Web Application container (on any operating system).

We assume that a standard installation of the Tomcat container is already available on the machine where PI-v1 is being deployed. Installation of Tomcat is not a complicated procedure. Detailed instructions on how to install Tomcat are given on [Sun2002jwsdp].

1. Place `jaxm-pi-proxy.war` in the following directory: `<JWSDP_HOME>/webapps`, where `<JWSDP_HOME>` is the directory where JWSDP and Tomcat are installed.
2. Restart Tomcat. This step completes the deployment of PI-v1.

We can see that PI-v1's deployment process is very simple and does not require any coding.

6.3.1.2 Objective 2: The Solution should not Place Control of the Information Flow and its Management in the Hands of a Single Party (Especially not a Third-Party)

Each enterprise has its own deployment of PI-v1 for its Web Service application. For example in the case of the Bank Web Service, the bank would have its own deployment of PI-v1 and the client would have its own. They can each configure their PI-v1 according to their own preferences using the PI proxy administration tool. Therefore each enterprise has equivalent control of the flow and management of its information. There is also architecturally no requirement for third-party involvement.

The bank and the client are free to use the PI proxy administration tool to configure their PI proxies with values different from each other. For example the bank could have the following settings:

```
Number of message IDs to retain in the ‘exactly-once
     delivery’ message ID list: 100000
Number of retries to send a message: 999
Interval between retries: 1 second
```

At the same time, the client could have the following completely different settings:

```
Number of message IDs to retain in the ‘exactly-once
     delivery’ message ID list: 100
Number of retries to send a message: 3
Interval between retries: 15 seconds
```

The bank's settings indicate that it wants a higher degree of reliability. These settings would provide better reliability but would also result in higher consumption of the computing resources. The bank may have significant computing resources available to it therefore it can decide to keep these settings. The client on the other hand may not have extensive computing resources available and therefore it chooses more moderate settings. The decision of both enterprises is independent of each other.

6.3.1.3 Objective 3: The Solution should be Easily and Quickly Deployable

Deployment of PI-v1 is easy and quick. We demonstrate this by laying out the steps required to deploy PI-v1 for the Bank Web Service client.

We assume that a standard installation of the Tomcat container is already available on the machine where PI-v1 is being deployed.

The following steps are required to deploy PI-v1 for the Bank Web Service client:

Install PI-v1 (the steps are the same as given in section 6.3.1.1):

1. Place `jaxm-pi-proxy.war` in the following directory: `<JWSDP_HOME>/webapps`, where `<JWSDP_HOME>` is the directory where JWSDP and Tomcat are installed. *(This is a simple file copy operation, estimated time required: 1 minute).*
2. Restart Tomcat. *(This operation requires the shutdown command followed by the startup command, estimated time required: 2 minutes)*

Configure PI-v1:

3. Run PI proxy administration tool. *(This operation requires a single command to execute the tool, estimated time required: 1 minute)*
4. Enter the following values (the administrator would have knowledge of these values beforehand) in the given fields:
   - Address of PI-v1 to be configured
   - Address of the remote_relay servlet of the Bank Web Service server's PI proxy
   - Number of message IDs to retain in the 'exactly-once delivery' message ID list
   - Number of retries to send a message
   - Interval between retries

   (This operation requires typing in five simple text values, estimated time required: 5 minutes)
5. Select the 'Update' button. *(This is a simple mouse operation, estimated time required: 30 seconds)*

Reconfigure the Bank Web Service client:

6. On the main dialog of the Bank Web Service client, select the 'Change' button. *(This is a simple mouse operation, estimated time required: 30 seconds)*
7. Change address from the address of the Bank Web Service server to the address of PI-v1. *(This operation requires typing in a simple text value, estimated time required: 1 minute)*
8. Select the 'OK' button. *(This is a simple mouse operation, estimated time required: 30 seconds)*

It should be noted that the estimated times mentioned are quite generous.

We can see that the deployment process is very straightforward and simple. It does not require any complex tasks. The time taken for one deployment is well under 15 minutes.

15 minutes is an extremely short time as compared to the time required to undertake a complete programming project for the deployment of some other Web Service intermediaries such as Grand Central.

6.3.1.4 Objective 4: The Solution should not Limit the Scalability of a Web Service

The current implementation of PI proxy allows communication with only one remote Web Service application at a time. This limitation was put in place to keep the development of PI-v1 simple. This limitation can however be removed in future versions of PI-v1. Provisions have been made in the code to modify PI-v1 to allow communication with multiple remote Web Service applications from a single PI proxy.

A practical evaluation of PI's scalability can therefore be conducted with a future version of the PI proxy implementation. A theoretical analysis of PI's scalability was given in section 5.3.4.2.

6.3.2 Test of Quality-of-Service Features Provided by PI Proxy Version 1.0

We have used the Bank Web Service to test the quality-of-service features provided by PI-v1. The tests and their results are given in the following subsections. Following is a description of the test environment:

The Bank Web Service server and its PI-v1 (let us call it PI-S) were hosted on a Tomcat container on a Solaris 7 machine – penguin.mcs.drexel.edu. The PI-v1 (let us call it PI-C) of the Bank Web Service client was hosted on a Tomcat container on a Windows 2000 machine – bluberrypi.mcs.drexel.edu. The Bank Web Service client, which is a self-contained application, was also run on the same machine that is bluberrypi.mcs.drexel.edu.

The address of PI-C's remote_relay servlet was: http://bluberrypi.mcs.drexel.edu:8080/jaxm-pi-proxy/remote_relay. The address of PI-S's local_relay servlet was: http://penguin.mcs.drexel.edu:8080/jaxm-pi-proxy/local_relay. The address of the Bank Web Service server was: http://penguin.mcs.drexel.edu:8080/jaxm-bank-service /bank_server. PI-C and PI-S were configured accordingly.

6.3.2.1 Guaranteed Delivery

The guaranteed delivery settings of PI-C were configured as follows:

```
Number of retries to send a message: 12
Interval between retries: 5 seconds
```

These settings mean that PI-C will try to resend a message every five seconds until the message is successfully sent or one minute has passed. The test was conducted as follows:

1. PI-S was stopped.
2. A 'Get Account Balance' message was then sent from the Bank Web Service client to PI-C's remote_relay servlet.
3. A time interval of approximately 30 seconds was then given before the next step.

4. PI-S was started.

Result: At the execution of step 4, the message was delivered to PI-S by PI-C and a response to that message was then received. Had the guaranteed delivery mechanism not been in place, the message delivery would have failed at step 2.

The following log entries were generated by PI-C's remote_relay during this test:

```
doPost entered: Sat Jul 20 22:23:19 EDT 2002
received from local web service: <request message>
msg_id added: blueberrypi.mcs.drexel.edu / Sat Jul 20
     22:23:19 EDT 2002 / 38
message being sent to:
     http://penguin.mcs.drexel.edu:8080/jaxm-pi
     -proxy/local_relay
message send failed, will retry in 5 seconds
message send failed, will retry in 5 seconds
message send failed, will retry in 5 seconds
message send failed, will retry in 5 seconds
message send failed, will retry in 5 seconds
message send failed, will retry in 5 seconds
message send successful
msg_id extracted: penguin.mcs.drexel.edu / Sat Jul 20
     22:23:20 EDT 2002 / 904
sent to local web service: <response message>
doPost exited: Sat Jul 20 22:23:20 EDT 2002
```

6.3.2.2 Exactly-Once Delivery

The 'exactly-once delivery' test setting of PI-C was configured as follows:

```
Number of times to resend each message for testing
'exactly-once delivery': 1
```

This setting means that PI-C will send every message one additional time. The test was conducted as follows:

1. A ‘Get Account Balance’ message was sent from the Bank Web Service client to PI-C’s remote_relay servlet.
2. PI-C’s remote_relay servlet according to the settings, sent the message twice to PI-S’s local_relay servlet

Result: At the end of the test only a single copy of the message was received at the Bank Web Service server. The second duplicate copy was discarded by PI-S’s local_relay servlet. Had the exactly-once delivery mechanism not been in place, the server would have received two redundant copies of the same message.

The following log entries were generated by PI-S’s local_relay during this test:

```
doPost entered: Sat Jul 20 17:38:11 EDT 2002
received from peer pi proxy: <request message>
msg_id extracted: blueberrypi.mcs.drexel.edu / Sat Jul 20
     17:38:11 EDT 2002 / 24
message being sent to:
     http://penguin.mcs.drexel.edu:8080/jaxm-bank
     -service/bank_server
message send successful
msg_id added: penguin.mcs.drexel.edu / Sat Jul 20 17:38:11
     EDT 2002 / 839
sent to peer pi proxy: <response message>
doPost exited: Sat Jul 20 17:38:11 EDT 2002

doPost entered: Sat Jul 20 17:38:11 EDT 2002
received from peer pi proxy: <request message>
msg_id extracted: blueberrypi.mcs.drexel.edu / Sat Jul 20
     17:38:11 EDT 2002 / 24
duplicate message received, discarded
doPost exited: Sat Jul 20 17:38:11 EDT 2002
```

6.3.2.3 Logging

PI-v1 generates detailed logs of all messaging activity. Sample log entries are given in the preceding two sections.

Chapter 7: Conclusion

7.1 Review

We opened the thesis with an introduction to Web Services. Web Services is a new technology for building distributed systems by enabling interaction between remote applications. The remarkable characteristic of the Web Services technology is that it allows interaction between heterogeneous applications regardless of their development and operational platforms. This is made possible by the use of XML for information exchange.

An enterprise-class Web Service was characterized as one that meets the quality-of-service requirements of its owner enterprise. Quality-of-service requirements include security, reliability and manageability.

We discussed that the Web Services technology does not currently support these quality-of-service features. Web Services protocols such as SOAP, have no provision for implementing quality-of-service requirements. Since Web Service applications communicate over the Internet (which is inherently insecure and unreliable), the security and reliability of the communication is put at risk.

We looked at some possible solutions to this problem. One obvious solution is to add quality-of-service features to the Web Services protocols. However, this solution may not be desirable because it would put extra burden on the currently simple protocols that are successful mainly due to their simplicity. A solution that does not have significant side-effects is that of a Web Service intermediary. In this solution Web Service applications communicate with each other through an intermediary. The

intermediary takes the responsibility of implementing the quality-of-service requirements in the communication.

This is a suitable solution and its current implementations do solve the problem of lack of quality-of-service but they introduce some problems of their own. One of these problems is the requirement to recode each Web Service application to enable it to communicate with the intermediary. This restriction makes the deployment of a Web Service intermediary difficult and time-consuming. Other problems are due to the central-server architecture of the existing Web Service intermediaries. A central server places control of the flow and management of all information in the hands of a single party and leaves other parties involved in the communication deprived of this control. In the Web Services architecture, a Web Service application may be a client as well as a server, therefore if two peer-to-peer Web Service applications want to communicate there is no basis to choose one of them over the other to host the Web Service intermediary server. Another significant problem with a Web Service intermediary based on a central-server is that it limits the scalability of a Web Service application.

To solve these problems we presented a novel Web Service intermediary, which we called PI. Our Web Service intermediary is based on peer-to-peer architecture. The key component of our Web Service intermediary architecture is a peer proxy. The peer proxy is designed to be a software program that is installed on the trusted network of each communicating Web Service application. A Web Service application sends and receives messages through its proxy. The peer proxies of two communicating applications interact to relay their messages. The peer proxies take measures to enforce the desired quality-of-service in the communication with each other. These measures include use of public-key

cryptography for security; use of message queues, ordered delivery and exactly-once delivery algorithms for reliability; and monitoring, logging and client management features for manageability.

We demonstrated that our Web Service intermediary does not suffer from the problems associated with earlier designs. It does not require Web Service applications to be recoded; it is easily and quickly deployable; due to its peer-to-peer architecture, the control of flow and management of information is not placed in the hands of a single party; there is no need to select one exclusive party to host a server; and it does not limit the scalability of a Web Service application.

We gave the description of our implementation of PI. Our current implementation implements only a subset of the complete design. With the help of this implementation we gave practical demonstrations that our solution does solve the problems associated with the designs of existing Web Service intermediaries. We did not provide a practical demonstration of the scalability of our solution since the current implementation of PI proxy cannot be configured to communicate with more than one peer PI proxy. We used our Bank Web Service to test various quality-of-service features provided by our implementation of PI. All tests had successful results.

7.2 Contributions

The main contribution of our work is the design of a novel Web Service intermediary. Our design solves some significant problems associated with earlier designs. Our Web Service intermediary has the following novel characteristics:

1. It does not require a Web Service application to be recoded to enable it to communicate with the Web Service intermediary.

2. It does not place the control of the flow and management of information in the hands of a single party.
3. It does not differentiate between a client and a server which is important for the Web Services architecture in which a single application can be a client and a server at the same time. It does not require selection of one exclusive party to host a Web Service intermediary server.
4. It is quickly and easily deployable.
5. It does not limit the scalability of a Web Service application.

Our Web Service intermediary makes building of enterprise-class Web Services convenient and practical: convenient because of characteristics 1 and 4, and practical because of characteristics 2, 3 and 5 mentioned above.

Although we have not given a full-featured, industrial-strength implementation of our design, we have provided a prototype which demonstrates that a practical implementation of our design is possible.

This thesis is a good introduction to the emerging Web Services technology. It can serve as a starting point and useful resource for students who would like to work in this area.

7.3 Future Research Possibilities

We have not given a standard which a proxy must adhere to in its communication with a peer proxy. If two proxies are implemented by independent groups of developers that use different communication mechanisms, then those two proxies would not be able to communicate with each other since they would not conform to each others

communication specifics. For example the two implementations could have a different format for message IDs. There is a need to develop a standard which different implementers of PI proxies can adhere to. The result would be that PI proxies implemented separately would be able to interact with each other.

We mentioned in section 5.3.2.2 that two peer PI proxies negotiate with each other to reach agreement on the quality-of-service to be implemented in their communication. We however did not give a negotiation algorithm. An ideal negotiation algorithm would be one that ensures that every possibility is considered which would lead to an agreement between the two parties. Research can be done towards developing such an algorithm. We stated that after negotiation the stricter of the two proxies' quality-of-service requirements is enforced. It is not mandatory to have this as the only possible condition of agreement. An acceptable condition of agreement may as well be a compromise between the quality-of-service requirements of the two proxies if both proxies are willing to accept it. This possibility should also be explored.

In section 5.3.4, we presented a simplistic analysis of the scalability of central-server-based Web Service intermediaries and PI. An in-depth analysis should be conducted for more thorough results.

Our implementation of PI is a simple prototype that should not be used for enterprise Web Service applications. A full-featured, industrial-strength implementation of PI should be developed. A thorough practical evaluation of PI with a real enterprise Web Service application would then be possible.

We have not explored the use of naming and directory services with PI. Naming and directory services are beneficial when the location of an application changes. Instead

of being manually updated, other applications with the help of naming and directory services, can programmatically rediscover the new location of the application. Use of naming and directory services with PI should be examined, with particular attention to the trade-offs between ease of PI proxy maintenance and quality-of-service guarantees.

Bibliography

Agrawal2001vinci — Rakesh Agrawal, Roberto Bayardo, Daniel Gruhl and Spiros Papadimitriou, "Vinci: A Service-Oriented Architecture for Rapid Development of Web Applications", *Proceedings of the Tenth International World Wide Web Conference on World Wide Web (ACM)*, Pages: 355 – 365, 2001.

Akkiraju2001ebusiness — Rama Akkiraju, David Flaxer, Henry Chang, Tian Chao, Liang-Jie Zhang, Frederick Wu and Jun-Jang Jeng, "A Framework for Facilitating Dynamic e-Business via Web Services", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Andrade2001coordination — Luis Filipe Andrade and Jose Luiz Fiadeiro, "Coordination Technologies for Web-Services", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Armstrong2002introxml — Eric Armstrong, "The Java Web Services Tutorial – Introduction to XML", *http://java.sun.com/webservices /docs/1.0/tutorial/doc/IntroXML2.html#64813*, 2002.

Atkinson2002wssecurity — Bob Atkinson, "Web Services Security (WS-Security)", *IBM developerWorks, http://www-106.ibm.com /developerworks/library/ws-secure/*, April 2002.

Barrett1999intermediaries — R. Barrett, "Intermediaries: An approach to manipulating information streams", *IBM Systems Journal*, April 1999.

Bodoff2002war — Stephanie Bodoff, "Web Application Archives (WAR)", *http://java.sun.com/webservices/docs/1.0/tutorial/doc/Web App3.html#64585*, 2002.

Bonifati2001reactive — Angela Bonifati, Stefano Ceri and Stefano Paraboschi, "Pushing Reactive Services to XML Repositories using Active Rules", *Proceedings of the Tenth International World Wide Web Conference on World Wide Web (ACM)*, Pages: 633 – 641, 2001.

Bosworth2001webservices Adam Bosworth, “Developing Web Services”, *Proceedings of the IEEE 17th International Conference on Data Engineering*, Pages: 477 – 481, 2001.

Box2002webservices Don Box, “The Continuing Challenges of XML Web Services”, *MSDN Magazine*, February 2002.

Buchmann2001crypt Johannes A. Buchmann, “Introduction to Cryptography”, *Springer*, 2001.

Burbeck2000wstao Steve Burbeck, “The Tao of e-business services”, *IBM developerWorks*, October 2000.

Cable2001J2EEwebservices Laurence P. G. Cable, “Creating Web Services with J2EE: iPlanet and Sun ONE”, *JavaOne Conference*, 2001.

Cardellini2001performance Valeria Cardellini, Emiliano Casalicchio and Michele Colajanni, “A Performance Study of Distributed Architectures for the Quality of Web Services”, *Proceedings of the 34th Hawaii International Conference on System Sciences (IEEE)*, Pages: 3551 – 3560, 2001.

Chaudhury2001webchannels Abhijit Chaudhury, Debasish N. Mallick and H. Raghav Rao, “Web Channels in E-Commerce”, *Communications of the ACM*, January 2001.

Chen2000discovery Harry Chen, Dipanjan Chakraborty, Liang Xu, Anupam Joshi and Tim Finin, “Service Discovery in the Future Electronic Market”, *Workshop on Knowledge Based Electronic Markets*, 2000.

Chester2001crossplatform Timothy M. Chester, “Cross-Platform Integration with XML and SOAP”, *IEEE IT Pro*, September – October 2001.

Coulouris2001distributed George Coulouris, Jean Dollimore and Tim Kindberg, “Distributed Systems: Concepts and Design”, *Addison-Wesley*, 2001.

Curbera2001integration Francisco Curbera, Ignacio Silva-Lepe and Sanjiva Weerawarana, “On the Integration of Heterogeneous Web Service Partners”, *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Curbera2001webservices — Francisco Curbera, William A. Nagy and Sanjiva Weerawarana, "Web Services: Why and How", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Damianiand2001access — Ernesto Damianiand, Sabrina De Capitani di, Stefano Paraboschi and Pierangela Samarati, "Fine Grained Access Control for SOAP E-Services", *Proceedings of the Tenth International World Wide Web Conference on World Wide Web (ACM)*, Pages: 504 – 513, 2001.

Davis2002soap — Dan Davis and Manish Parashar, "Latency Performance of SOAP Implementations", *Proceedings of the IEEE International Symposium on Cluster Computing and the Grid*, Pages: 377 – 382, 2002.

Drexel2002bannerweb — Drexel University, "BannerWeb", *http://bannerweb.drexel.edu/*, 2002.

Duftler2001invocation — Matthew J. Duftler, Nirmal K. Mukhi, Aleksander Slominski and Sanjiva Weerawarana, "Web Services Invocation Framework (WSIF)", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Dumas2001semantic — Marlon Dumas, Justin O'Sullivan, Mitra Heravizadeh, David Edmond and Arthur ter Hofstede, "Towards a Semantic Framework for Service Description", *IFIP Conference on Database Semantics*, 2001.

EvansData2002evansdata — Evans Data Corporation, "Evans Data Corporation", *http://www.evansdata.com/*, 2002.

Flamenco2002flamenco — Flamenco Networks, "Flamenco Networks", *http://www.flamenconetworks.com/*, 2002.

Flamenco2002wsnetworks — Flamenco Networks, "Web Services and the Need for Web Services Networks", *Flamenco Networks*, *https://www.flamenconetworks.net/Pages/Help/Web_Services_White_Paper.pdf*, 2002.

Fontana2002security — John Fontana, "Top Web services worry: Security", *Network World*, *http://www.nwfusion.com/news/2002/0121webservices.html*, January 2002.

Gisolfi2001corba — Dan Gisolfi, "Is Web Services the reincarnation of CORBA?", *IBM developerWorks, http://www-106.ibm.com/developerworks/webservices/library/ws-arc3/,* 2001.

Gokhale2002corba — Aniruddha Gokhale, Bharat Kumar and Arnaud Sahuguet, "Reinventing the Wheel? CORBA vs. Web Services", *http://www2002.org/CDROM/alternate/395/*, 2002.

GrandCentral2002faq — Grand Central Communications, "Grand Central Communications Frequently Asked Questions (FAQ)", *http://www.grandcentral.com/services/faq.htm,* 2002.

GrandCentral2002gc — Grand Central Communications, "Grand Central Communications", *http://www.grandcentral.com/,* 2002.

GrandCentral2002gcservice — Grand Central Communications, "Enabling the Extended Enterprise", *Grand Central Communications, http://www.grandcentral.com/pdf/Service_Brochure.pdf,* 2002.

Hailpern2001webservices — Brent Hailpern and Peri L. Tarr, "Software Engineering for Web Services: A Focus on Separation of Concerns", *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Helal2002enterprise — Sumi Helal, Stanley Su, Jie Meng, Raja Krithivasan and Arun Jagatheesan, "The Internet Enterprise", *Proceedings of the Second IEEE/IPSJ Symposium on Applications and* the Internet, Pages: 54 – 62, 2002.

Hoffman2001parnas — Daniel M. Hoffman and David M. Weiss, "Software Fundamentals – Collected Papers by David L. Parnas", *Addison-Wesley*, 2001.

IBM2002websphere — IBM, "WebSphere Software Platform", *http://www.ibm.com/websphere/,* 2002.

iDTwo2002pki — iD2 Technologies, "Public Key Infrastructure (PKI)", *http://www.id2tech.com/,* 2002.

Infoworld2002wsjavastyle — Maggie Biggs, "Web services Java style", *InfoWorld*, *http://www.infoworld.com/,* February 2002.

InterKnowlogy2002ik — InterKnowlogy, "InterKnowlogy", *http://www. interknowlogy.com/*, 2002.

Irani2001intermediaries — Romin Irani, "Web Services Intermediaries", *Web Services Architect, http://www.webservicesarchitect.com/*, November 2001.

Ives2001xmldata — Zachary G. Ives, "Integrating Network-Bound XML Data", *IEEE Data Engineering Bulletin*, June 2001.

Januszewski2001uddi — Karsten Januszewski, "Web Service Description and Discovery Using UDDI Part I", *Microsoft Developer Network*, October 2001.

Jepsen2001interoperability — Tom Jepsen, "SOAP Cleans up Interoperability Problems on the Web", *IEEE IT Pro*, January – February 2001.

Johnston2002webservices — Stuart J. Johnston, "State of Web services", *InfoWorld, http://www.infoworld.com/*, February 2002.

Jones2002enterpriseclassws — Mark Jones, "Implementing Enterprise-class Web services", *http://www.xml.org/xml/presentations /implementing_web_services.pdf*, 2002.

Kao2001webservices — James Kao, "Developer's Guide to Building XML-based Web Services with the Java 2 Platform Enterprise Edition (J2EE)", *http://www.theserverside.com/resources /article.jsp?l=WebServices-Dev-Guide*, 2001.

Karvonen2000simplicity — Kristiina Karvonen, "The Beauty of Simplicity", *Proceedings of the Conference on Universal Usability (ACM)*, Pages: 85 – 90, 2000.

Kazakos2001coastbase — Wassili Kazakos, Andreas Schmidt and Heiko Paoli, "XML based Virtual Catalogue Module in CoastBase", *15th International Symposium Informatics for Environmental Protection*, 2001.

Keating2002thirdparty — Michael Keating, "Third-Party Dependence – A Potential Disaster Domino", *http://www.manufacturingweek.com /manufacturingweek/show/schedet.asp?row=33#top*, 2002.

Kirda2001experiences — Engin Kirda, Mehdi Jazayeri, Clemens Kerer and Markus Schranz, "Experiences in Engineering Flexible Web Services", *IEEE Multimedia*, January – March 2001.

Kirda2001multidevice — Engin Kirda, Clemens Kerer, Mehdi Jazayeri and Christopher Kruegel, "Supporting Multi-Device Enabled Web Services: Challenges and Open Problems", *Proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Pages: 49 – 54, 2001.

Kirtland2000web — Mary Kirtland, "The Programmable Web: Web Services Provides Building Blocks for the Microsoft .NET Framework", *MSDN Magazine*, September 2000.

Kirtland2001secure — Mary Kirtland, "Secure Web Services Using the SOAP Toolkit", *Microsoft Developer Network*, January 2001.

Klein2001semantic — Mark Klein and Abraham Bernstein, "Searching for Services on the Semantic Web Using Process Ontologies", *The First Semantic Web Working Symposium*, 2001.

Larsen2000costmodel — Kai R.T. Larsen and Peter A. Bloniarz, "A Cost and Performance Model for Web Service Investment", *Communications of the ACM*, February 2000.

LeonGarcia2000networks — Alberto Leon-Garcia and Indra Widjaja, "Communication Networks – Fundamental Concepts and Key Architectures", *McGraw-Hill*, 2000.

Mani2002qos — Anbazhagan Mani and Arun Nagarajan, "Understanding quality of service for Web services", *IBM developerWorks*, January 2002.

McIlraith2001semantic — Sheila A. McIlraith, Tran Cao Son and Honglei Zeng, "Semantic Web Services", *IEEE Intelligent Systems*, March – April 2001.

Mckee1999directions — P. F. McKee, I. W Marshall and I. D. Henning, "Research Directions in Distributed Systems", *BT Technol*, April 1999.

Mckee2001uddi — Barbara McKee, Dave Ehnebuske and Dan Rogers, "UDDI Version 2.0 API Specification", *http:/www.uddi.org/*, 2001.

Microsoft2002dcom — Microsoft, "Distributed Component Object Model (DCOM)", *http://www.microsoft.com/com/tech/DCOM.asp*, 2002.

Microsoft2002dotnet — Microsoft, “Microsoft .NET”, *http://www.microsoft.com /net/*, 2002.

Microsoft2002myservices — Microsoft, “Microsoft .NET Services”, *http://www. microsoft.com/myservices/*, 2002.

Microsoft2002passport — Microsoft, “Microsoft .NET Passport”, *http://www. passport.com/Consumer/Default.asp?lc=1033*, 2002.

Microsoft2002security — Microsoft Corporation, “XML Web Services Security”, *http://msdn.microsoft.com/vstudio/techinfo/articles/XMLwebservices/security.asp*, 2002.

Microsoft2002webservices — Microsoft Corporation, “What are XML Web Services?”, *http://www.microsoft.com/net/basics/xmlservices.asp*, 2002.

OConnell2001configuration — Paul O'Connell and Rachel McCrindle, “Using SOAP to Clean Up Configuration Management”, *Proceedings of the IEEE 25th Annual International Computer Software and Applications Conference*, Pages: 555 – 560, 2001.

OMG2002corba — Object Management Group (OMG), “Common Object Request Broker Architecture (CORBA)”, *http://www.corba.org/*, 2002.

Piccinelli2001ebusiness — Giacomo Piccinelli, Mathias Salle and Christian Zirpins, “Service-oriented Modeling for e-Business Applications Components”, *Proceedings of the Tenth IEEE International Workshops on Enabling Technologies: Infrastructure for Collaborative Enterprises*, Pages: 12 – 17, 2001.

Piccinelli2001service — Giacomo Piccinelli and Eric Stammers, “From E-Processes to E-Networks: an E-Service-oriented approach”, *OOPSLA 2001 Workshop on Object-Oriented Web Services*, 2001.

Powell2001architecture — Matt Powell, “Physical Architecture (of Web Services)”, *Microsoft Developer Network*, June 2001.

Powell2001security — Matt Powell, “Real SOAP Security”, *Microsoft Developer Network*, November 2001.

Powell2002wsdl — Matt Powell, “Building XML Web Services Using Industry Standardized WSDLs”, *Microsoft Developer Network*, February 2002.

Richter2000microsoft — Jeffrey Richter, "Microsoft .NET Framework Delivers the Platform for an Integrated Service-Oriented Web", *MSDN Magazine*, October 2000.

Roy2001web — Jaideep Roy and Anupama Ramanujan, "Understanding Web Services", *IEEE IT Pro*, November – December 2001.

RSA2001scalability — RSA Security, "Scalability Proof of Concept: RSA Keon Certificate Authority Eight Million Certificate Test", *http://www.rsasecurity.com/*, 2001.

RSA2002keon — RSA Security, "RSA Keon", *http://www.rsasecurity.com/products/keon/*, 2002.

RSA2002rsa — RSA Security, "RSA Security", *http://www.rsasecurity.com/*, 2002.

Ruiz2001quality — Antonio Ruiz, Rafael Corchuelo, Amador Duran and Miguel Toro, "Automated Support for Quality Requirements in Web-Service-Based Systems", *Proceedings of the Eighth IEEE Workshop on Future Trends of Distributed Computing Systems*, Pages: 48 – 54, 2001.

Russell2001web — James R. Russell, "Web Services and Java Technology - A Natural Fit", *JavaOne Conference*, 2001.

Saaresto2001xmlapi — Marko Saaresto, "A Review on Java API for XML Messaging (JAXM)", *Research Seminar on Advanced Java Specifications*, 2001.

Samulowitz2001pervasive — Michael Samulowitz, Florian Michahelles and Claudia Linnhoff-Popien, "Adaptive Interaction for Enabling Pervasive Services", *Proceedings of the Second ACM International Workshop on Data Engineering for Wireless and Mobile Access*, Pages: 20 – 26, 2001.

Schwartz2002deployability — Jeff Schwartz, "Deployability: The New Business Frontier", *http://www.gantthead.com/article/1,1380,94592,00.html*, 2002.

Seebeyond2001webservices — Seebeyond, "Web Services: Sharing Business Processes over the Internet", *http://www.seebeyond.com/products/whitepapers/WebServicesWhitepaper.pdf*, 2001.

Seely2001documenting	Scott Seely, "Documenting Your Web Service", *Microsoft Developer Network*, July 2001.
Seely2001interoperability	Scott Seely, "Designing Your Web Service for Maximum Interoperability", *Microsoft Developer Network*, December 2001.
Seely2001uddi	Scott Seely, "Web Service Description and Discovery Using UDDI Part II", *Microsoft Developer Network*, October 2001.
Seely2001xml	Scott Seely, "An XML Overview towards Understanding SOAP", *Microsoft Developer Network*, November 2001.
Seely2002standardWSDL	Scott Seely, "Building Industry Standard WSDL", *Microsoft Developer Network*, February 2002.
Stonebraker2001ebusiness	Michael Stonebraker and Joseph M. Hellerstein, "Content Integration for E-Business", *Proceedings of the 2001 ACM SIGMOD International Conference on Management of Data*, Pages: 552 – 560, 2001.
Sun1999J2EEguide	Sun Microsystems Inc, "Simplified Guide to the Java 2 Platform Enterprise Edition", *http://www.serverworldmagazine.com/webpapers/2000/07_sunj2ee.shtml*, 1999.
Sun2002iforce	Sun Microsystems, "iForce Ready Centers", *http://www.sun.com/executives/iforce/readycenters/sunmatrix.html*, 2002.
Sun2002j2ee	Sun Microsystems, "Java 2 Platform, Enterprise Edition (J2EE)", *http://java.sun.com/j2ee/*, 2002.
Sun2002j2eeFAQ	Sun Microsystems Inc, "Java 2 Platform Enterprise Edition (J2EE) FAQ", *http://java.sun.com/j2ee/faq.html*, 2001.
Sun2002j2se	Sun Microsystems, "Java 2 Platform, Standard Edition (J2SE)", *http://java.sun.com/j2se/*, 2002.
Sun2002javaFAQ	Sun Microsystems Inc, "Java Technology & Web Services FAQ", *http://java.sun.com/webservices/faq.html*, 2002.
Sun2002javaidl	Sun Microsystems, "Java Interface Description Language (IDL)", *http://java.sun.com/products/jdk/idl/*, 2002.

Sun2002javarmi — Sun Microsystems, “Java Remote Method Invocation (RMI)”, *http://java.sun.com/products/jdk/rmi/*, 2002.

Sun2002jwsdp — Sun Microsystems, “Java Web Services Developer Pack”, *http://java.sun.com/webservices/webservicespack.html*, 2002.

Sun2002sunone — Sun Microsystems, “Sun Open Net Environment (Sun ONE)”, *http://wwws.sun.com/software/sunone/*, 2002.

Sun2002webservices — Sun Microsystems Inc, “Initial Thoughts on Web Services”, *http://java.sun.com/blueprints/guidelines /WS_article.html*, 2002.

Sun2002xmlFAQ — Sun Microsystems Inc, “Java Technology & XML FAQ”, *http://java.sun.com/xml/faq.html*, 2002.

Tapang2001wsdl — Carlos C. Tapang, “Web Services Description Language (WSDL) Explained”, *Microsoft Developer Network*, July 2001.

Tauber2001webservices — James Tauber and Andy Roberts, “The Web Services Revolution”, *JavaOne Conference*, 2001.

Todd2002httpr — Stephen Todd, “A Primer for HTTPR”, *IBM developerWorks*, April 2002.

Tominaga2002gcflamenco — Mana Tominaga, “Secure infrastructures”, *NewArchitect Magazine*, *http://www.newarchitectmag.com/*, April 2002.

Trastour2001semantic — David Trastour, Claudio Bartolini and Javier Gonzalez-Castillo, “A Semantic Web Approach to Service Description for Matchmaking of Services”, *Proceedings of the International Semantic Web Working Symposium*, Pages: 447 – 461, 2001.

Treese2001data — Win Treese, “It’s Ten O’clock - Do You Know Where Your Data Are?”, *ACM netWorker*, December 2001.

Tsur2001revolution — Shalom Tsur, “Are Web Services the Next Revolution in E-Commerce?”, *Proceedings of the 27th International Conference on Very Large Data Bases*, Pages: 614 – 617, 2001.

UDDI2002tech	UDDI Community, “UDDI Technical White Paper”, *http://www.uddi.org/pubs/Iru_UDDI_Technical_White_Paper.pdf*, 2002.
UDDI2002uddi	UDDI Community, “Universal Description Discovery Integration (UDDI)”, *http://www.uddi.org/*, 2002.
Vawter2001J2EEvsNet	Chad Vawter and Ed Roman, “J2EE vs. Microsoft.NET - A comparison of building XML-based web services”, *http://www.theserverside.com/resources/article.jsp?l=J2EE-vs-DOTNET*, 2001.
VeriSign2000services	VeriSign, “VeriSign Authentication Services”, *http://www.verisign.com/*, 2000.
VeriSign2002verisign	VeriSign, “VeriSign”, *http://www.verisign.com/*, 2002.
W3C2002http	W3C, “Hypertext Transfer Protocol (HTTP)”, *http://www.w3.org/Protocols/*, 2002.
W3C2002soap	W3C, “Simple Object Access Protocol (SOAP)”, *http://www.w3.org/TR/SOAP/*, 2002.
W3C2002wsdl	W3C, “Web Services Description Language (WSDL)”, *http://www.w3.org/TR/wsdl*, 2002.
W3C2002xml	W3C, “Extensible Markup Language (XML)”, *http://www.w3.org/XML/*, 2002.
Waterhouse2001wssecurity	Mark Waterhouse, “Establishing Trust in Web Services”, *Web Services Architect*, *http://www.webservicesarchitect.com/*, October 2001.
Weiss2001microsoft	Aaron Weiss, “Microsoft .NET: Platform in the Clouds”, *ACM netWorker*, December 2001.
Wolter2001wsbasics	Roger Wolter, “XML Web Services Basics”, *Microsoft Developer Network*, December 2001.
WSorg2002wsoverview	Kevin Jones, “Web Services Overview and How They Fit into Your Future”, *http://www.webservices.org/article.php?sid=444*, 2002.

Zou2001migration Ying Zou and Kostas Kontogiannis, “Towards a Web-centric Legacy System Migration Framework”, *Proceedings of the ICSE 2001 Workshops of 3rd International Workshop on Net-Centric Computing: Migrating to the Web*, Pages: 70 – 74, 2001.

Appendix A: Glossary of Acronyms

API	Application Programming Interface
CLR	Common Language Runtime
CORBA	Common Object Request Broker Architecture
DCOM	Distributed Component Object Model
FAQ	Frequently Asked Questions
FDDI	Fiber Distributed Data Interface
FTP	File Transfer Protocol
GUI	Graphical User Interface
HTML	HyperText Markup Language
HTTP	HyperText Transfer Protocol
ID	Identification
IDE	Integrated Development Environment
IDL	Interface Definition Language
IP	Internet Protocol
J2EE	Java 2 Platform, Enterprise Edition
J2SE	Java 2 Platform, Standard Edition
JAXM	Java API for XML Messaging
JAXP	Java API for XML Processing
JAXR	Java API for XML Registries
JAX-RPC	Java API for XML-based RPC
JWSDP	Java Web Services Developer Pack
LAN	Local Area Network
ORB	Object Request Broker
PI	Peer-to-peer Intermediary
PIN	Personal Identification Number
PI-v1	PI Proxy Version 1.0
QoS	Quality-of-Service
RMI	Remote Method Invocation
RPC	Remote Procedure Call
SDK	Software Development Kit
SMTP	Simple Mail Transfer Protocol
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
UML	Unified Modeling Language
URL	Uniform Resource Locator
WAR	Web Application Archive
WSDL	Web Services Definition/Description Language
WWW	World Wide Web
XML	Extensible Markup Language

Appendix B: Platforms and Tools Used for the Implementation and Evaluation of PI Proxy Version 1.0

Platform / Tool	Function
Microsoft Windows 2000	Operating System used for development and evaluation
Sun Solaris 7	Operating System used for evaluation
Java 2 Platform, Standard Edition (J2SE) v1.4.0	Software Development Kit for building Java applications
Java Web Services Developer Pack (JWSDP) v1.0	Software Development Kit for building Web Service applications. JWSDP is used in conjunction with J2SE.
Java API for XML Messaging (JAXM) v1.1	API for creating and sending XML messages over the Internet. The API is part of JWSDP.
Forte for Java 4.0 CE	Integrated Development Environment for Java
Apache Tomcat 4.1.2 Container	Web Server and Web Application Container