# A DEPLOYMENT-READY SOLUTION FOR ADDING QUALITY-OF-SERVICE FEATURES TO WEB SERVICES

*O. Hasan*
*Department of Computer Science, Drexel University, Philadelphia, PA 19104, USA*
*B.W. Char*
*Department of Computer Science, Drexel University, Philadelphia, PA 19104, USA*
*Correspondence Email: oh23@drexel.edu*

## ABSTRACT

Web Services is a recent technology that enables an application to interact with a remote application over the Internet and use its functionality. The novelty of the Web Services technology is that it allows heterogeneous applications to interact with each other regardless of their development and operational platforms. Quality-of-Service (QoS) features such as security, reliability and manageability are vital to enterprise-class applications. Several solutions exist to achieve QoS in Web Services, one of which is Web Service intermediaries. The advantage of Web service intermediaries over other solutions is that it takes most of the load of implementing QoS in Web Services off their developers. However, current implementations of the Web Service intermediary solution have some undesirable constraints, which include: 1) the requirement of recoding to enable a Web Service application to interact with the intermediary, and 2) the need for a central server in the intermediary architecture. Recoding a Web service application is a time and effort consuming task. A central server is not suitable when centralization of control in a single authority is undesirable. We present a Web Service intermediary based on the peer-to-peer architecture which does not suffer from these problems. The solution does not require recoding of a Web Service application. Due to the peer-to-peer architecture, it also does not require a central server.

**Keywords:** Distributed Systems, Web Services, Quality-of-Service, Intermediaries, Peer-to-Peer.

## 1 INTRODUCTION

Web Services is a recent technology that provides the means to make the functionality of an application available over the Internet such that it can be used by remote applications. For example with the Web Services technology it is possible for a company's inventory application to access a supplier's sales application and place an order for depleted supplies. A key advantage of the Web Services technology over other similar technologies (for example, CORBA [1], Java RMI [2]) is that it enables heterogeneous applications to interact with each other over the Internet regardless of their development and operational platforms. This is possible because the Web Services technology is based on XML [3]. XML is a universally accepted and platform independent standard for information representation. The Web Services technology has several other novel features which include the following:

- Web Service applications can be published on the Internet with special registries based on a protocol called UDDI [4]. Clients can search and locate suitable Web Services (we refer to Web Service applications simply as Web Services) from these registries. For example a client could be looking for a Web Service that performs a complex 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.
- 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 a credit card processing Web Service and a shipping information processing Web Service as its modules. The credit card processing Web Service can itself use other Web Services such as a customer credit rating Web Service. This software architecture is known as the Service Oriented Architecture (SOA) [5].

Microsoft's .NET [6] and Sun's J2EE [7] / Sun ONE [8] are major commercial implementations of the Web Services technology.

Although The Web Services technology offers several benefits, achieving Quality-of-Service (QoS) features such as security, reliability and manageability in Web Services has been an issue since its inception. QoS is very important for enterprise-class applications (applications used by enterprises for their operation). For example a bank making a transaction must be absolutely certain that the system is secure and reliable. A Web Service that is able to meet an enterprise's QoS requirements can be termed as an enterprise-class Web Service.

Following are some reasons why it is difficult to achieve QoS in Web Services:

- The Web Services message delivery protocol, SOAP, relies primarily on the Hyper Text Transfer Protocol (HTTP) for transport of its messages. HTTP is a best-effort delivery service [9]; it does not guarantee 1) delivery of a message to its destination, 2) delivery of messages in the correct order or 3) that a message will be delivered only once.
- Web Services communicate over the public Internet. This means that all communication is visible to the public and vulnerable to a myriad of security threats.

Various protocols and technologies exist for achieving QoS in Web Services, such as WS-S [10], WS-R [11]. Web service intermediaries are another solution for this purpose. In this solution an intermediary is placed between the communicating Web Services. The Web Services then communicate through this intermediary instead of communicating directly. The intermediary takes the responsibility of enforcing the QoS requirements. One reason this solution is of good value is that a significant portion of the code that is concerned with handling QoS is implemented in the Web service intermediary. The developers of a Web service can therefore focus more on implementing its functionality and worry less about implementing QoS. However, the existing implementations of this solution have some drawbacks:

- Some implementations of the Web Service intermediary solution require that the Web Service applications be recoded to be able to communicate with the intermediary.
- The existing implementations of the Web Service intermediary solution are based on central-server architecture. This architecture places control of the information flow and the configuration of QoS requirements with a single party.

In this paper we present a Web service intermediary which does not pose these problems.

## 2 WEB SERVICE INTERMEDIARIES

An obvious solution to achieving QoS in a Web service is to hard code the desired QoS features into it at development time. This can be done using the many protocols and technologies available for this purpose, such as WS-S [10], WS-R [11]. This approach however has one drawback. Manually implementing QoS features into a Web Service is a time and effort consuming task.

An alternative solution is Web service intermediaries. The advantage of this solution is that developers are not required to manually implement QoS handling mechanisms into a Web Service, instead they can rely on a Web Service intermediary to take care of QoS.

As defined by [12]:

> "An intermediary is a computational element that lies between an information producer and an information consumer on an information stream … Intermediaries can turn ordinary information streams into smart streams that enhance the quality of communication"

A Web Service intermediary is an intermediary placed between two Web Services. An intermediary may not be a single entity. It may be composed of a system of entities.

### 2.1 How does a Web Service Intermediary Work?

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

The only channel that the provider and the client can communicate on directly is the Internet, which does not meet the QoS 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 QoS requirements.

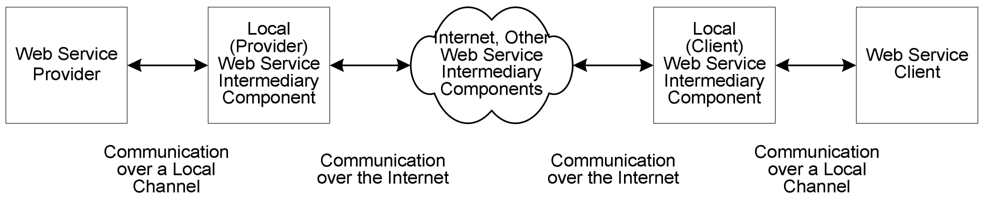
Web Service Provider
Local (Provider) Web Service Intermediary Component
Internet, Other Web Service Intermediary Components
Local (Client) Web Service Intermediary Component
Web Service Client
Communication over a Local Channel
Communication over the Internet
Communication over the Internet
Communication over a Local Channel

**Figure 1:** 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 components of the Web Service intermediary take measures to make their interaction over the Internet meet the QoS 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.

## 2.2 Some Implementations Based on the Concept of Web Service Intermediary

### 2.2.1 Grand Central's Web Service Intermediary

Grand Central [13] hosts a central server at its site. Web Services that wish to communicate must go through this server to exchange messages. The Web Service provider and the Web Service 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 QoS.

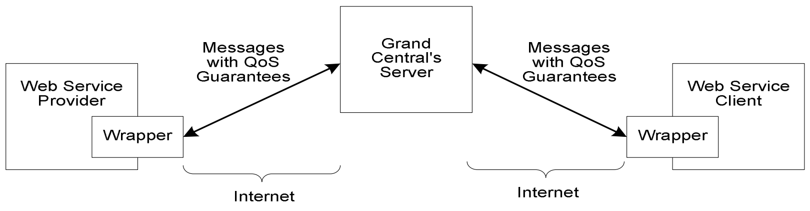

**Figure 2:** Architecture of Grand Central's Web service intermediary.

### 2.2.2 Flamenco Networks' Web Service Intermediary

Flamenco Networks' [14] Web Service intermediary is a hybrid of client-server and peer-to-peer models. A proxy is installed on each communicating party's trusted network. The Web Service provider and the Web Service client communicate only with the proxy installed on their own trusted network. When a proxy receives a message from the local Web Service 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 QoS 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 server plays an indispensable role in Flamenco's Web Service Intermediary.

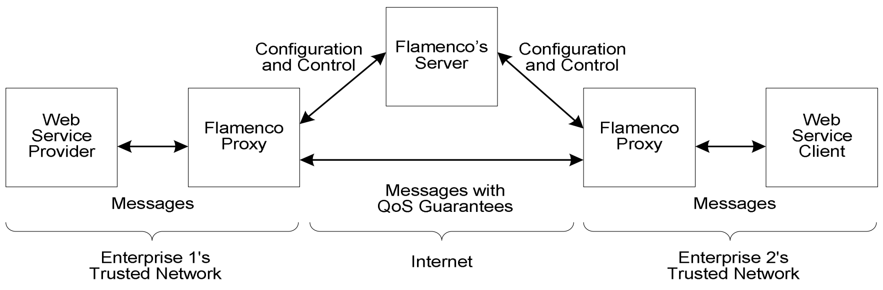
Configuration and Control
Flamenco's Server
Configuration and Control
Web Service Provider
Flamenco Proxy
Flamenco Proxy
Web Service Client
Messages
Messages with QoS Guarantees
Messages
Enterprise 1's Trusted Network
Internet
Enterprise 2's Trusted Network

**Figure 3:** Architecture of Flamenco Networks' Web service intermediary.

### 2.3 Problems in Existing Web Service Intermediaries

#### 2.3.1 Requirement of Customization through Additional Coding

Grand Central requires each Web Service to be customized by hard coding for it to be able to use its Web Service intermediary. According to Grand Central [13]:

> "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."

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.

David L. Parnas in [15] highlights several risks involved in modification of software in general and especially by people who do not thoroughly understand the original design of the software.

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

#### 2.3.2 An Enterprise's Control of the Flow and Management of its Information

In case of both Grand Central and Flamenco 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 a third-party for the flow and management of their 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 [16]:

> "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 Services 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.

## 3 A PEER-TO-PEER WEB SERVICE INTERMEDIARY (PI)

We have designed a peer-to-peer Web Service intermediary, such that it meets the following objectives:

- 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)

The solution is termed Peer-to-peer Intermediary, abbreviated as PI (pronounced as π). In this section we discuss a high level design of the solution. Implementation level details are discussed in the next section.

### 3.1 Architecture

The components of the PI architecture are the Web Service provider, the Web service client and a pair of PI proxies. A PI proxy is a software program whose basic function is to relay messages between its local Web Service and its peer PI proxy. For each Web Service, a copy of the PI proxy is installed on its local trusted network. The Web Service provider and the Web Service client each interact only with its local PI proxy. It is assumed that this internal interaction on the local trusted network already meets the enterprise's QoS requirements. The two peer PI proxies interact over the unreliable and insecure Internet but take measures to make this interaction conform to the QoS requirements.

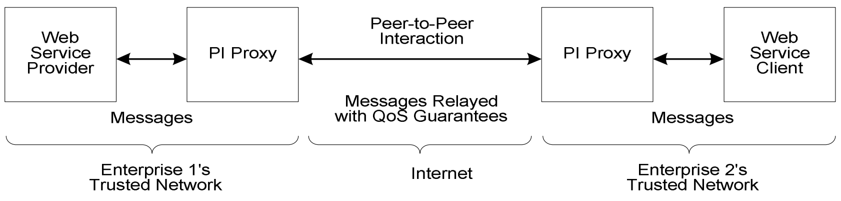

**Figure 4:** PI's architecture.

#### 3.1.1 Trusted Network

We define a trusted network as a computer environment which provides a communication channel between a Web Service and a PI proxy that meets the QoS requirements of the owner enterprise. The QoS requirements that we consider here are reliability and security from external threats.

Popular LANs such as Ethernet 100BaseT, 100BaseF, 1000BaseT and FDDI can serve as trusted networks since they provide reliable communication if designed and administrated properly. 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.

### 3.2 Message Exchange Model

The basic function of a PI proxy is to relay messages between its local Web Service application and its peer PI proxy. To perform this function, each PI proxy is provided with the address of the local application and the address 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 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/. 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 'weather', it would send the message to an address such as http://piproxyaddress/weather. The PI proxy upon receiving the message would forward it to the peer PI proxy identified by the ID 'weather'.

The steps involved in delivering a message from one Web Service 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 QoS requirements. Since the two PI proxies are independently configured, the QoS requirements are negotiated before messages are transmitted.

## 3.3 How does PI Meet the Objectives?

### 3.3.1 Requirement of Customization through Additional Coding

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 communicates with the local PI proxy instead of the other 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.

### 3.3.2 An Enterprise's Control of the Flow and Management of its Information

This problem is resolved in PI primarily due to its peer-to-peer architecture. Each enterprise has a PI proxy hosted on its own network for each of its Web Service applications. Since each enterprise is in full control of its PI proxy, it can define its own QoS requirements. When two PI proxies communicate, they negotiate as to the strictness of QoS that they will implement.

### 3.4 Quality-of-Service Features Provided to Web Services by PI

PI uses standard algorithms to implement the QoS features. PI has been designed to implement the following QoS features:

- Security: Authentication, Authorization and Access Control, Confidentiality and Non-repudiation
- Reliability: Guaranteed Message Delivery, Ordered Delivery and Exactly-Once Delivery
- Manageability: Monitoring, Logging and Client Management

All these features are customizable both at the Web Service provider and the Web Service client end. The measures taken by the PI proxies to achieve the QoS requirements are invisible to the Web Services using them.

How the requirements of exactly-once delivery are met is discussed from a design perspective in the following subsection. Other QoS requirements are also fulfilled using standard algorithms (please see [17] for descriptions).

#### 3.4.1 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 or number of messages. 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.

## 4 IMPLEMENTATION

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. We have used the Tomcat Web application container to host PI-v1. PI-v1 is written in Java (JWSDP, JAXM APIs).

PI-v1 has three servlets: system, local_relay and 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 servlets local_relay and remote_relay provide the function of relaying messages between two Web Service applications. These servlets also implement the QoS features provided by PI-v1.

PI-v1 has the limitation that it can be configured for communication with only one remote Web Service application. 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.

### 4.1 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 the following figure. If WS-2 sends a request message to WS-1, the exact same process occurs but in the opposite direction.

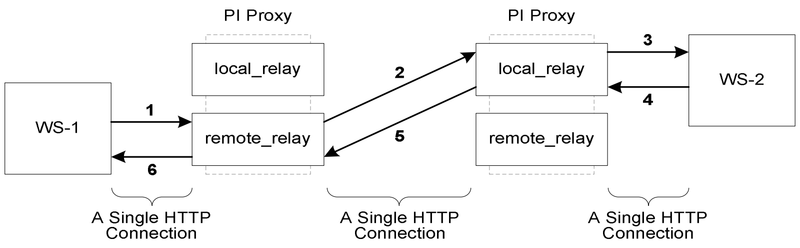

**Figure 5:** Basic message relay.

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.

### 4.2 Implementation of Quality-of-Service Features

The implementation of exactly-once delivery is discussed in the following subsection. Implementations of some other QoS features can be found in [17].

#### 4.2.1 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:

| | | |
|---|---|---|
| IP/DNS address of the sending PI proxy | Send time with millisecond precision | The value of a counter which is incremented by 1 for each new message ID |

**Figure 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.

### 4.3 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. The tool interacts with the servlet system of a PI proxy. The tool has a GUI that has been developed using the Java Swing API.

## 5 EVALUATION

### 5.1 Does PI Proxy Version 1.0 Achieve the Design Objectives of PI?

In section 3.3 we theoretically demonstrated that PI meets its design objectives. In this section we give practical demonstrations to substantiate our claim. PI-v1 and our simple banking web service application are used for this purpose.

#### 5.1.1 Requirement of Customization through Additional Coding

PI-v1 is a Java Web Application, packaged in a single distributable Web Application Archive (WAR) [18] file: jaxm-pi-proxy.war. 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.

1) Place jaxm-pi-proxy.war in the following directory: <HOME>/webapps, where <HOME> is the directory where Tomcat is installed.
2) Restart Tomcat.
3) Configure the PI proxy through the GUI of the PI proxy administration tool.

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

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

Each enterprise has its own deployment of PI-v1 for its Web Service application. For example in the case of our banking 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

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

We have used the banking Web Service to test the QoS features provided by PI-v1. The test of exactly-once delivery is given in the following subsection. Tests of some other QoS features can be found in [17]. Following is a description of the test environment:

The banking Web Service provider and its PI-v1 (let us call it PI-S) were hosted on a Tomcat container on a Solaris 7 machine. The PI-v1 (let us call it PI-C) of the banking Web Service client was hosted on a Tomcat container on a Windows 2000 machine. The banking Web Service client, which is a self-contained application, was also run on the same machine.

### 5.2.1 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 banking 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 banking Web Service provider. The second duplicate copy was discarded by PI-S's local_relay servlet. Had the exactly-once delivery mechanism not been in place, the banking Web Service provider would have received two redundant copies of the same message. The test was repeated several times with the same outcome.

## CONCLUSION

The contribution of this work is the design of a Web Service intermediary that can be used to add Quality-of-Service features (security, reliability and manageability) to Web Services. It does not suffer from the drawbacks associated with the existing Web Service intermediaries. Web Services that use it do not need to be recoded. It is based on the peer-to-peer architecture therefore control is not centralized in a single authority. Due to these characteristics it makes building enterprise-class Web Services convenient and practical.

## REFERENCES

[1] Object Management Group (OMG), *"Common Object Request Broker Architecture (CORBA)"*, http://www.corba.org/, 2004.
[2] Sun Microsystems, *"Java Remote Method Invocation (RMI)"*, http://java.sun.com/products/jdk/rmi/, 2004.
[3] W3C, *"Extensible Markup Language (XML)"*, http://www.w3.org/XML/, 2004.
[4] UDDI Community, *"Universal Description Discovery Integration (UDDI)"*, http://www.uddi.org/, 2004.
[5] Steve Burbeck, *"The Tao of e-business services"*, IBM developerWorks, October 2000.
[6] Microsoft, *"Microsoft .NET"*, http://www.microsoft.com/net/, 2004.
[7] Sun Microsystems, *"Java 2 Platform, Enterprise Edition (J2EE)"*, http://java.sun.com/j2ee/, 2004.
[8] Sun Microsystems, *"Sun Open Net Environment"*, http://wwws.sun.com/software/sunone/, 2002.
[9] Anbazhagan Mani and Arun Nagarajan, *"Understanding quality of service for Web services"*, IBM developerWorks, January 2002.
[10] OASIS, *"WS-Security"*, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wss, 2004.
[11] OASIS, *"WS-Reliability"*, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=wsrm, 2004.
[12] R. Barrett, *"Intermediaries: An approach to manipulating information streams"*, IBM Systems Journal, April 1999.
[13] Grand Central Communications, *"Grand Central Communications"*, http://www.grandcentral.com/, 2002.
[14] Flamenco Networks, *"Flamenco Networks"*, http://www.flamenconetworks.com/, 2002.
[15] Daniel M. Hoffman and David M. Weiss, *"Software Fundamentals – Collected Papers by David L. Parnas"*, Addison-Wesley, 2001.
[16] Michael Keating, *"Third-Party Dependence – A Potential Disaster Domino"*, http://www.manufacturingweek.com/manufacturingweek/show/schedet.asp?row=33#top, 2002.
[17] Omar Hasan, *"A Peer-to-Peer Intermediary for Building Enterprise-Class Web Services"*, Masters Thesis, Department of Computer Science, Drexel University, PA, USA. 2002.
[18] Stephanie Bodoff, *"Web Application Archives (WAR)"*, http://java.sun.com/webservices/docs/1.0/tutorial/doc/WebApp3.html#64585, 2002.