University of New Orleans Theses and Dissertations   Dissertations and Theses

5-20-2005

Adapting the Single-Request/Multiple-Response Message Exchange Pattern to Web Services

Michael Ruth
University of New Orleans

Follow this and additional works at: https://scholarworks.uno.edu/td

ADAPTING THE SINGLE-REQUEST/MULTIPLE-RESPONSE MESSAGE EXCHANGE PATTERN TO WEB SERVICES

A Thesis

Submitted to the Graduate Faculty of the
University of New Orleans
in partial fulfillment of the
requirements for the degree of

Master of Science
in
The Department of Computer Science

by

Michael Ruth

B.S. University of New Orleans, 2002

May, 2005

This thesis is dedicated to
Rebecca, Anthony,
my family and friends.

Acknowledgement

I would like to thank Dr. Shengru Tu, my advisor, for providing me with the guidance needed to see this research project through to its completion, and of course, not allowing me to give up.

I would also like to thank Dr. Vassil Roussev and Dr. Nauman Chaudhry for being a part of my thesis committee.

I would also like to thank Rebecca, my girlfriend, without whom I would still be wearing white socks with dress pants.

I would like to thank Anthony as well, for providing me a constant source of amusement every minute we are together.

Also, my friends have my gratitude for both keeping me safe, and preventing me from doing anything that may keep me out of office in the wild world of New Orleans nightlife.

Lastly, and most importantly, I would like to thank each and every member of my family for their support and understanding throughout. Without them I would not even have survived long enough to finish this work.

Table of Contents

List of Figures

Abstract

Single-Request/Multiple-Response (SRMR) is an important messaging exchange pattern because it can be used to model many real world problems elegantly. However, SRMR messaging is not directly supported by Web services, and, since it requires Callback to function it is hampered by current in-practice security schemes, such as firewalls and proxy servers. In this thesis, a framework will be proposed to support SRMR and Callback in the context of Web services and the realities of network security. The central component of the proposed solution is a Clearinghouse Web service (CWS), which serves as a communication proxy and realizes the correlation of responses with requests. One and only one CWS will be needed per enterprise that wishes to handle any number of SRMR Web services and their respective clients. Using the framework and related code generation utilities, a non-trivial case study, a Purchase Order System, has been implemented.

Chapter 1: Introduction

Web services have become the de facto means to enable business-to-business (B2B) applications. Web services are interoperable building blocks enabling business process automation and integration across organizational as well as departmental lines [1]. Due to many efforts by researchers in academia and the industry to enhance their functionality, Web services are in the process of outgrowing the synchronous request-response model and emerging as a flexible distributed computing platform in which the asynchronous model, as well as more complicated messaging patterns, such as Single-Request/Multiple-Response messaging, take an important role.

In the synchronous model, when the client calls the server the client is blocked waiting for the result. In contrast, in the asynchronous model the client is not blocked waiting for the result. When the result of the call is produced it will be returned to the client at some later time. Specifically, this asynchronous model is Callback, a fundamental pattern for the realization of asynchronous, loosely-coupled interactions.

The Single-Request/Multiple-Response message exchange pattern (SRMR) refers to message passing in a conversational manner. In the request/response message exchange pattern, there exists a one to one correlation between the request and response. In SRMR messaging, each request may result in many responses. SRMR messaging inherently requires the asynchronous model to operate because after the first response, the rest of the responses have to be asynchronous. Also, message correlation is required, since there are multiple responses to a single request, each of the responses must be correlated to the request that generated it.

In B2B applications, an extra dimension has been added in the realm of distributed computing by the proliferation of ever increasing security measures. In order to realize the asynchronous model, specifically callback, the caller must be accessible from the external service provider. In typical enterprise installations, common security measures such as firewalls and proxy servers often prevent the client from being accessible from the outside. These measures severely handicap any implementation of the asynchronous model.

Recently, Web service composition and choreography, which attempt to support any interaction model [2], have been a strong trend in research and development. Web service composition and choreography assume that every business application is a Web service. However, the communication between asynchronous Web services and their client applications in the context of network security has been largely overlooked. Many business applications need to utilize available Web services, but may not necessarily need to be exposed as Web services themselves. Promoting every application into a Web service may circumvent the firewall barrier, but, such a practice is not feasible in many environments due to security concerns. Most business processes should never be exposed to the outside of the enterprise.

The focus of this thesis is on SRMR messaging in the context of both Web services and the realities of network security. Its importance lies in its applicability to many real-world problems. For example, a purchase order request may result in multiple sales deals, a document request may obtain multiple files, and a large dataset may have to be broken into more manageable pieces. These problems can be modeled elegantly with the SRMR message exchange pattern. In this thesis, the Web services that provide services using the SRMR model are simply called SRMR Web services.

An application-level framework to facilitate the use of the SRMR message exchange pattern will also be proposed in the context of Web services and the current in-practice security measures. This framework uses software design patterns at the following scalability levels: object, system, enterprise, and global [3]. The actual patterns that were used in the design process are Observer, Mediator, Proxy, Memento, Abstract Factory, and Strategy [4], Half-Object Plus Protocol, Router [5], and Gateway [3].

The center piece of this framework is a Clearinghouse Web services (CWS) which serves two distinct roles: (1) a proxy between the service provider Web service and the client application, and more importantly, (2) a message manager that realizes the correlation between the responses coming from the service providers and the clients' requests. In this framework, one and only one CWS will be required for any enterprise that wishes to handle any number of SRMR Web services and their respective clients. In order to do so, the proposed CWS will be capable of handling any type of response from external Web services. The framework consists of the CWS, a set of client-side helper components, and a suite of code generation utilities. Using this framework, and the code generation utilities, a Purchase Order System was implemented to provide an example of the interactions between a client of the framework and the involved Web services.

The remaining part of this thesis is organized as follows: Chapter 2 provides background information regarding Web services, their supporting technologies, and network security. Chapter 3 highlights some of the related approaches that support callback and the SRMR message exchange pattern. Chapter 4 will outline some objectives for an ideal solution, then present both a detailed walkthrough and an architectural overview of the developed framework, including design decisions made in the development of the framework. Chapter 5 will discuss

some of the more important design decisions in depth and implementation details involved in the development process. Chapter 6 will describe a non-trivial case study, a purchase order system, in detail, while providing further details of the framework's development. Chapter 7 will discuss some performance considerations, and finally, Chapter 8 concludes.

Chapter 2: Background

Broadly, Web services refer to self-contained web applications that are loosely coupled, distributed, capable of performing business activities, and possessing the ability to engage other web applications in order to complete higher-order business transactions, all programmatically accessible through standard internet protocols, such as HTTP, JMS, SMTP, etc. More specifically, Web services are Web applications built using a stack of emerging standards that together form a service oriented application architecture (SOA), an architectural style whose goal is to achieve loose coupling among interacting software components through the use of simple, well defined interfaces [6]. Also, in [6], a stack of emerging standards on which Web services are built were described, and will be summarized here. Figure 2.1 shows a conceptual overview of the Web Services stack.

UDDI	Service Discovery Layer
	Service Publication Layer
WSDL	Service Description Layer
SOAP	Messaging Layer
HTTP, SMTP, etc.	Network Layer
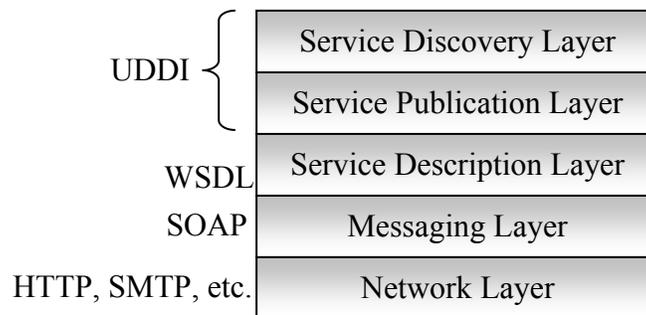
Figure 2.1 Conceptual Web Services Stack

Extensible Markup Language (XML) provides the basis for most of the standards that Web services are based on. XML is a standard that was developed by the World Wide Web Consortium (W3C) [7]. XML is a text-based meta-language for describing data which is extensible and therefore used to define additional markup languages. The mechanism with which a markup language is defined in XML is termed a schema definition. A schema definition is a set of rules that define the structure and content of an XML document. Since XML is text-

based and extensible, it provides the standard on which other standards are built in the realm of Web services.

The lowest layer of the Web services stack, the network layer, is defined since a Web service must be network accessible to be invoked by its clients. Although Web services are typically thought of as operating over HTTP, they are capable of operating over many different types of transport layers, such as HTTPS, Java Message Service (JMS), and even SMTP, providing a great deal of flexibility to application developers. Although, just about any internet traversable transport layer can be used underneath Web services, HTTP is by far the most commonly used Web service transport.

The next logical layer in the stack is the messaging layer, and its related standard is SOAP [8]. SOAP defines a common message format for use by all Web services. SOAP is designed to be a lightweight protocol for information interchange among disparate systems in a distributed environment. The actual format consists of an envelope that defines what contents the message contains and how to process it. In the envelope are a number of standard headers, and a body. SOAP is entirely encoded in XML. The very minimum requirements of a service provider or consumer of Web services are to be able to build, process, and send (over the network layer) these SOAP messages.

The layer above the messaging layer is the description layer, and its specification is the Web Service Definition Language (WSDL) [9]. It provides a mechanism for describing Web services in a standard way. The description provides an interface for using the Web service, in terms of available operations, in terms of their name, parameters and return types. The description binds a service, termed abstract endpoints in the specification, to concrete endpoints, which is a description of the service defined abstractly then bound to a concrete network protocol

and message format. This description is represented using XML as well. This layer is the key element that gives Web services their loose coupling allowing for a new level of interoperability, platform and language neutrality.

The highest layer of the protocol stack is the discovery layer, and is modeled by the Universal Description, Discovery and Integration (UDDI) [10] specification. UDDI provides a means to locate and use Web Services programmatically. Service providers publish high level descriptions of their Web services into a UDDI repository, with which their services can be looked up and used. When an application wants to use a service published in the repository it downloads what the application needs to connect to and consume the Web service it found in the repository. These standards have addressed the connectivity, messaging, description, and discovery issues for Web services, providing the simple, well-defined interfaces required for the loosely coupled, interoperable building blocks known as Web services.

Not only have these standards led to numerous software tools to aid in development of Web services, they have also supported numerous sub-standards. Sub-standards in the context of Web services are typically vendor contributed extensions to the SOAP protocol. Since most of the sub-standards that will be discussed in this thesis have not yet become actual standards, they: (1) exist only as draft specifications, (2) do not yet enjoy industry wide support, and (3) do not have reference implementations provided by the vendors. Sub-standards which do not have reference implementations means that if a developer wishes to use the additional functionality provided by it, the developer must develop the code to perform the functionality. This adds additional complexity to any Web services project, because now, not only does the developer have to maintain their own service, they must also maintain the code created to provide the functionality provided by the sub-standard. In other words, if a sub-standard undergoes changes

on its way to becoming a recommendation, the application may will have to be adapted to fit the new changes. Additionally, some sub-standards are conflicting, created by different vendors to accomplish the very same goal. In this confusing environment of sub-standard proposals, developers are forced to either choose a sub-standard that may not be adopted or may continue to be only supported by a few vendors and implement it, or wait until one of the sub-standards becomes an actual standard.

In the realm of network security, firewalls, proxies, and DMZs [11] are commonly used security measures in enterprise networks. Firewalls provide the means with which organizations protect their computer resources from outside networks. They block packets that are received either from untrusted networks, or from an inside source that is trying to request information from disallowed domains. A DMZ (Demilitarized Zone) is a pair of firewalls working together to form an area, which is separated from internal and external networks, but is logically a part of each. The publicly accessible servers such as Web servers, servlet engines and proxy servers are placed here. Proxy servers provide the means for enterprise networks to prevent outsiders from accessing inside computers by making internal applications (Web service clients) anonymous. If a proxy is in use, all calls for specific protocols will be routed through it, and when the outside service receives the call, the sender's address is that of the proxy and not the client.

Regarding Design patterns, Christopher Alexander, an architect, once said "Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over with out doing it the same way twice". [12]   Even though, as an architect, Alexander was referring to the architecture of buildings and such, his central idea, patterns, was applied to software design as well, thus, software design patterns describe recurring general software design

problems and a proven solution to those recurring problems. Software design patterns allow novice developers access to the best practice of more seasoned developers, as well as providing a common vocabulary for developers to discuss their designs with.

The patterns that were used to develop the framework at the core of this thesis are Observer, Mediator, Proxy, Memento, Abstract Factory, Flyweight, and Strategy [4], Half-Object Plus Protocol, Router [5], and Gateway [3]. Observer defines a one to many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically. Mediator defines an object that encapsulates how a set of objects interact. It promotes loose coupling by preventing objects from referencing each other explicitly. The Memento pattern defines a manner in which an object's internal state can be captured and externalized without violating encapsulation. The Abstract Factory pattern provides an interface for creating families of related objects without specifying their concrete classes. The Flyweight patter is used to support large number of fine-grained objects efficiently. Half-Object Plus Protocol provides a mechanism to allow a single entity, or relationship, to exist in two or more address spaces. The Router pattern allows for decoupling multiple sources of information from the targets of that information. Strategy defines a family of algorithms, encapsulate each one, and make them interchangeable. Gateway provides seamless interoperability between two disparate systems, domains, or object model.

Software design pattern scalability refers to the scope at which the pattern is applied. The scalability model in [3] defines several architectural levels corresponding to the scope of software solutions. By defining such a model, the field of design patterns can be expanded to apply to larger levels of abstraction. At the bottom of the model is the Object level, which refers to the interaction between objects. The next level is the MicroArchitecture level and it refers to

interaction between groups of objects, solving larger problems. At the third level,

MacroArchitecture, this is focused on the development of software frameworks. The next level

is the Application architecture, which refers to the organization of applications developed to

meet a set of user requirements. The system level deals with communications and coordination

between applications and sets of applications. The enterprise level is focused on coordination

between groups of systems within a single organization. And finally, the global level deals with

design issues applicable across all systems, inside or outside the organization.

Chapter 3: Related Works

In the context of Web services, SRMR messaging has not been an active research topic on its own, and is usually associated with asynchronous messaging. Included in the WSDL 1.2 specification are two message exchange patterns, *Out-Multi-In* and *In-Multi-Out* that, if applied together, are SRMR messaging. The W3C working group assigned to WSDL removed this message pattern in the WSDL 2.0 specification to prevent confusion with multicast-capable patterns [13]. In Web services architecture, the role of WSDL is to serve as a low-level description language, just enough to specify the interface of every operation. Logic should not be built into WSDL, which has been a deliberate tactic of the W3C WSDL working group, in order to have WSDL remain a robust standard language. Applications that require this interaction will have to solve this issue at application level. Unlike the SRMR messaging pattern, its requirement, Callback, has been an active research topic and the relevant approaches towards that goal can be logically divided into the following four categories:

3.1 Multithreading

The client-side multiple threads approach [14] suggests that for each synchronous call by the application, we activate a thread to maintain the connection for the call and to wait for the response from the server. The central idea of this approach is to defer the waiting for responses from synchronous calls to the service provider to threads which after receiving the result pass the result back to the application. While this approach unblocks the calling application's control flow, the threads do not release the connections to the service provider, even for long duration transactions. Relying heavily on maintaining a connection to the service provider is the major

liability of this approach. If the connection is lost, the solution fails. Also, this approach does not support resumable clients, which are clients that wish to shut down, restart, and resume operations.

3.2 Specific Asynchronous Protocols

The approach developed by Holt Adams at IBM involves using asynchronous transports to perform the needed asynchronous calls and the use of threads to perform the asynchrony [15]. The major drawback of this approach is that would force the developer to use specific asynchronous protocols such as SMTP or JMS, and such a requirement may not be feasible for some applications and environments. A characteristic of this particular approach is the total transparency of the underlying callback to the client. While this makes its application easier to implement, it is also a drawback, because it does not release the calling client from being blocked. The client would have to use multiple threads to unblock its control flow.

Another closely related work is the Web Service Invocation Framework (WSIF), which IBM initially developed, and later donated to the Apache XML project [16]. WSIF is a client API that invokes web services using a local proxy. WSIF can support Web service callbacks but requires the use of JMS as the underlying transport layer, which is a serious limitation to most applications, and may not be feasible in others.

3.3 Extensional Web Service Sub-Standard Protocols

This approach involves the use of extensional Web service sub-standards, which are typically SOAP extensions, developed for specific problem domains. BEA developed the powerful "WS-Callback" protocol [17]. It is a SOAP based solution that defines "standard" new

headers in the SOAP messages that the requestor can use to dynamically specify where to send

asynchronous responses to a SOAP request. WS-Callback does not have built in support for

message correlation, and this poses two problems. First, the responses will have to be sent

directly to the waiting application, which in a secure environment is impossible. Second,

without message correlation, WS-Callback cannot be directly applied to SRMR messaging,

because the application which makes a number of similar calls reliably would have to decide

which partial response is a result of which request, which may not be feasible. WS-Callback can

be extended to handle message correlation, but any attempt to do so would deviate from the sub-

standard. Any deviation from the sub-standard would have to have support on both ends of the

service provider/service requestor chain, thus greatly complicating its deployment. Ensuring that

all parts of the chain support all requirements of the system may not be feasible, especially in the

context of external Web services in which the external entity will not be under the control of the

developer. Figure 3.1 shows an example of SOAP messaging using WS-Callback extensions.

```
<s:Envelope xmlns:s="http://schemas.xmlsoap.org/soap/envelope/">
    <s:Header>
        <wscb:CallBack
            xmlns:wscb="http://www.openuri.org/2003/02/soap/callback/"
            s:mustUnderstand="1">

            <wscb:callbackLocation>
                    http://merres1.cs.uno.edu/axis/WSClearinghouse
            </wscb:callbackLocation>
        </wscb:CallBack>
    </s:Header>
    <s:Body>
        …
    </s:Body>
</s:Envelope>
```

Figure 3.1 Example of WS-Callback SOAP Message

WS-Addressing, currently being proposed, allows a service request to pass a "reply-to"

address of a callback listener to the operation call [18]. WS-Addressing also supports message

correlation. This sub-standard cannot be considered a final solution, because in order to pass the

result from the service provider to application, the service provider must know the address of the final recipient and be able to reach the final recipient directly. Some entity may act as a proxy for the return path, but since there is no routing information described in the WS-Addressing specification, the entity would have to be an application-level gateway, very similar to the one being proposed in this thesis. In other words, WS-Addressing may be used in conjunction with the framework being proposed, but not instead of the framework being proposed. Figure 3.1 shows an example of SOAP messaging using the extensions provided by WS-Addressing.

```
<S:Envelope xmlns:S="http://www.w3.org/2003/05/soap-envelope"
  xmlns:wsa="http://schemas.xmlsoap.org/ws/2004/08/addressing"
  xmlns:mrms="http://merres1.uno.edu/">
   <S:Header>
      <wsa:MessageID> 1</wsa:MessageID>
      <wsa:ReplyTo>
         <wsa:Address>
            http//merres1.uno.edu:8080/axis/CWS
         </wsa:Address>
      </wsa:ReplyTo>
        …
   </S:Header>
   <S:Body>
      …
   </S:Body>
</S:Envelope>
```

Figure 3.2 Example of WS-Addressing SOAP Message

Also, more ambitious specifications, such as BPEL4WS [19], WS-CDL [20], and BTP [21], have been proposed to handle not only callback, but choreography and orchestration for Web services as well. These specifications describe the behavior and relationship of business processes and their partner business process by defining standard ways to compose these basic Web services into larger composite business processes. The major drawback involved in using any of these sub-standards, is that for any message the service provider wants to send to the client directly, the client must be accessible from that service, which in a secure environment is impossible. Also, since most of these are experimental or draft specifications, to use these sub-standards would require hand-coding.

3.4 Web Service Based Client-Side Listeners

The central idea of this approach is to have the client either create a listener, or become a listener itself. If one were to use a listener application accessible from the outside for each client application, then callback would truly be supported. This approach is described in [22]. While it is necessary to have a listener Web service for accepting the callback messages from the servers, deployment and management on a one listener service per application basis would be excessively costly, considering the client-side applications are bound to be numerous and volatile.

Also, related to this approach is the Faux Implementation Pattern in [23]. This approach suggests that we have the client application pretend to be a Web service, receiving and processing SOAP messages, on the client-side. The client itself would expect the callback response from the server. This would also be a true callback pattern, if and only if, the listener is reachable from the service provider, but in a secure enterprise environment it will not be accessible.

Chapter 4: Framework

The proposed client-side framework for utilizing SRMR Web services is composed of two major components. As mentioned earlier, the key component of the framework is the Clearinghouse Web service, CWS and other component is an agent which is used directly by an application that wishes to consume SRMR Web services. The agent component is instantiated by the application, and is therefore, running locally for each application. Figure 4.1 illustrates a simplified overview of the architecture of the framework. This diagram attempts to emphasize that many applications in a given secure enterprise through their corresponding agents may interact with different Web services while sharing the same CWS.
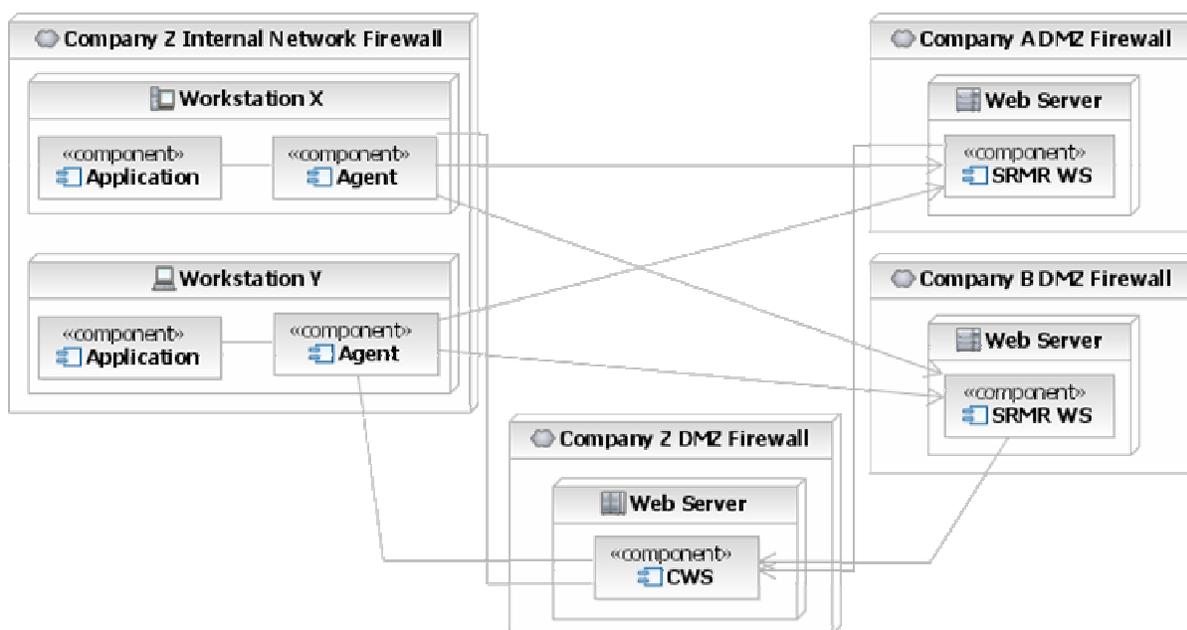

Figure 4.1 Simplified Overview of Framework

This chapter will first highlight the objectives of an ideal framework that supports SRMR messaging, and then give a detailed walkthrough of the system, and finally, discuss each of the

major components of the developed framework. In the discussions, particular attention has been

paid to optimize the design by applying design patterns where appropriate.

4.1 Ideal Solution Objectives

An Implementation of SRMR messaging by itself is trivial, but in the context of Web

services and enterprise network security its implementation is anything but trivial. Using the

related works as a guide, an ideal framework developed to allow client applications to utilize

Web services that support SRMR messaging should support the following features:

- Unblock clients after making a successful call.

- Release the server immediately after the service accepts the initial call.

- Minimize the management of listener services for all applications using the framework.

- Avoid the inherent complexity involved when using Web service sub-standards.

- Allow any underlying communication protocols instead of requiring any specific one.

- Support resumable clients for long duration transactions.

- Shield the complexity of using SRMR messaging from client applications.

- Maintain the level of interoperability provided by Web services, making the solution both
  platform and language neutral

4.2 The Process: A Detailed Walkthrough

A detailed walkthrough of the process that takes place when an application makes a call

to a supported Web service through the framework with brief explanations along the way is

described in this section. In this walkthrough, a SRMR Web service will be denoted as simply a

server. An application must first instantiate an agent object, if it does not already exist, before it

attempts to invoke an operation of the server. This agent object will be used by the application

as a proxy to the involved Web services. The application will make calls through the agent and

receive results through the agent, thus shielding the complexity of the framework from the

application developer. This is an application of the Proxy pattern. The collaboration diagram

shown in Figure 4.2 describes the process, which is performed by the actors in the collaboration

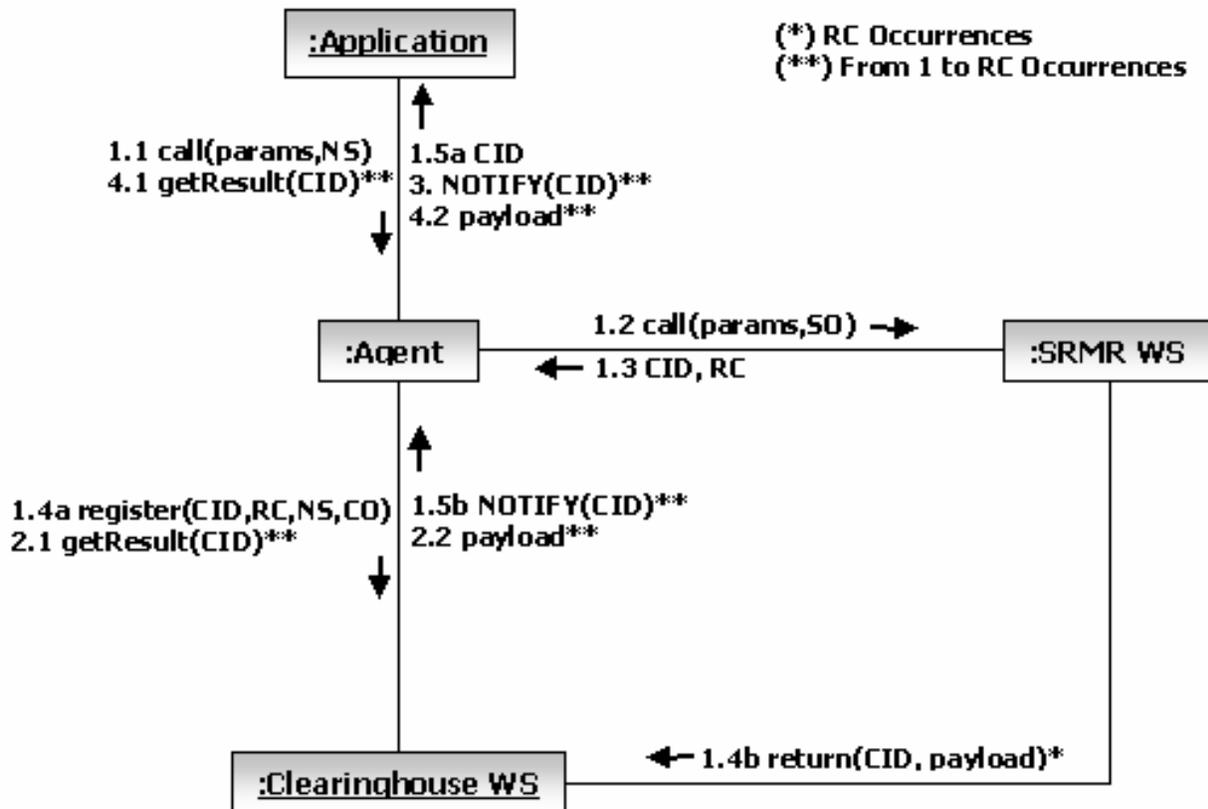carrying out the following four multipart steps:

Figure 4.2 Collaboration Diagram (Callback Agent)

In step 1.1, the application passes the call to the agent along with all the parameters

needed to make the call, and a notification style for that request. The notification style sets the

notification strategy that the framework will use to notify the application. For instance, an

application may only want to know when all responses have been received for local pickup, or an application may want to know about each individual response.

In step 1.2, the agent calls the corresponding operation on the server on behalf of the client. This call takes as parameters, the parameters that were just passed to the agent by the application, and a Server Observer object. This object holds the address information (the URI) of the CWS where the responses should be sent. An important point to note at this point is that at no time is the clients address information sent outside the firewall, which is done not only to prevent giving outsiders more information that they need, but also because the outside service cannot use the information to send the callback directly because of the firewalls in place.

In step 1.3 the server receives the call, performs a validity check of the parameter list, ensuring that the parameters meet the requirements of the service contract. If the call is valid, the server returns the correlation identifier, or CID, and the number of responses the server will eventually send. The agent maintains a counter to track messages that have been received and not yet received, based on the response counter. This two part response is completed using a Multiresponse object that simply carries the two values.

In step 1.4a, the agent registers at the clearinghouse, a call which takes four parameters: The CID and response count, which were just received from the server, a Client Observer object, and the notification style it received from the application in step 1.1. The Client Observer object holds the address information of the agent object, in the form of hostname and port. In step 1.4b, at some point, the server finishes processing the request and begins returning responses to the CWS. These calls to the CWS from the server are synchronous, so that if calls are not received properly, they may be resent. The number of calls that the server will make to the CWS will be given by response count, and a response counter is kept at the CWS to manage multiple response

correlation. Each response in step 1.4b consists of a partial payload and the CID that the server sent to the agent in step 1.3.

In Step 1.5a, the agent returns the CID to the application. In step 1.5b, which occurs when an agent is registered for a result with a certain CID and a response is received for that CID, and according to the notification style strategy, it is time for a notification, the CWS informs the agent that registered for that CID, this notification takes the form of a string passed over a TCP/IP socket. This socket connects since the CWS exists in the client-side DMZ, and therefore can reach the agent. The agent may be notified anywhere between once and response count number of times, depending on the notification style in place for that CID.

In step 2, the agent queries the CWS for the results, and receives all responses that have been received up to that point by the CWS. In step 3, the agent notifies the application, using the object level observer pattern, that results are ready at the Agent.

Finally, in step 4, the application queries the agent for results, and receives all responses for that CID that the agent has received up to that point. Note that steps 1.4a and 1.4b may happen concurrently, as well as steps 1.5a and 1.5b. Note also, that in steps 3 and 4, every time the agent is notified of results by the clearinghouse, it notifies the application, queries, and receives the responses, so these steps also follow the pattern set forth by the notification style, which results in between one and response count notifications.

In the above process, the agent and application are related via the Observer pattern. The application may also poll the agent for specific results. The process involved when an application polls the agent for results is somewhat different than the process for when the agent and application are related via the Observer/Observable pattern. The collaboration diagram for

this process shown in Figure 4.3 describes the process involved when an application polls the
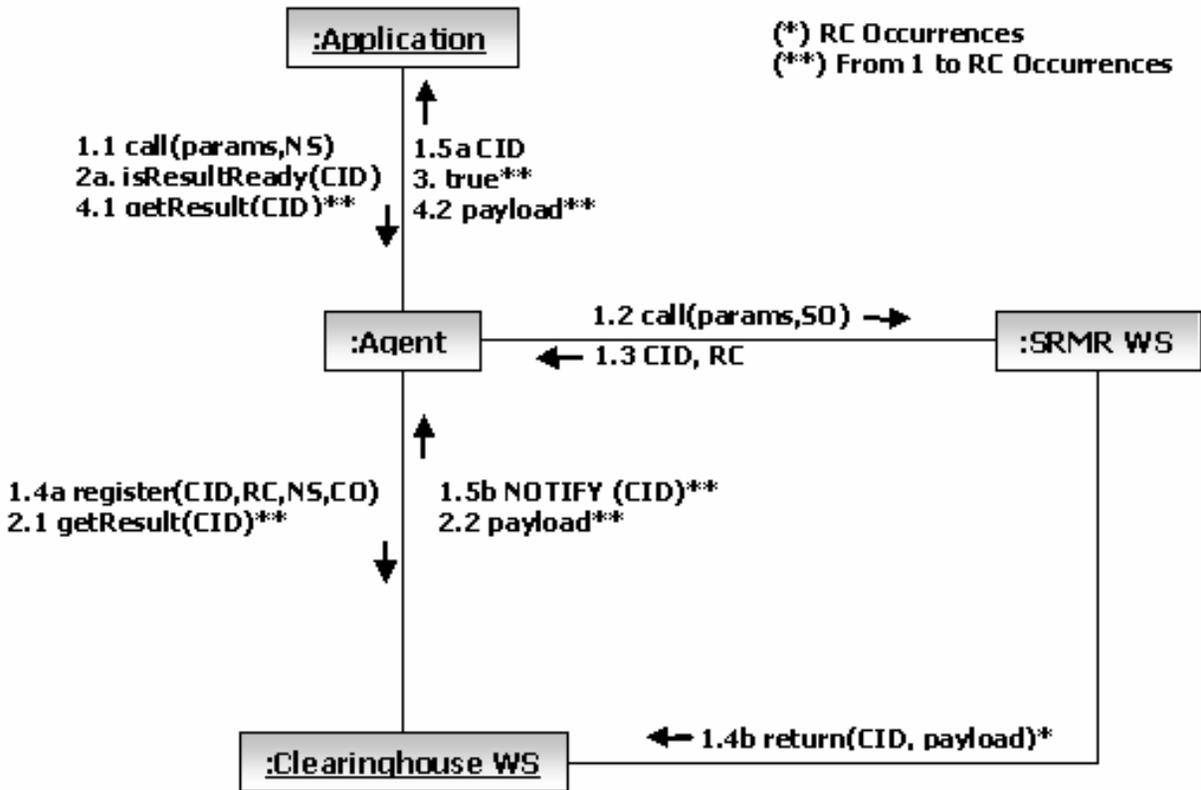
agent for specific results.

Figure 4.3 A Collaboration Diagram (Polling agent)

When the application uses the polling agent process shown in figure 4.3, steps 2, 3, and, 4

are different. Just after receiving the CID from the agent, in step 2a, the application starts polling

the agent. When the result of those polls returns true, it gets the result as it did in step 4. This

interaction was also included in the developed framework to allow a higher level of flexibility

for those applications who wish to consume these types of services through the use of the

developed framework. An interaction is chosen when the application instantiates the agent. The

agent instantiates a polling agent using a constructor with zero arguments, or a callback agent

using a constructor that takes an Observer as an argument.

4.3 Architectural Overview

The framework consists of two major components and several related helper objects. The components and their helper objects will be discussed in terms of their responsibilities, functionality, and the design patterns used to construct them. The components and their helper objects are as follows:

Clearinghouse Web service (CWS) As mentioned earlier, this is the key component of the developed framework. The use of a clearinghouse for centralized correlation processing was proposed for CSP-like communication in [24]. This clearinghouse of this framework is similar; it is used to centralize the correlation of messages, but also handles the message distribution. The decision to use a centralized listener Web service (the clearinghouse) rather than having a single listener Web service per application as [22] is based on a number of considerations. First, the central listener service decouples the timing coupling relationships between the calling applications and the callback services. The CWS accepts response messages for agents, therefore applications, that may, or may not, be active. This timing decoupling allows the framework to support resumable clients. Second, management of a single CWS per enterprise is much simpler than the management that is required to create and manage a listener Web service for each application and agent pair, which as mentioned earlier can be prohibitively costly.

The CWS component consists of four logical operations: registration, deregistration, send-result, and fetch-result operations. The agents use the registration operation to inform the CWS what message identifiers, or CIDs, that they are interested in. The agents use the unregistration operation to inform the CWS that they are no longer interested in receiving notifications for a message identifier until it re-registers. The fetch-result operation is used by

the agents to actually get the results from the CWS. Finally, the send-result operation is utilized by the server to return the results of the clients' calls.

When an agent registers at the CWS by calling the registration operation, the CWS stores the response count and the notification style as well as the correlation information (the agent's address and the expecting CID) into the clearinghouse database. Note, to prevent any possible identical CIDs produced by different service providers, each stored CID is concatenated with the service provider's URI. When the service providers call the send-result operation to deliver response messages, the CWS stores the payload according to its CID. After both of these operations, the CWS checks for matching registration and payloads based on their CID. If a match is found, it uses the notification style to determine if a notification is necessary, if it is, it immediately notifies the client about the arrival of the results. Once informed, the client is free to pick up the messages from the CWS, by calling the fetch-result operation. The CWS will return all the correlated messages that the CWS has received thus far for the CID. Once the response count for a CID drops to zero and the response messages for this CID have all been fetched by the corresponding agent, the CWS will purge the corresponding registration entry from the clearinghouse database.

The final operation, the deregistration operation, is provided to allow the applications to disconnect from the CWS temporarily. No notification will occur for those CIDs that the agent initially registered for, until the client resumes and re-registers. This functionality is provided simply by removing the agent's registration entry from the correlation table used by the CWS. This prevents matches, and thus notifications, from occurring, but does not interrupt the flow of messages containing payloads from service providers. These payloads are stored until a match occurs, and that will not occur until the agent reregisters.

A number of design patterns were applied in the design of the CWS component including Observer, Router, Proxy, Gateway, Abstract Factory, Half-Object Plus Protocol, and Strategy at different levels of scalability. The CWS uses the Strategy pattern at the Object level to implement the handling of notifications based on their notification style. The Abstract Factory is also applied at the Object level to create the handler strategies. The agent and the CWS use the Observer pattern at the enterprise level. How the CWS operates is best described using the Router, Proxy, and, Gateway patterns.

The router pattern was applied in the sense that it routes the messages based on CID, clearly content-based routing, to the appropriate agent. This is necessary among many clients and many servers, all interacting through the same CWS. The Proxy strategy is also used to return messages from the server to the appropriate agent through the CWS. The CWS also follows the Gateway pattern in transmitting messages, since the server cannot call the agent directly due to disparate domains. The Router, Proxy, and Gateway patterns are applied at the global level.

The CWS utilizes two helper classes:

o **PayloadHandler** – This object is used by the CWS to determine what action to take upon receiving a set of messages based on notification style for the CID in question.

o **ClearinghouseObserver** – This object allows the CWS to seamlessly use the Observer pattern. It acts as a half-bridge between the agent and CWS to actually send the notifications to the agent. The CWS uses this as an object level Observer pattern, and when the CWS needs to notify the client, it notifies this object, which in turn serves as a proxy to notify the agent's helper. This is how the "Half Object Plus Protocol" pattern is applied. To eliminate unnecessary notifications, and to reduce overall network traffic, the CWS will not notify the

same agent about the same CID again until the agent takes an action upon receiving the previous notification.

Agent The purpose of the agent component is to shield the application from the complexity of the framework by serving as a proxy. Every application that uses our framework instantiates an agent object to in order to interact with the framework. If the application prefers the Observer pattern, the application passes a reference to itself in the agent's constructor. If it would prefer to poll the agent, it would use a zero argument constructor. In either case, the agent would still support applications that may need to shutdown temporarily. The decision to support multiple types of agents was made to breed in flexibility regarding how the application will consume the results of the Web services.

The patterns used in the development of the agent are Observer, Proxy, Half-Object Plus Protocol, and Memento. The Proxy pattern is applied to model the end to end nature of the agent handling the call, and finally returning the results. The Proxy pattern would then be applied at a global level. The Observer pattern is applied to model the interaction with the CWS at the enterprise level. As mentioned earlier, they are related via the Observer/Observable relationship, with the agent performing the Observer role. The agent can be related to the application using the Observer pattern at object level as well, in which case it becomes the observable part of the interaction. Lastly, the agent also takes part in the Memento pattern at object level, when it deregisters, saves its state persistently, and shuts down, so that at some later point, it can be re-instantiated using the saved state, reregister at the clearinghouse, and, therefore resume operating where it left off.

Once the agent registers at the clearinghouse, the agent informs its helper to start listening for a notification from the clearinghouse. Once the notification is received in the form of a string, the agent parses it to get the CID. The agent then calls the fetch-result operation of the CWS using this CID. This call will return all the responses that have arrived at the CWS associated with that CID since the last retrieval by this agent. As mentioned earlier depending on the relationship between application and agent, there are two paths this interaction can take. Using the Observer/Observer relationship, the agent notifies the application that results were received by using an object level notify passing the CID in the notification. Using the other relationship, the agent's poll would return true. The application would then get the results from the agent. The agent then updates its count of the returns. Upon returning all messages to the application for a given CID, the agent will then free all resources associated with that CID.

The agent utilizes only one helper class:

o**AgentHelper** – This helper object is created upon instantiation of the agent. It serves as the other half of the bridge that allows the agent and the CWS to be related via the Observer pattern. The "Half Object Plus Protocol" pattern is applied at enterprise level to provide the illusion that the agent is a local object of the CWS. The agent informs the helper when listening is necessary, and the helper informs the agent when notifications arrive from the CWS. The helper and agent are also related via Observer/Observer relationship at the object level. It performs its duties by managing the sockets for the agent. Once the server socket accepts a connection from the clearinghouse, the helper receives the message (the CID string) and passes the string to the agent.

Shared Helper Classes The following helper objects are used by multiple components of the framework. The pattern they observe is the Flyweight pattern. These objects are shared in the sense that they are passed from one component to another and provide no functionality other than the information that they contain within.   The three shared helper classes are:

o**ClientObserver –** models the hostname and port number of the agent's helper. It is passed from agents to Clearinghouse, and is used by the CWS to create a ClearinghouseObserver.

o**ServerObserver** – models the URI at which the CWS is listening for incoming responses from remote servers. It is passed from agents to the service provider.

o**MultiResponse –** models the response count and CID of a valid request. It is passed from the service provider to the agent as the result of a valid call.

Chapter 5: Implementation Details

In this chapter, implementation details important to the design of the framework shall be discussed along with an overview of the generation utilities. The two major design decisions that relate to the design process that will be discussed are the communication between the CWS and agent, and the generalization of the return type to allow for only one CWS to handle any type of return. The discussion of the framework generation utilities will follow the discussion of the two major design decisions.

5.1 Clearinghouse to Agent Communication

Since the CWS and the agents exist in different environments, the communication between them should be carefully considered. When the CWS receives a result, for which a notification should be sent, depending on the notification style in force, it should notify the agent as soon as possible. This notification is performed using a callback mechanism to minimize latency. The CWS then should send the CID to the agent.

Furthermore, the communication means the CWS uses to notify the agents is socket-based. Sockets were chosen for both their simplicity and their interoperability. Sockets are an interoperable choice, because Sockets exist in every modern operating system, and are supported by every modern programming language. When a CWS wishes to notify an agent, the CWS, by way of a helper, opens a socket, and passes the CID in the form of a string to the agent. The CWS will not notify the same agent regarding the same CID again until the agent takes an action upon the previous notification. This is done to eliminate unnecessary notifications and reduce overall network traffic.

5.2 Generalization of Return Types

In order to support any number of service providers, the CWS must be able to handle any type of response. More specifically, the return type should be generalized into an acceptable form so that only one CWS will ever be needed per enterprise. This generalization process should be language neutral, and should be unambiguous in the manner it specifies return types and objects. XML provides the vehicle for which all these objectives can be achieved. Using XML, and schemas, the response type is generalized into a schema definition, thus providing a language neutral type definition. Using this definition, the service provider would then serialize the response into an XML string, and pass the string to the CWS. The CWS can handle the return type String, because String is a simple type in any schema definition. When the agent obtains the String as a result of a call to the CWS, the agent uses the same definition the service provider used to deserialize the XML string into an object the agent's language can understand. This definition could be retrieved by the application from the UDDI repository entry for the service provider, or sent as requested by the service provider. A specific example of how this return type generalization is performed will be discussed in detail in Chapter 6 along with the discussion of the case study.

5.3 Generation Utilities

The code generation utilities, along with the generated reference implementation, were developed in Java using only the following non-standard API libraries: Apache Axis [25], Java API for XML Binding (JAXB) [26], and JavaDoc [27]. Axis provides client-side interfaces for connecting to and consuming related Web services. JAXB provides the mechanism with which

the result types are generalized using XML schemas. JavaDoc is used by the generation utilities

to gather information from Java interfaces to create the necessary Web services.

The generation of the individual pieces of the framework is done in two parts. First, the

client-side CWS is created, and then the Agent is generated afterwards. It is performed in two

parts, since for each enterprise only one CWS will be needed regardless of how many different

web services it may use. The remote services were not generated because they are thought to

already exist. The generation toolkit consists of two separate utilities, one to create a CWS, and

another to create the Agent and helper classes that an application may use to interact with the

service provider that uses the SRMR message exchange pattern.

The CWS has two groups of operations: three operations for agents, and one for the service

providers. In the following class diagram of the CWS and supporting classes shown in figure

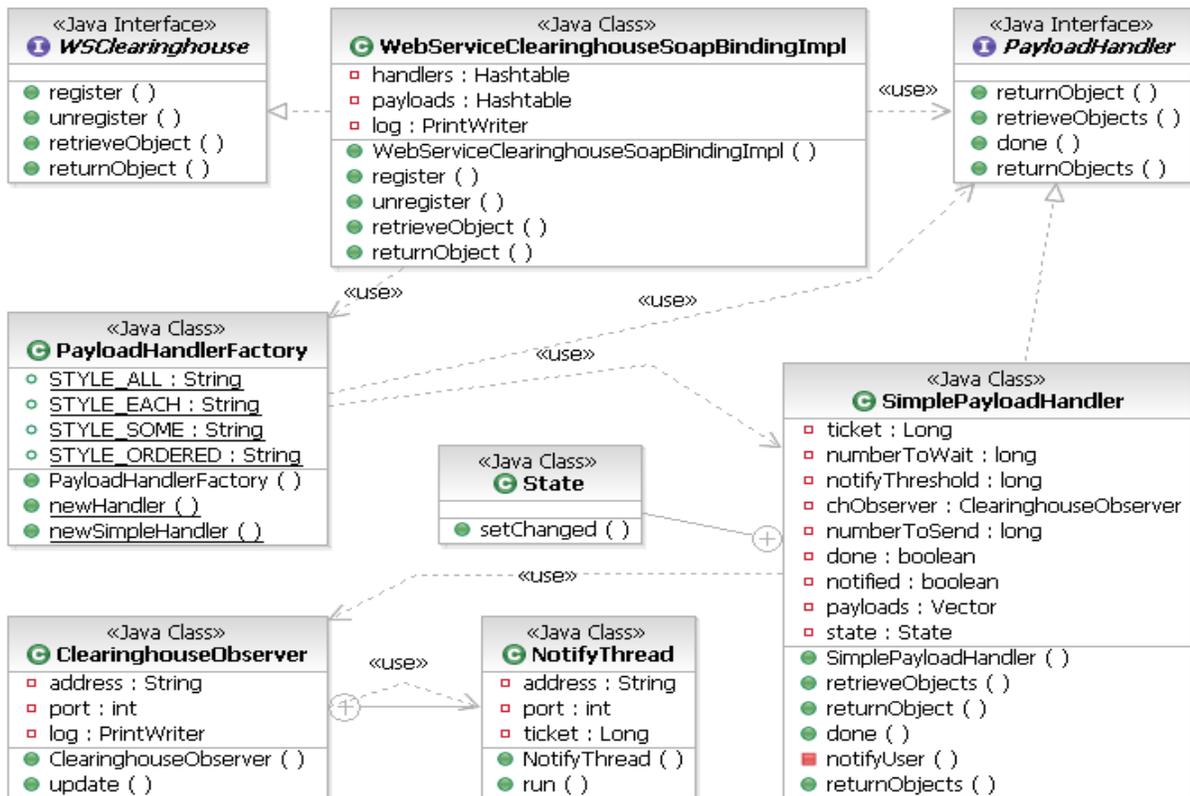5.1, the client-side connection code generated by Axis has been left out for clarity.

Figure 5.1 A Class Diagram of the CWS and Supporting Classes.

The generation utility that builds the CWS requires only the URI of its eventual deployment as a parameter. It uses a static interface to generate the CWS by passing it to Axis' Java2WSDL tool, and then passing the newly generated WSDL file through its WSDL2Java tool, and finally replacing the final implementation classes with an implementation already created. Once the utility is finished, deploying it is as simple as copying the files into the Web service container and using the container provided deployment mechanism that takes as parameter the new generated web deployment descriptor.

The agent is generated with the following parameters: (1) the WSDL of the Web service from the service provider, (2) the URI of the client-side CWS, and (3) a schema definition of the return type. The procedure begins with generating a Java interface from the WSDL document using the Axis utility, WSDL2Java. From this Java interface, we extract useful information using a Java doclet, which will be used to generate the client-side framework. This information includes method name, parameter lists, and class name. The client-side connection code that was generated by WSDL2Java is copied to the shared folder of the destination directory since it will be used by the agent to actually connect to the remote service. The client-side connection code generated by Axis can be thought of as an adapter. It uses Java classes and interfaces to call the Web services on the behalf of the caller, through the framework, thus shielding the complexity of using Web services from the caller. This client-side connection code includes not only the classes used by the Agent to connect to the remote web service, but any parameter objects as well. Then, the JAXB libraries and the XJC command are used to create the classes to marshal and unmarshal the return type in the destination directory, which will be discussed in more detail in Chapter 7, along with the case study. The XJC command takes as parameters: a destination directory and a schema definition file. Then, the files that do not need customization, such as

31

helper objects, the client-side code to connect to the CWS, etc are copied to the shared folder of

the destination directory. Finally, a Java Template class is used to customize the agent based on

the collected information. The Java Template class contains a template, a file that contains a

parameterized version of the agent, and uses regular expressions to perform pattern matched

replacements on a template. Finally, the code is compiled and ready for use by the application.

Chapter 6: A Case Study

In this chapter, an example, a purchase order system, will be discussed as well as some of the internal details of the framework not discussed earlier. A class diagram for the Agent and supporting classes is shown in figure 6.1.
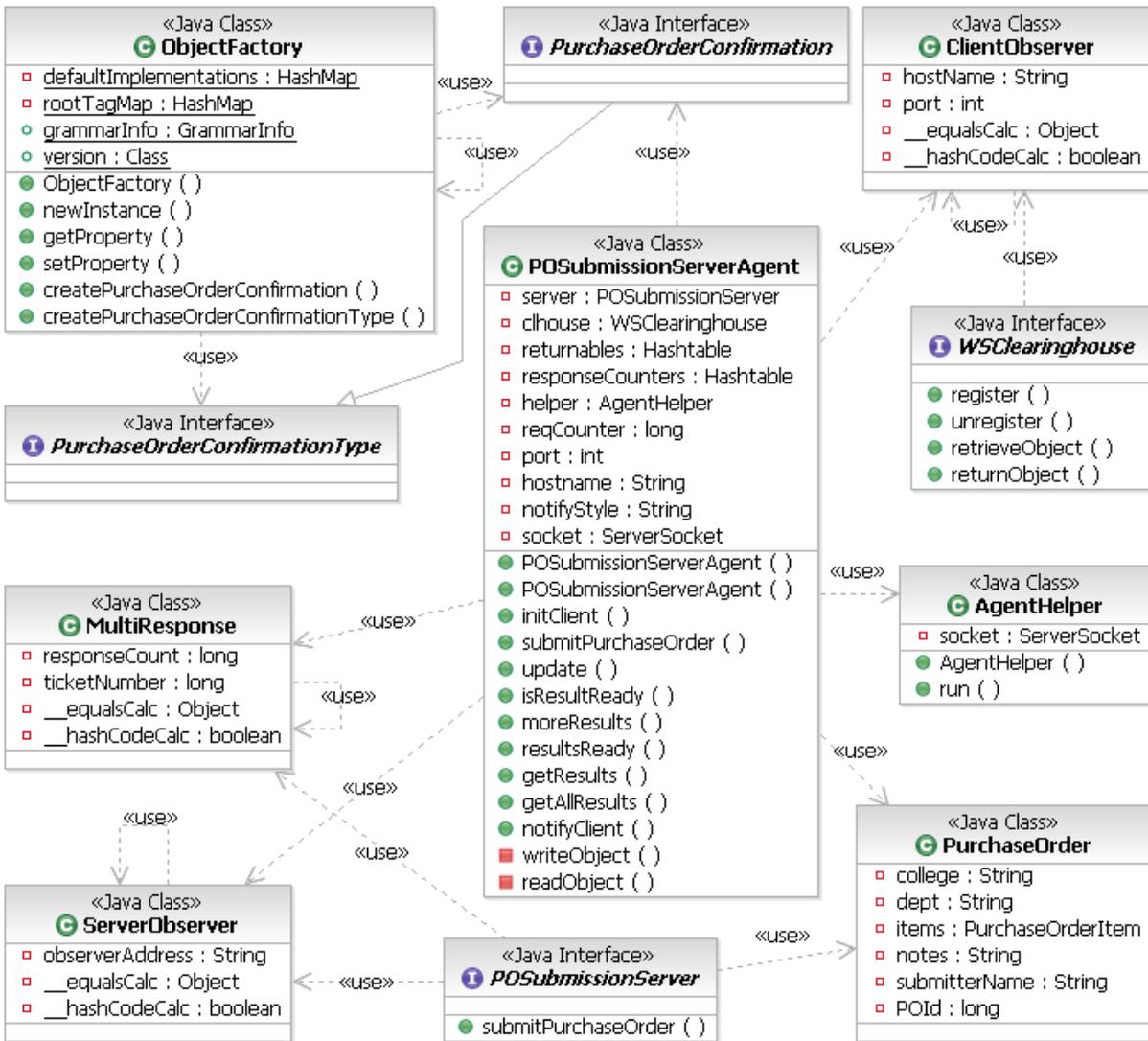

Figure 6.1 Class Diagram of Agent and Supporting Classes

The purchase order system creates purchase orders (POs) and submits them to an "order accepting" operation of a Web service provider. Someone submitting purchase orders may have

many orders to submit and waiting for each submission's fulfillment is simply not feasible,

especially since the orders may take a very long time to fulfill considering things like availability,

human interactions, etc. Instead of waiting for the vendor to complete each order, the user would

prefer that the vendor simply accept the order and notify the user of the results of the submission

later. Also, since some items are handled by different departments within the vendor company,

the departments handling each part of the PO send their partial results directly back to the user.

This example is representative of existing purchase order systems and an example of how a

request may lead to multiple responses. The following diagram, figure 6.2, shows a deployment
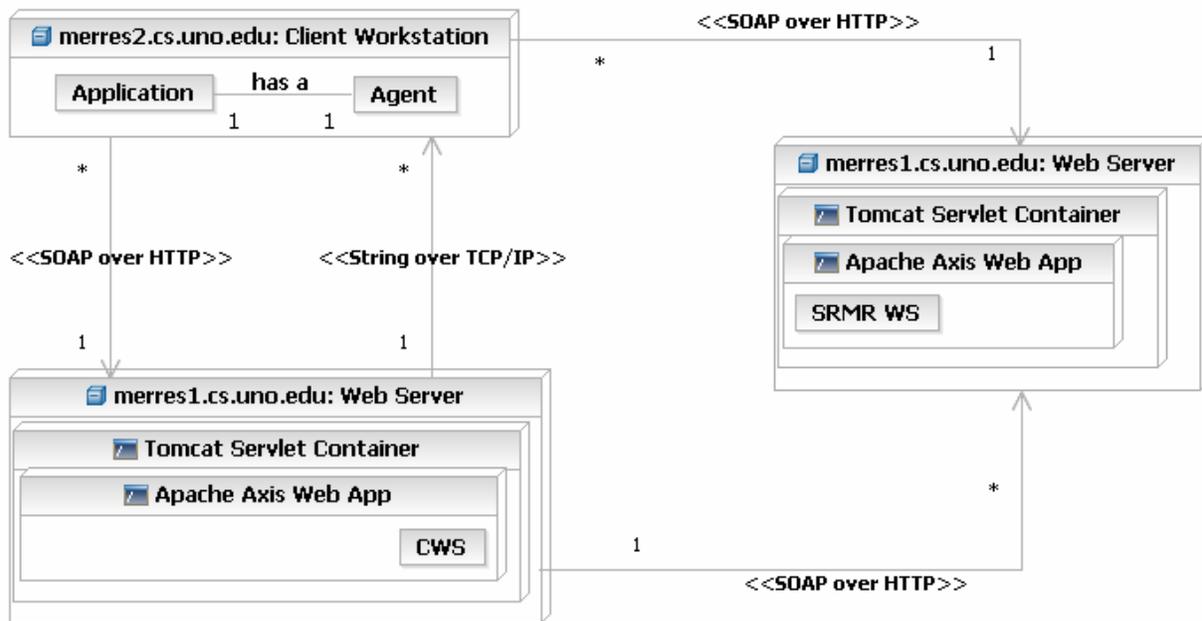
diagram of the resulting system.

Figure 6.2 Deployment Diagram of PO System

In this system, the application and agent are related via the Observer pattern. The system

follows the process outline in chapter 4. The agent and the CWS were generated using the

developed code generation utilities. All the objectives outline in Chapter 4 were achieved with

minimal coding effort.

The specific implementation detail that will be discussed using the specific example provided by the case study is the mechanism used to generalize the return type using the JAXB libraries. In order to utilize the JAXB libraries, an XML schema must be used. Figure 6.3 shows the XML schema that defines the result type of a PO submission, a PurchaseOrderConfirmation (POS) object which is passed to the XJC command along with a destination directory and package name:

```
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<xsd:element name="PurchaseOrderConfirmation"
type="PurchaseOrderConfirmationType"/>
<xsd:complexType name="PurchaseOrderConfirmationType">
  <xsd:sequence>
    <xsd:element name="cost" type="xsd:double"/>
    <xsd:element name="shippingCost" type="xsd:double"/>
    <xsd:element name="manifest" type="xsd:string"/>
    <xsd:element name="shippingInfo" type="xsd:string"/>
  </xsd:sequence>
</xsd:complexType>
```
Figure 6.3 Schema Definition of PurchaseOrderConfirmation Return Type

In our reference implementation, the service provider was also implemented in Java and used the shared folder with the generated code for unpacking/packing the return type. The code used by the service provider to marshal the return type into an XML string is shown in Figure 6.4.

```
JAXBContext jc = JAXBContext.newInstance( "shared" );

ObjectFactory objFactory = new ObjectFactory();

PurchaseOrderConfirmation poc = objFactory.createPurchaseOrderConfirmation();

Marshaller m = jc.createMarshaller();

m.setProperty( Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE );

ByteArrayOutputStream byteStream = new ByteArrayOutputStream();

m.marshal(poc,byteStream);

String payload = new String(byteStream.toByteArray());
```
Figure 6.4 Service Provider Marshalling the Object into an XML String

The process shown in Figure 6.4 above is as follows: First, a JAXBContext is created using the package that the JAXB code generator used earlier, in this case the "shared" package. It then creates and uses a JAXB ObjectFactory to create an empty POC. A marshaller is then created, and configured. A ByteArrayOutputStream is instantiated to marshal the output to. The object is then marshalled into the byte stream and the payload string is created from the byte stream.

Figure 6.3 shows the code used by the Agent to unmarshal the XML string into a POC object.

```
JAXBContext jc = JAXBContext.newInstance( "shared" );

Unmarshaller u = jc.createUnmarshaller();

ByteArrayInputStream inputStream = new
ByteArrayInputStream(payload.getBytes());

PurchaseOrderConfirmation pocIn =
             (PurchaseOrderConfirmation)u.unmarshal(inputStream);
```

Figure 6.5 Agent Unmarshalling the XML String into a POC object

The process outlined by figure 6.3 is as follows: A JAXBContext object is created using the package "shared" and is then used to instantiate an Unmarshaller Object. A ByteArrayInputStream is then instantiated by passing the string in byte array form into its constructor. Then a POC is unmarshalled from the byte array stream, and then finally, ready for use by Agent.

Chapter 7: Performance Considerations

In this chapter, some performance considerations regarding the use of the proposed framework compared to a synchronous framework will be discussed. Compared to a synchronous Web service call that goes through a direct communication between the application and the Web service provider as shown in Figure 7.1(a), the proposed clearinghouse approach requires a going through a sort of communication triangle as shown in Figure 7.1(b).
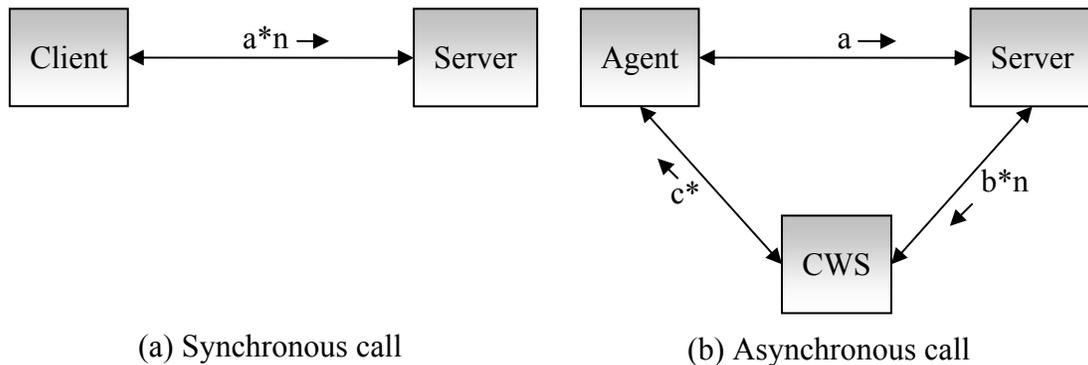

(a) Synchronous call   (b) Asynchronous call

Figure 7.1 Cost Comparison Diagrams

Suppose a request results in n responses. In a synchronous fashion, the client would make n calls; each call including a request and the response. Thus the total communication cost for the synchronous way will be a*n, where a represents the cost of a call. Note that this cost of a call (a) would include both the send time, response time, and return send time. Also, the call would go across the firewall. In the fashion described by the developed framework, the client's initial call (a) is followed by n callbacks (b), and then finally c* models the communication between Agent and CWS. Note that the cost of a call (a) would include both the send time, response time, and return send time and the cost of a call (b) is logically the same as call (a). In c*, the communication being modeled is the notifications being sent as well as the retrieval of

responses from the CWS. Thus, the total communication cost for the proposed framework would be a+b*n + c*.

Since both the initial call (a) and the responses (b) go across the same logical distance across the firewall, it is easy to see that a ≈ b thus producing b+b*n + c*. Also, since the CWS and the Agent are in the same local area network, the communication cost between them (c*) is negligible. Assuming the cost of the calls from each model are similar since they cross the same logical distances we have a+a*n > a*n. Thus, the overhead of the proposed approach is that there is a single extra call (a). Since the approach carries such a trivial cost, any developer using the framework may benefit from the enhanced functionality and ease of use without significant performance loss.

Chapter 8: Conclusion

In the context of secured enterprise environments, this thesis has only addressed the aspect of accessibility, which applies to having multiple components being able to communicate. End-to-end security measures for Web services were not addressed due to the extensive amount of research in that area, and therefore ready to apply approaches are available, such as those provided by the Web service sub-standard WS-Security family [28]. This group of SOAP extensions has established the means to provide quality of protection through message integrity, message confidentiality, and single message authentication. The main objective of the framework was to provide a means to enable client applications to call remote Web services and have the response received in an asynchronous manner, without conflicting with any network-level security measures commonly deployed in enterprise networks, such as firewalls and proxy servers. The core of the framework, the CWS, is just a Web service. Any necessary layer of security should be added as a layer on top of the framework, as one would do with any developed Web service to secure the Web service. For example, the XACML [29] and the Service View [30] describe permitting or denying access at a very fine grained level. These solutions can be used to apply different security measures for internal and external clients of the CWS, preventing outsiders from calling the operations meant for the internal agents.

As mentioned in the related works, WS-Addressing supports return addresses and message correlation which can be used to enhance the framework, but not to replace it. The major reason WS-Addressing was not utilized by the framework was that it exists only a reference specification. WS-Addressing lacks both vendor support and reference implementations. If WS-Addressing would have had such support and implementations, the

proposed framework would have been simpler, but not obsolete. The framework would have

used WS-Addressing to correlate and address responses, but still would use the CWS to handle

the content based routing and as well as serve as a proxy for the return path of the responses.

The agent would still register for responses based on CID and the CWS would still need to notify

the agent when results are ready.

The framework that has been developed to implement Web services that support the

SRMR message exchange pattern in the context of secure enterprise environments has

accomplished all the objectives set forth earlier in Chapter 4. The framework leaves clients

unblocked after making successful calls, releases the server immediately after the server accepts

the initial call, minimizes the management of listener services throughout the enterprise, avoids

inherent complexity of using Web service sub-standards, avoids using specific underlying

communication protocols, supports resumable clients, shields the complexity of using SRMR

messaging, and maintains the interoperability of Web services, leaving the framework platform

and language neutral.

References

[1] J-Y Chung, K-J Lin, R.G. Mathieu, Web Services Computing, IEEE Computer Vol. 36, Issue 10, pp 38-44, Oct. 2003.

[2] C. Peltz, Web Services Orchestration and Choreography, IEEE Computer, pp 46-52, October 2003.

[3] T. Mowbray, R. Malveau, CORBA Design Patterns, John Wiley & Sons, Inc, Jan. 1997.

[4] Erich Gamma, et al, Design Patterns, Addison-Wesley Professional; Jan. 1995.

[5] S. Stelting and O. Maasen, Applied Java Patterns, Prentice Hall, Palo Alto, CA, 2002.

[6] H. Kreger, Web Services Conceptual Architecture WSCA 1.0, May 2001, http://www-306.ibm.com/software/solutions/webservices/pdf/WSCA.pdf

[7] T. Bray, et al., Extensible Markup Language (XML) 1.0, W3C Recommendation, Feb. 2004, http://www.w3.org/TR/2004/REC-xml-20040204/

[8] Martin Gudgin, et al, SOAP Version 1.2 Part 1: Messaging FrameworkW3C Recommendation 24 June 2003. http://www.w3.org/TR/soap12-part1

[9] M. Gudgin, A. Lewis, J. Schlimmer, Web Services Description Language (WSDL) Version 2.0 Part 2: Predefined Extensions, Oct. 2004, http://www.w3.org/TR/wsdl20-extensions/

[10] L. Clement, UDDI Version 3.0.2, UDDI Spec, Oct. 2004. http://www.oasis-open.org/committees/uddi-spec/doc/spec/v302.htm

[11] M. Bauer, "Paranoid Penguin: Designing and Using DMZ Networks to Protect Internet Servers", Linux Journal, Vol. 2001, Issue 83, March 2001.

[12] C. Alexander, et al, A Pattern Language, Oxford University Press; New York, 1977

[13] Mailing list used by the WSDL Working Group for technical discussions, http://lists.w3.org/Archives/Public/www-ws-desc/

[14] U. Zdun, M. Voelter, M. Kircher, "Design and Implementation of an Asynchronous Invocation Framework for Web Services"; M. Jeckle; L-J Zhang, (Eds.), ICWS-Europe 2003, Erfurt, Germany, pp 64-78, Sept. 2003.

[15] H. Adams, Asynchronous Operations and Web Services, Part 1. http://www-106.ibm.com/developerworks/webservices/library/ws-asynch1.html?dwzone=webservices

[16] Apache Web service project, Web Service Invocation Framework (WSIF), http://ws.apache.org/wsif/

[17] Y. Goland, M. Nottingham, D. Orchard (BEA) WS-CallBack Protocol (WS-CallBack) 0.91. http://dev2dev.bea.com/technologies/ webservices/WS-CallBack-0_9.jsp

[18] M. Gudgin, M. Hadley, Web Services Addressing 1.0 – Core, Feb. 2005, http://www.w3.org/TR/ws-addr-core

[19] T. Andrews, F. Curbera, et al, BPEL4WS Specification, May 2003. http://www-106.ibm.com/developerworks/webservices/library/ws-bpel/

[20] Nickolaos Kavantzas, et al, Web Services Choreography Description Language Version 1.0, April 2004. http://www.w3.org/TR/2004/WD-ws-cdl-10-20040427/

[21] S. Dalal, et al, BTP, Version 1.0.9.5, Nov. 2004. http://docs.oasis-open.org/business_transaction

[22] M. Brambilla, et al, Managing asynchronous web services interactions, Proceedings of ICWS, San Diego, pp 80 – 87, July 2004.

[23] P. Monday, Web Service Design Patterns, Chapter 14, Apress, Apr. 2003.

[24] G. R. Andrews, Concurrent Programming, Chapter 8, Addison-Wesley Pub., Menlo Park, CA, 1991.

[25] Apache Web service project, Web Services – Axis. http://ws.apache.org/axis/overview.html

[26] Sun Microsystems, The Java Web Services Tutorial, Chapter 1, http://java.sun.com/webservices/docs/1.5/tutorial/doc/index.html

[27] Sun Microsystems, Doclet Overview, http://java.sun.com/j2se/1.3/docs/tooldocs/javadoc/overview.html

[28] B. Atkinson, et al, Web Service Security, Version 1.0, April 2002, http://www-128.ibm.com/developerworks/webservices/library/ws-secure/

[29] T. Moses (ed.), eXtensible Access Control Markup Language (XACML) Version 2.0, Oasis committee draft, Dec. 2004, http://www.oasis-open.org/committees/tc_home.php?wg_abbrev=xacml

[30] M. Fuchs, Adapting Web services in a heterogeneous environment, Proceedings of ICWS'04, pp 656-664, July 2004.

Vita

Michael Edward Ruth was born in New Orleans, Louisiana and received his B.S. from the University of New Orleans in December of 2002. In July 2004, he was awarded the Crescent City Doctoral Scholarship and began working as a Research Assistant under Dr. Shengru Tu. In April 2005, Michael submitted a paper for publication at the International Computer Software and Application Conference 2005, and the paper was accepted for publication.