A Client-Side Framework for Resumable Applications to Utilize Web Services With SRMR Callbacks

Michael Ruth, Feng Lin, and Shengru Tu

University of New Orleans, Department of Computer Science,
2000 Lakeshore Drive, New Orleans, LA, USA
[mruth, flin, shengru]@cs.uno.edu

Abstract: We have designed and implemented a framework assisting client-side applications to utilize asynchronous Web services that deliver results by calling back the applications in the context of enterprise network security. This framework can support a number of rich features such as resumable client applications, the Single-Request/Multiple-Response (SRMR) message exchange pattern, intra-enterprise user and terminal mobility, and flexible interaction styles for client applications to communicate with asynchronous Web services. Practical deployment of this framework in a number of cases has illustrated that this framework can effectively minimize development efforts in facilitating client applications that use Web services with callbacks.

Keywords: callback, correlation, request-response, Web service, WSDL.

I. Introduction

Web services have been a platform to develop applications capable of not only performing business activities on their own but also possessing the ability to engage other web services to complete higher-order business tasks [1, 2]. They have been widely used as interoperable building blocks enabling business processes automation and integration for companies and organizations. Being enhanced by many efforts on research and standards, Web services are in the process of outgrowing from the synchronous request-response model into a flexible distributed computing platform in which the asynchronous service model takes an important role. By asynchronous service model, we mean the client can continue working on other tasks without waiting for the result after it calls a service; the service will inform the client about the results of the call at a later point in time. In contrast, the client has to wait for the result in the synchronous request-response model. However, ever increasing security measures have complicated distributed computing by adding an extra dimension. In a real enterprise installation, a common network security measure, firewalls, often make the client inaccessible from outside the firewall, thus complicating any implementation of the asynchronous service model. The service provider then has a predefined difficulty in calling the client back when the job is done. We believe this client-side accessibility problem will become more and more serious due to at least four trends. First, network security is being tightened everywhere. Second, Web services have been enriching their features and embracing more complicated business processes which require asynchronous interactions and thus callbacks more often. Third, Web services have become dominant in the information services for all kinds of industries, which leads to rapidly increasing service consumers – the applications that utilize Web services. Fourth, consolidation of Web service providers will be inevitable as the technology becomes more mature, which means more client applications will be pure consumers. By pure consumers, we refer to those applications which utilize Web services but are not Web services by themselves.

In a sense, the client-side accessibility problem in Web services very much analogizes telecommunications’ last-mile problem. While the communication speed of the backbone (fiber optic) networks has increased by a thousand times, most end users at their residence are still choked by twisted wire. Even though the WS-Callback [3] and WS-Addressing [4] protocols directly support the callback pattern, and advanced Web service protocols such as WS-CDL [5], BPEL [6] and BTP [7] allow services to interact mutually in any complicated manner, pure-consumer client applications in secure networks are still limited to simple synchronous interactions only. This is because every Web service standard just mentioned assumes that the clients are directly accessible from the services. Unfortunately, this precondition cannot be met by pure-consumer applications in secure networks. Furthermore, these pure-consumer applications may have intermittent connectivity, or may intentionally disconnect from the networks and reconnect later. The requirement for this kind of scenario has been considered more often in business applications, as mobile computing has been reaching more and more business processes. We call this kind of application resumable clients.

We have designed and implemented a client-side framework that enables callbacks from external Web services to those internal pure-consumer applications in secure networks that initiated the call to the external Web services. Compared to polling – an alternative asynchronous interaction manner [8], callback can minimize the involved communication and simplify the design of the clients. Callback is particularly suitable to implementations of remote events and subscribe/publish interactions. Supporting resumable clients will involve additional complexity caused by additional requirements: (1) Time decoupling: The service processes...
can send the results to the client when the client application is inactive. (2) User and terminal mobility: The resumable client should be able to obtain the result from the service, even if the client moves from computer to computer in the secure network.

In our design, we have paid particular attention to a powerful message exchange pattern, the Single-Request/Multiple-Response (SRMR for short) pattern. Many applications can benefit from its use. For instance, a purchase order request may result in multiple sale deals, a document request may obtain multiple files, and a large dataset of the result has to be broken down into multiple manageable pieces. The implementation of the SRMR message pattern firmly demands the asynchronous model such as callback. By optimizing the solution for SRMR, we will also be able to optimize the general asynchronous callback model in the context of the realities of network security. We will also consider the resumable clients that communicate with Web service providers adopting the SRMR pattern.

Our framework has achieved the following objectives: (a) to release the connection to the server instantly after the service accepts the initial call, and before the service result is produced by the server, (b) to avoid deploying complex service choreography servers on the client side, and to allow the use of simple Web servers (such as Tomcat) on the client side that may not support any specific Web service sub-standard, (c) to provide a very flexible mechanism for the applications to consume the callback responses, (d) to extend simple callback to the Single-Request/Multiple-Response (SRMR) message exchange pattern, (e) to decouple the time dependency between the services and the clients and support resumable clients, (f) to support intra-enterprise user mobility and terminal mobility so that a resumable client can continue interacting with the asynchronous service calls even if the client moves from computer to computer in the same network, (g) to assure reliable delivery so that neither the service calls nor the callbacks are allowed to be lost without alarm on the client-side. For reliability-critical systems, a redundant CWS can be deployed. Having hot-swappable Web services is a common practice in heavy-duty Web sites, although we did not implement this in our experiments.

We have deployed our framework to a number of non-trivial applications including monitoring large-scale simulations, military vehicle reconnaissance applications, and purchase order applications. These applications demonstrate the essential benefits of using the framework: satisfactory performance and minimal development efforts.

The remaining part of this paper is organized as the following. Section II provides the background information pertinent to this work. Section III presents the architecture, message flows, and design decisions our framework. Section IV outlines the Web service features provided by our framework. Section V describes the implementation details of our solution. Section VI provides a case study exemplifying the solution and more usage details. Section VII discusses two additional example applications. Section VIII gives a brief discussion about the communicational and computational overhead with experimental data. Section IX highlights the related works. Finally, Section X concludes.

II. Background

In the realm of network security, firewalls, proxies, and Demilitarized Zones (DMZs) [9] 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 received from untrusted networks. A DMZ 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 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, a call going out of the enterprise, being routed through the proxy will carry the address of the proxy instead of the client’s address. Only the synchronous calls made through the proxy can receive a response.

III. The Client-Side Framework

![An Architectural Overview](image)

**Figure 1. An Architectural Overview**

In design of global, distributed software, loosely-coupled systems usually lead to better scalability. It is often desired to achieve decoupling in time, flow, and space. Space decoupling refers to the interacting partners not needing to know each other. Time decoupling refers to the interacting parties not needing to be actively participating at the same time. Flow decoupling is defined as the calling and called parties not being blocked by the calls [10]. For instance, message passing is asynchronous for the producer, while message consumption is generally synchronous. With the traditional callback pattern, the interacting parties are
decoupled in flow, but coupled in time and space.
In this section, we describe the architecture of our framework [11]. The key component of the framework is a centralized correlation-proxy service, called the clearinghouse service (CWS for short). This is a Web service that accepts the callback messages from external callback Web services, and forwards them to the registered calling applications. For each client-side internal network, only one CWS is needed. To encapsulate complexity from the client applications, we developed an agent component that assists client applications that wish to interact with external callback Web services.
Figure 1 illustrates an overview of the architecture of the framework. This diagram illustrates that the CWS is shared by multiple applications. Each application may invoke multiple external Web services that will later perform callbacks.

A. The Process
The UML communication diagram in Figure 2 describes the process when an application makes a call to an external Web service (server service for short) using our framework. Before an application calls an outside Web service, the application instantiates an agent object.

Figure 2. Communication Diagram (observer/observable)

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 way that the framework will behave in notifying the application (to be explained in detail in Section III.B). 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 the arrival of every individual response. In step 1.2, the agent then calls the corresponding operation on the server on behalf of the client. This call takes the operation parameters and a Server Observer object. This object holds the address information (the URI) of the CWS where the responses should be sent. 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 a correlation identifier, or CID, and the number of responses the server will eventually send. Based on this number, the agent maintains a counter to track messages that have been received and not yet received. This response is packed into a Multiresponse object that carries the two values.

In step 1.4a, the agent registers at the clearinghouse, which takes four parameters: The CID and response count, a Client Observer object, and the notification style. The Client Observer object holds the address information of the agent object, in the form of hostname and port. In step 1.4b, the server finishes processing the request and begins returning responses to the CWS at some point. The number of messages that will be sent to the CWS was reported to the agent in step 1.3. Each response consists of a CID and a partial payload. In Step 1.5a, the agent returns the CID to the application. In step 1.5b, which occurs when a response is received for a registered CID and it is time for a notification, the CWS informs the agent that registered for that CID. This notification takes the form of a socket call. Depending on the notification style, the agent will be notified somewhere between zero and response count times. Note that two pairs of steps can happen concurrently: (1) steps 1.4a and 1.4b, (2) steps 1.5a and 1.5b.

Figure 3. Communication Diagram (polling)

In steps 2.1 and 2.2, the agent queries the CWS for the results, and receives all responses that have arrived at the CWS up to that point. In step 3, the agent notifies the application, 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 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/Observable pattern. The application may also poll the agent for specific results. In doing so, steps 2, 3, and 4 would be different. This process is shown in the communication diagram in Figure 3. In step 2.1a, the application could start polling the agent. When the result of those polls returns true, it gets the result as it did in step 4. It is always possible for the client application to reach to a point when certain result is required to continue processing. At this point, the client application has to start polling for the results.

Since every step is a part of a simple synchronous request-response Web service call, each step is reliable. If any single synchronous call fails, the SOAP built-in error reporting will trigger the error handling of the CWS and the agents. Thus, the accomplished asynchronous interactions are reliable.

In Figures 2 and 3, an agent communicates with the remote Web service and the CWS in a triangular way consisting of steps 1.2, 1.4a and 1.4b. An alternative way to support callbacks across firewalls could be the proxy approach. Rather than directly calling the Web service, the client would call the proxy. In the call, the client may provide the proxy server with the information about the future incoming messages (response) such as the sender’s URI. We did not apply this approach to our framework for at least two reasons. First, some information (such as the response count) about the callbacks is not known until the initial response comes back from the remote Web service (step 1.3). Second, if the proxy server wants to be an intelligent proxy and to carry out all the functionality that our agent object does, it would have to understand every protocol that can exist between the client applications and the remote Web services because the intelligent proxy would act as a client of all the remote Web services. Considering there are many possible protocols used by Web services on the market such as WS-Call back [3] and WS-Eventing [12], the intelligent proxy server would be overly complex, which goes against our objective (b) described in Section I. Using the CWS, the remote Web services with callbacks act as clients of our CWS. Therefore, the interaction protocol between these servers and the CWS is determined by the CWS. As a result, the protocol between the remote Web services and the CWS is independent of the protocol between the remote Web services and the client applications. It is easy for the agent to comply with the remote service’s protocols because each agent object is generated for each application according to the WSDL specification of the specific Web service.

Because messages could be modified or forged in transmission, every newly proposed WS-sub-standard such as WS-Transaction [13] strongly recommends that process implementations use WS-Security to ensure messages’ integrity. This comprehensive security model directly supports referencing other message elements with signature references or correlating signatures with security tokens. We plan to apply these security measures in the near future.

B. The Clearinghouse Web Service (CWS)

The clearinghouse is a dedicated Web service that accepts the callback messages from (external) Web services, correlates the callback messages with the original caller application, and forwards the callback messages to the original caller application. The idea of correlation by a clearinghouse was proposed for CSP-like communication in [14]. Our decision to use this centralized listener Web service (the CWS) rather than having one listener Web service per application is based on a number of considerations. First, the central listener service decouples the timing relationships between the calling applications and the callback services. The clearinghouse will accept the callback messages on behalf of applications that may or may not be active. Second, management of the clearinghouse is simpler than the management required to create and maintain one individual listener Web service for each callback Web service application. Third, it more simply handles the security issues and incorporates with the firewalls.

On the other hand, centralizing the listener web service introduces a certain complexity in correlating the callback messages from the external services to the applications that requested these messages. The CWS consists of four operations: the registration operation, the deregistration operation, the send-result operation, and the fetch-result operation.

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. To prevent any possible identical CIDs produced by different service providers, each stored CID is concatenated with the service provider’s URI.

The service providers call the send-result operation to deliver callback messages. This operation takes two parameters: a CID and a payload object. Each time the send-result operation is called, the clearinghouse checks for a match between the CID carried by the just arrived callback message and the CID that a registered client is expecting. If no match is found, the callback message along with its CID will be stored in the clearinghouse database. Once a match is found, the clearinghouse will notify the matched agent about the arrival of results, depending on the associated notification style. A similar matching search is also carried out whenever an agent registers at the clearinghouse. This search is to check for callback messages that have already arrived at the clearinghouse before the agent registers. Once informed, the agent is free to pick up the messages from the clearinghouse by calling the fetch-result operation.

This call has one parameter, the CID. The clearinghouse will return to the calling agent all the correlated messages that the clearinghouse has received thus far. Once the response count for a CID drops to zero and the callback messages for this CID have all been fetched by the corresponding agent, the clearinghouse will purge the corresponding registration entry from the clearinghouse database.

C. The Agent

The main purpose of the agent is to help the application get the results from the clearinghouse for each callback message. Each application that utilizes callback Web services instantiates an agent object. If the application prefers to use the Observer/Observable relationship, the application passes
itself as an Observer object to the callback agent’s constructor. On the other hand, if we want the application to poll for the results, we instantiate the agent using a zero argument constructor. The agent is responsible for performing service requests and receiving service responses from the called Web services.

Immediately after its instantiation, the agent creates a Helper object that will hold a server socket when necessary. Once the agent registers at the clearinghouse, the agent informs the helper to start listening for a notification from the clearinghouse. 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. The agent parses it to get the CID. The agent then calls the fetch-result operation of the clearinghouse using this CID. This call will return all the responses (callback messages) that have arrived at the clearinghouse associated with that CID since the last retrieval by this agent.

The agent then updates its count of the returns. The callback agent and the polling agent will work differently. Using the Observer/Observable relationship, the callback agent will notify the application. Then, the application calls the getResults method of the callback agent to retrieve the results. The polling agent will wait for the application to call Method moreResponses, which lets the application know whether or not there are more results for a given CID at the agent.

IV. Supporting Web Service Features

In this section, we describe the features that are supported by our framework. They include the SRMR messaging, choices of the notification styles, resumable applications, user and terminal mobility.

A. Single-Request/Multiple-Response (SRMR) Messaging

The Single-Request/Multiple-Response (SRMR for short) message exchange pattern can often model real-world problems elegantly, and reduce the number of messages for requests. The WSDL 1.2 specification included two message exchange patterns named Out-Multi-In and In-Multi-Out; they were for the SRMR semantics [15]. This indicates that the SRMR message exchange pattern is useful. However, these two patterns were removed from WSDL 2.0 [16]. Consequently, when this type of interaction is needed, we have to implement it at the application level.

To implement this pattern, the server returns a response count along with the Correlation Identifier (CID) to the requestor – the agent [17]. As described in step 1.4a in Section III.1, the agent sends the response count to the CWS when it registers. Both the agent and the CWS keep their own counters to track transmitted messages. When the results start arriving in the callbacks from the Web service, the CWS updates its counter variable corresponding to the CID. At some point, the clearinghouse notifies the agent according to the information in the clearinghouse database. The agent then gets the results and updates its own counters for that CID. Both the clearinghouse and the agent will not stop expecting messages until the counter meets the response count.

B. Notification Style

Our framework provides three choices of notification styles which give the application flexibility to express its level of urgency by telling the framework if it wants to be notified for every part of the response, upon receiving some parts (N), or upon receiving all parts [18]. Notification of the occurrence of every part of the response is suitable to applications such as purchase order systems. If the purchase order includes many big items, the financial office wants to process each payment for the items upon arriving. Notification upon receiving some part of the response is useful when the result is huge, but the client application can only accept it in a piece by piece manner. In large scale simulations, the data providers and the simulation engine often have such a relationship. Notification upon receiving all the parts is for those clients who can only process the result in its entirety. The notification style does not affect the behavior of the server Web services. The pattern in force between the server services and the CWS is always the SRMR message exchange pattern as described earlier.

C. Resumable Client Applications

As mentioned earlier, the client applications that consume Web services may have intermittent connectivity, or may intentionally disconnect from the networks and later reconnect because it is aware that the requested computation will take a long duration. This kind of application is the resumable client.

Specifically, we have considered the resumable clients that communicate with Web service providers in an asynchronous manner, and expect callback responses from the Web services [19]. Supporting resumable clients involves an additional complexity caused by the additional requirements, namely partial time decoupling. Although when an application calls a Web service the service must be active, the application and agent can be inactive when the server service calls back because the call will be received and stored by the CWS. Thus, the service processes can send the result regardless whether the client application is active.

In our framework, when the application wishes to shutdown, it informs the agent to shutdown. The agent object will first un-register from the CWS, close its socket, and then serialize its own state into an XML document (a large string). Then it is up to the application to determine how to store this XML document. One solution is to store this XML document at a commonly accessible site. Along with the CWS service, we provides another simple Web service, AgentStore, to keep track of the relationships between the applications and the stored agent state XML documents (to be explained in Section V.E). When the application reactivates at some later time, the application first calls the login operation of the AgentStore service and obtains the identifier of its previously stored agent state information. Using this identifier, it then retrieves the XML document representing the agent’s state from AgentStore. The application can then re-instantiate its agent object using the XML document just retrieved. Once the agent is instantiated, it re-registers at the clearinghouse with all the CIDs that are still outstanding. As mentioned in Section III.B, the registration operation will trigger the CWS to find all the matching callback messages for the application to fetch.

It should be pointed out that the actions of shutting down and resuming have to be initiated by the applications themselves,
considering the agent object is just an object in the process of the application.

D. User and Terminal Mobility

User mobility and terminal mobility are special cases of resumable clients. User mobility refers to those applications in which the user moves from terminal to terminal; terminal mobility refers to those applications in which the terminal relocates to a new location along with the user. While there have been a number of researchers delving into supporting user and terminal mobility, very little has been done in the realm of Web services.

In the case of user mobility, research has focused on two aspects: personalization or contactability [20]. Personalization addresses the issues associated with personalizing a user’s operating environment regardless of the terminal being used. Contactability addresses the issues associated with enabling a user to be contacted regardless of the user’s location and device. In this framework, we will only discuss the issue of contactability since we are only interested in having the client application receive the callback response.

In [21] the authors provide an analysis of mobile applications’ requirements for publish/subscribe middleware. The single most important requirement that pertains to the framework presented in this paper is the account auditing function, i.e. subscriber accounting. Subscriber accounting is performed by the agent object in our framework. When an agent unregisters and registers, it provides the CWS with up-to-date information regarding its state in terms of its current residing address.

[22] discusses the requirements for handling the stored sessions (results that have not yet been retrieved), which are session distribution or session relocation. Session distribution refers to distributing the session across the domain, and session relocation simply moves the session to the new access server. JEDI uses a distributed event dispatcher that has moveIn and moveOut functions that perform similarly to our register and unregister functions provided by the CWS. The ToPSS system uses a high availability centralized event dispatcher. As explained in Section III.B, our framework has chosen the centralized event dispatching.

Another key issue in supporting user mobility is for the client application to use the same agent object when the client application resumes. This was solved in our framework by using an additional Web service, AgentStore. It could also be solved by using a SmartCard as presented in [23].

Our framework allows the clients to obtain the result from the service in two mobile cases: (1) the client moves from computer to computer in the same network – the terminal mobility case; and (2) the application along with the hosting computer moves around through the wireless connections in the same network – the user mobility case [19].

Supporting terminal mobility is simple for the CWS. When the agent reregisters, the CWS takes the newly registered address and send the notification to the newly registered agent. Supporting user mobility is more complicated. The key issue is for the client application to fetch the same agent object when the client application resumes. This was solved by adding an additional Web service, AgentStore (to be explained in Section V). By using this Web service, the process is completely automated; no human interaction such as surfing to another Web page is needed.

V. Implementation

In this section, we will discuss the implementation details regarding the interface of the CWS, the generation of the framework, communication between the agent and CWS, generalization of the results, and how the agent maintains its state between invocations.

A. The Clearinghouse Web Service (CWS)

The core of the clearinghouse is the clearinghouse Web service, CWS. The four operations of CWS outlined in Section III.B are listed below. The first three are for the agents, and the last is for the callback server services to deliver results by calling back. The port type and operation names in the WSDL of the clearinghouse service are given in Figure 4.

```java
boolean register(String CID, int responseCount, 
ClientObserver co, String 
commStyle)

boolean unregister(String CID)

Object[] retrieveObjects(String CID)

boolean returnObject(String CID, Object msg)

<wsdl:operation>

<wsdl:input message="impl:registerRequest"

name="registerRequest"/>

<wsdl:output message="impl:registerResponse"

name="registerResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:unregisterRequest"

name="unregisterRequest"/>

<wsdl:output message="impl:unregisterResponse"

name="unregisterResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:retrieveObjectRequest"

name="retrieveObjectRequest"/>

<wsdl:output message="impl:retrieveObjectResponse"

name="retrieveObjectResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:returnObjectRequest"

name="returnObjectRequest"/>

<wsdl:output message="impl:returnObjectResponse"

name="returnObjectResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:notifyObjectRequest"

name="notifyObjectRequest"/>

<wsdl:output message="impl:notifyObjectResponse"

name="notifyObjectResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:returnObjectRequest"

name="returnObjectRequest"/>

<wsdl:output message="impl:returnObjectResponse"

name="returnObjectResponse"/>

</wsdl:operation>

<wsdl:operation>

<wsdl:input message="impl:notifyObjectRequest"

name="notifyObjectRequest"/>

<wsdl:output message="impl:notifyObjectResponse"

name="notifyObjectResponse"/>

</wsdl:operation>
```

Figure 4. Port type and the operations CWS

We have developed the CWS using Apache Axis [24]. Using an Axis utility tool, WSDL2Java, we generated the CWS stub and skeleton and their supporting classes. Among them, the WebServiceClearinghouseSoapBindingStub class is for agent objects to call the CWS. Since the CWS is shared by
every application in the same secure network, the same CWS stub will be used by all the agent objects of any applications.

B. The Agent and Supporting Classes

The common functionality and information used by the agent objects are encapsulated in four classes listed below.

AgentHelper – A class used by agent objects to listen (on a socket) for updates from CWS. It interacts with the agent object with the observer/observable relationship;

ClientObserver – A class used to pass data to CWS that contains the listener information from the AgentHelper object such as the host name and the port number;

ServerObserver – A class used to pass data to the server that contains address of the CWS;

MultiResponse – A class for passing data from the server to the agent objects that contain the information about a request (responses, CID).

Since an agent object has to interact with the called Web service, the code of the Agent class depends on the Web service with which the agent interacts. We have constructed a code generation utility to provide a solution for application developers who wish to use the Web services with callbacks. The agent code is generated based on the WSDL document of the Web service and the schema specification (an XML file) of the return type of the operations of the service. These two specifications are typically provided by the service providers. As an example, the port type and operation of a Web service with callbacks for a purchase order site is given in Figure 5.

```xml
<wsdl:portType name="AgentDataStore">
  <wsdl:operation name="submitPurchaseOrder" parameterOrder="in0 in1">
    <wsdl:input message="impl:submitPurchaseOrderRequest" name="submitPurchaseOrderRequest"/>
    <wsdl:output message="impl:submitPurchaseOrderResponse" name="submitPurchaseOrderResponse"/>
  </wsdl:operation>
</wsdl:portType>

Figure 5. Port type and the operations of a service
```

C. Client-Side Local Communication

Since the clearinghouse and the applications are in different computers, we should carefully consider the communication between them. When the CWS receives a service result (a callback message) and found its correlated party (an agent), it is time to notify the agent (depending on notification style). The clearinghouse informs the agent using a callback mechanism in order to minimize latency. The clearinghouse sends the CID to the matched agent.

Furthermore, we have to choose a communication means for the clearinghouse to notify the applications. In our solution, we have chosen to use sockets for their simplicity and interoperability, since sockets exist in every modern system and are supported by every modern programming language environment. To notify an agent, the clearinghouse opens a socket, and passes the CID as a string to the agent. The clearinghouse will not notify the same agent regarding the same CID again until the agent takes an action upon the previous notification. Thus, unnecessary notifications are eliminated, and the overall network traffic is reduced.

D. Serializing/Deserializing Payloads

In order to support applications to use Web services from any providers, the CWS must be able to handle any type of payload in the response messages. More specifically, the return type should be generalized into an acceptable form so that only one CWS will ever be needed. 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. The service provider should serialize (marshal) the response into an XML string, and pass the string to the CWS. When the agent obtains the string as a result of a call to the CWS, the agent has to deserialize (unmarshal) the XML string into an object that the agent's language can access. This allows not only for the return type to be generic, but also allows it to be language neutral. To achieve this, we used the XML serialization capability of the Java Architecture for XML Binding (JAXB) Reference Implementation based on the XML schema specification of the return type. The JAXB libraries require an XML schema of the converted object.

```xml
<wsdl:portType name="AgentDataStore">
  <wsdl:operation name="save" parameterOrder="in0 in1"/>
  <wsdl:operation name="load" parameterOrder="in0"/>
  <wsdl:operation name="login" parameterOrder="in0 in1"/>
</wsdl:portType>

Figure 6. Port type and the operations of AgentStore
```

E. Resumable Clients and User/Terminal Mobility

In addition to CWS, we have also developed another Web service, AgentStore. The key issue of supporting user mobility and continue the applications’ business process across multiple computers is for the client application to fetch the same agent object when the client logsins at different computer and tries to resume the application. Thus, the Agent’s state must be stored persistently. This was solved by the addition of a Web service with this functionality: the AgentStore service, which has three operations as listed below: save (stores the state of the Agent with a given application specific string which will be used to retrieve the Agent later), load (retrieves the state of Agent identified by the application specific string), and login (return the identifier (AppID) to the calling application that attempts to resume). The port type and the operations are specified in...
the WSDL for AgentStore in Figure 6. The operations login and load must be called by the applications directly. Based on this WSDL document, we used the WSDL2Java tool provided by Axis to generate the skeleton, the stub and other classes.

```java
boolean save(String AppID, String stateXML)
String load(String AppID)
String login(String username, String password)
```

F. Code Generation

The procedure to generate the agent class and the classes required to use the AgentStore service consists of multiple steps as shown below.

**Step 1** Parse the WSDL document of the service provider’s Web service in order to find out the following information: (We used a DOM parser.)
- The value of the attribute “targetNamespace” in the “wsdl:definition” element – This is necessary for generating the Java code from WSDL correctly using the WSDL2Java tool;
- The value of the attribute “name” in element “wsdl:portType” – This is the class name that Axis will generate.

**Step 2** Create the stubs using the WSDL2Java tool that the agent will use to call the server web service, and compile the generated code.

**Step 3** Run a docket on the interface created in step 2 to get the following information:
- method names;
- parameter list expanded (with types);
- parameter list short (without types).

**Step 4** Parse the XML schema document of the return type and look up the value of the attribute “name” in the “xsd:element” element. (This will be the class name of the XML serialization code generated by JAXB.)

**Step 5** Create code for marshalling and unmarshalling the return type object and SOAP message, using JAXB based on the XML schema document.

**Step 6** Copy the AgentHelper.java file to the corresponding directory.

**Step 7** Generate the Agent class based on the information gathered in steps 1, 3, and 4, using our utility program JavaTemplate.

**Step 8** Parse the WSDL document of the AgentStore service and look up the value of the targetNamespace attribute in the “wsdl:definition” element.

**Step 9** Generate the stub classes of the AgentStore using the WSDL2Java tool based on the WSDL document of AgentStore.

**Step 10** Compile the generated code.

VI. A Case Study

In this case study, we will discuss the implementation of a purchase order system using our framework, as well as some additional details of the framework. This system creates purchase orders (POs), and submits them to the “accept order” operation of the vendors Web services. A purchase order is typically composed of several different items. The purchasing application may have many orders to submit and does not wish to wait for the vendor to fulfill each order, especially considering that these orders may take long time to fulfill. Instead of waiting for the vendor to complete each order as they are submitted, the application would prefer that the vendor accepts the submission, unblocks the application, and uses a callback to notify the application of the results. Also, if the items being submitted are handled by different departments within the vendor company, each component of the purchase order that requires special handling needs to go to its specific department. When each part of the purchase order is fulfilled, the handling department sends the results directly back to the buyer’s purchasing application. The purchase department is willing to handle each response individually upon its arrival. This example is representative of practical purchase order systems and an example of how a request may lead to multiple responses.

This case study is a good example of: (1) an application that would benefit from callback because of long duration calls, (2) an application that would benefit from SRMR messaging, (3) an application scenario that would benefit from user mobility considering someone who submits purchase orders frequently that has to move from office to office using available computers on the spot, (4) an application scenario that would benefit from terminal mobility considering someone who submits purchase orders frequently that has to move from office to office with his laptop.

The purchase order vendor provides their service WSDL specification listed below.

```xml
<?xml version="1.0" encoding="UTF-8"?>
<wsdl:definitions targetNamespace="urn:POSubmissionServer" ... >
<wsdl:types>
<schema targetNamespace="http://shared"
 xmlns="http://www.w3.org/2001/XMLSchema">
<import namespace="urn:POSubmissionServer"/>
<import namespace="http://schemas.xmlsoap.org/soap/encoding/"/>
<complexType name="PurchaseOrderItem">
<sequence>
<element name="description" nillable="true" type="soapenc:string"/>
<element name="name" nillable="true" type="soapenc:string"/>
<element name="quantity" type="xsd:int"/>
<element name="vendor" nillable="true" type="soapenc:string"/>
</sequence>
</complexType>
<complexType name="PurchaseOrder">
<sequence>
<element name="POId" type="xsd:long"/>
<element name="college" nillable="true" type="soapenc:string"/>
<element name="dept" nillable="true" type="soapenc:string"/>
<element name="items" nillable="true" type="impl:ArrayOf_tns1_PurchaseOrderItem"/>
<element name="notes" nillable="true" type="soapenc:string"/>
<element name="submitterName" nillable="true" type="soapenc:string"/>
</sequence>
</complexType>
</schema>
</wsdl:types>
</wsdl:definitions>
```
As explained in Section V.F, our utility program generated the Agent class and the supporting classes are shown by the class diagram in Figure 7. All the objectives outlined in Chapter IV were achieved with very limited coding effort. As described in Section IV.C, we instantiated a callback agent object that is capable of performing these asynchronous, SRMR calls for the purchasing application. The application and agent are related using the Observer/Observable relationship. When the application calls the “accept order” operation through the callback agent, it specifies a notification style, “individually”. The agent instantiates a ServerObserver object containing the URI of the clearinghouse associated with the purchasing application, and then calls the vendor’s Web service. As explained in Section IV.A, the “accept order” operation immediately analyzes the order and replies with a MultiResponse object, which contains this call’s correlation ID (CID) and the number of responses. The agent then registers at the...
clearinghouse with a Client object which contains the host name, port number, the CID, the number of responses, and the choice of notification styles, which is "individually" in our case.

Figure 8. Deployment Diagram of PO System

import shared.*;

public class POSystemCallbackApp implements Observer, Runnable {

    private POSubmissionServerAgent client;

    // Constructor
    public POSystemCallbackApp () {
        this.client = new POSubmissionServerAgent(this);
    }

    // Observer method
    public void update(java.util.Observable o, Object arg) {
        long ticket = ((Long)arg).longValue();
        PurchaseOrderConfirmation[] pocs = client.getResults(ticket);
        for (int j=0; j<pocs.length; j++) {
            PurchaseOrderConfirmation result = pocs[j];
            //utilize result …
        }
    }

    public void run() {
        boolean toContinue = true;
        while (toContinue) {
            try {
                PurchaseOrder po = new PurchaseOrder(poID, …);
                long ticket = client.submitPurchaseOrder(po, "individual");
            } catch (Exception e) { … }
        }
    }

    public static void main(String[] args) {
        Thread app = new Thread(new POSystemCallbackApp());
        app.start();
    }
}

Figure 9. A Minimal PO Client Application

The purchase order system creates purchase orders (POs) and submits them to an "accept order" operation of a Web service provider. Figure 8 shows a deployment diagram of the resulting system. In this system, the application and agent are related via the Observer/Observable pattern. The system follows the process outlined in Chapter IV.

Once the above code generations are done, the client application requires very limited coding efforts. For example, the code listed Figure 9 is sufficient to carry out the process in a minimum way. Note that the generated agent class and its supporting classes are in package shared.

The approach to generalization of the return type using the JAXB library was explained in Section V.E. Figure 10 shows the XML schema that defines the return type of a purchase order 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>
</xsd:schema>

Figure 10. XML Schema of PurchaseOrderConfirmation Return Type

In our implementation, the service provider was also implemented in Java and used the generated code for marshalling/unmarshalling the return type. The code used by the service provider to marshal the return type into an XML string is shown in Figure 11. The process shown in Figure 11 is as follows: First, a JAXBContext is created using the JAXB code generator, 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 ByteArrayInputStream is instantiated to marshal the output. The object is then marshalled into the byte stream and the payload string is created from the byte stream.

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 11. Marshalling an object into XML

JAXBContext jc = JAXBContext.newInstance( "shared" );
Unmarshaller u = jc.createUnmarshaller();
ByteArrayInputStream inputStream = new ByteArrayInputStream(payload.getBytes());
PurchaseOrderConfirmation pocIn = (PurchaseOrderConfirmation)u.unmarshal(inputStream);

Figure 12. Agent Unmarshalling XML into a POC object

Figure 12 shows the code used by the agent to unmarshal the XML string into a POC object. The process outlined by Figure 12 is as follows: A JAXBContext object is created using the "shared" package 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. The code generated by JAXB is in packages shared.impl and shared.impl.runtime.

VII. Additional Examples

In this section, we will discuss two additional applications of our framework: A simulation monitoring application, and a military vehicle reconnaissance application.

A. 5.1. Monitoring Large-Scale Simulation

Large-scale simulation tasks often require multiple gigabytes of data and computation in the order of hours and days. In our experiment, we used the Parallel Advanced Circulation Model (PADCIRC), a finite element hydraulic model for coastal areas [25], to simulate the hurricane flooding in the New Orleans area. It requires over 20 gigabytes of datasets including the hydraulic data, geographic data, and the weather data in the area during the storm. The datasets were first loaded into a 72-node computer cluster. The computation took between 8 hours to 6 days depending on the chosen grid sizes and the length of the period. From time to time, PADCIRC produces large resulting data sets that need to be interpreted by other programs. Many times, the results after days of computation were invalid due to some small mistakes in the complex data loading. A reason for this was that the application users, the hydraulic engineers, were located in another organization. If the error could be detected earlier, a lot of computational time and engineers’ waiting time could be saved.

Using our framework, we developed a remote monitoring system to deliver early warning to the hydraulic engineers. On the computer cluster site, we implemented a Web service named PMmonitor supports the three operations:

- MultiResponse distributeData(Dataset da, int numNode, String distributionManner, ServerObserver so)
- JobInfo getJobInfo(String jobID)
- MultiResponse simulate(JobInfo jobInfo, ServerObserver so)

Both the distributeData and the simulate operations are long-duration ones. Therefore, they are designed to deliver results by calling back; the callback address is given by their last parameter serverObserver which contains the parameter callbackURI. Operation distributeData kicks off the data distribution phase for the PADCIRC program. This operation immediately returns a MultiResponse object including the expected number of responses from the server and a unique job identifier (jobID) which will be used in the life cycle of the entire job. As mentioned in Section V.B, class MultiResponse is defined in the package shared that encloses the Agent class. For distributeData, the expected number of responses is one because we do not invoke simulation until the data distribution phase is completed. The meanings of the four parameters are listed below.

- da – the identifier of the dataset. Since the simulation datasets are typically complicated, the identifier of the dataset is an object of type Dataset.
- numNode – the desired number of nodes to carry out the simulation. The actual number of deployed nodes may be different from this requested number due to occupation of other on-going jobs. The actual number of deployed number will be reported back to the client application in an object of JobInfo type.
- distributionManner – the choice of the data distribution manners from a set of predefined ones such as “equal size” or “proportion in CPU power”;
- so – a ServerObserver object. Class ServerObserver is in the package shared that encloses the Agent class. This class was explained in Section V.A. An important aspect of so is the callback address. As explained in Section III.A, this is the URI of the CWS at the network where the client application resides.

Operation getJobInfo is a traditional synchronous operation. It takes one parameter, the jobID (obtained from a call to the distributeData operation), and returns a comprehensive object that carries all the details of a simulation job including the jobID, the actual data allocation status, the chosen data distribution manner. This object is particularly important for the client application to continue correctly executing simulation after the data distribution is done. Before data distribution is done, getJobInfo returns null.

Operation simulate actually starts simulation. This operation takes two parameters, the jobInfo and the callbackURI (the same as that in operation distributeData). The jobInfo is an object of type JobInfo which is typically obtained from a call to the getJobInfo operation. Similar to operation distributeData, simulate also returns a MultiResponse object. However, in contrast to distributeData, the value of the expected number of responses is many.

On the client side, we wrote a simple application for the hydraulic engineers that controls and monitors the simulation job. The PADCIRC program has two phases: data distribution and simulation. Since the computational results of the PADCIRC program, the intermediary or the final ones, are so massive that a separated interpreter program, PMValidator, is available for validating the results. The life cycle of this large-scale simulation can be summarized into the following steps.

- Step 1 distribute data into each node of the cluster;
- Step 2 wait until data distribution is done;
- Step 3 get the actual data distribution status;
- Step 4 kick off simulation;
- Step 5 repeat {
  fetch the result of certain amount of data;
  validate the (intermediary) result;
  if (data is invalid) halt simulation and report error
}

A sketch of our simulation monitor is outlined in Figure 13. The PMmonitor web service has three operations. The getJobInfo operation is a regular synchronous operation; no agent is needed. Since the operations distributeData and simulate return different types, the client application has to create two agent objects, pollAg (Line 10) and cbAg (Line 11), respectively. These two agent objects differ in two aspects. First, they handle different return types and are of
two different classes which were generated by our code generator. These two classes are `PMdistributeDataAgent` (used in Lines 3 and 10) and `PMsimulateAgent` (used in Lines 4 and 11). Second, they interact with the application (the PADCIRCMonitor object) in different patterns: polling and calling back. As explained in Section III.C, by calling the different constructors (without parameter and with a parameter), PADCIRCMonitor obtains one agent (pollAg) for polling and another agent (cbAg) for calling back.

In steps 1 and 2, PADCIRCMonitor has to wait for the completion of the distribution operation, and can do nothing else. Thus, PADCIRCMonitor polls on agent pollAg (Lines 24 and 25) to enforce the logical synchronization constraint at step 2. On the other hand, since result validation is performed by a different program, we can spawn multiple threads or processes to speed up validation, when results are available. Furthermore, the computational time may vary dramatically even for simulating the equal amount of data. Therefore, PADCIRCMonitor utilizes the callback agent cbAg for the operations in step 5 (Line 28).

```java
1 import shared.*;
2 public class PADCIRCMonitor implements Observer, Runnable {
3  private PMdistributeDataAgent pollAg;
4  private PMsimulateAgent cbAg;
5  PMValidator va;
6  Dataset da;  // information about the input data
7  // constructor
8  public PADCIRCMonitor(Dataset da, …) {
9     PMValidator va = new PMValidator(…);
10     pollAg = new PMdistributeDataAgent();
11     cbAg = new PMsimulateAgent(this);
12  }
13  // observer method
14  public void update(String[] args) {
15     SimResult res = (SimResult)arg;
16     // validate result ...
17     if (ta.validate(res)) {
18         // report error and halt the simulation
19     }
20     // the process
21     public void run() {
22         try {
23             long cID = pollAg.distributeData(da, 64, ...
24                 "equal-size", "all");
25             while (!pollAg.isResultReady(cID))
26                 Thread.sleep(napTime);
27             MultiResponse mr = pollAg.getResult(cID);
28             JobInfo jobInfo = pmws.getJobInfo(mr.getJobID());
29             cbAg.simulate(jobInfo, "individual");
30         } catch (Exception e) { … }
31     }
32     public static void main(String[] args) {
33         Thread app = new Thread(new PADCIRCMonitor("AndrewDS", …));
34     }
```

**Figure 13** A Sketch of the Simulation Monitor

Since operation `getJobInfo` is a traditional synchronous operation, PADCIRCMonitor directly calls the PMonitor (in Line 27), where pmws is a simple stub of PMonitor generated by Axis. For asynchronous calls, the notification styles are specified in Lines 23 and 28. It is natural to choose "individual" in calling the simulate operation (Line 28) because we want to validate each intermediary result. It is not so obvious why we need to specify a notification style (such as the "all" style in Line 23) when the agent object is for polling only. The style choice is necessary because the agent mode (callback or polling) is about the interaction between the client application and the agent object. However, the notification style specifies the interaction style between the agent object and the CWS.

The result validation is automated in the observer method `update`. Every arrival of the result comes from the simulation program’s calling to the CWS’s `returnObject` operation. The CWS will then notify the corresponding agent. Upon this happening, the observable object, agent’s `AgentHelper` object, invokes the agent object’s `update` method. Consequently, validation is invoked, and the remote hydraulic engineers can monitor the computation effectively.

In our future enhancement, we will allow the simulation to start with a partially loaded dataset. In that case, operation `getJobInfo` will provide the actual status of partial data distribution before data distribution is done.

### B. Military Vehicle Recon Application

A soldier in a vehicle performing recon once in the field, where communication towers are not always nearby, needs send and receive to and from the headquarter the information such as location information, maps, enemy locations, and orders. The soldier needs to send information such as the soldier’s location, and the enemy’s location and expects a response that may take the form of orders, or supporting information, such as maps of the interesting areas. Human interaction is clearly necessary in producing a response on the server side of the interaction, thus this application is a good candidate for the use of asynchronous communication.

The vehicle, however, is not stationary, it may move from location to location and from tower to tower, which could change the address information of the vehicle. Also, a sudden loss of connectivity may occur if a radio tower is damaged. However, no response from the headquarter is allowed to be lost. When the vehicle regains connectivity somewhere else, the responses from its earlier calls must be retrieved. This application is a typical example of terminal mobility.

Using our framework and our code generation utilities, we developed an application that performs very well in the simulated conditions as adverse as those described in the above. The application, running on the device inside the vehicle, determines when connectivity is lost or changed, and shuts down and restarts the agent object accordingly. This shutting down of the agent, forces the agent to unregister and the restarting of the agent, forces the agent to reregister. In this way, the CWS always has the current address of the application on the vehicle when the application is connected to a communication medium. Since the results are stored at the CWS, the results of the calls are not lost while the application was disconnected. If somehow the application couldn’t react fast enough and the agent is shutdown without being able to unregister, the CWS will continue attempting to notify the agent at the same address, if it has results to deliver. When the vehicle is in the range of some tower to reconnect, the application on the vehicle will restart the agent, and the agent will reregister. When the agent reregisters, the CWS will stop trying to notify the agent at the old location and begin attempting to reach the new address.
Using our framework and the related code generation utilities, we have achieved all the design goals of this application with minimal coding effort.

VIII. Performance Considerations

Compared to a synchronous Web service call from a client to the Web service provider (in which the response is transmitted back as part of the same call), the callback interaction pattern requires the service provider to send back the response in a separated call, so that the client is released during the server processes for the called service. The additional call was not a concern regarding performance for intra-network interaction. However, a (remote) Web service call includes an Internet communication. Under what condition this additional communicational cost can be offset by releasing the client application from waiting?

Furthermore, the synchronous Web services directly support an alternative asynchronous interaction pattern – polling. That is, the client would send the request to the service provider in the first call. Then, the client would repeatedly call the server to poll for the result, until the result is available. Compared to a polling client that directly communicates with the server, our clearinghouse approach requires the callback message to go through the CWS and form a triangular path as shown in Figure 14.

![Figure 14. Callback using a clearinghouse](image)

Inevitably, interposing the CWS between the server and the client will cause certain overhead. In this section, we are to analyze the client’s waiting time by comparing our clearinghouse approach against the polling approach in various scenarios. The following analysis considers the SRMR case for generality. We will use the symbols listed below:

- **B** – average callback time
- **CA** – average CWS acknowledging time
- **L** – client-side local network communication time
- **n** – number of responses upon a request
- **ND** – average Internet delay
- **R** – average response time of a synchronous request call to the server without waiting for the result. **R** represents the cost of three kinds of calls: (1) the initial service request in the clearinghouse approach, (2) a polling call without getting the result, (3) a polling call that gets the result.
- **SP** – average server processing time
- **SA** – average server acknowledging time

In fact, **R** = **ND** + **SA** and **B** = **ND** + **CA**. Reasonably we can assume that **ND** and **B** are smaller than **SP**. Otherwise, asynchronous interaction would have little advantage. Also, **SA** is typically very small for powerful server of most service providers. **L** is negligible compared to **ND**. We will omit **SA** and **L** in later analysis. **CA** should also be small. **CA** depends on the Web server that hosts the CWS. Since CWS usually does not receive many hits, achieving small **CA** is very feasible. In our analysis, we have also assumed that the server always process clients’ requests immediately.

The time saved by releasing blocked client depends on the behavior of the client application. For a “busy” client, after the request is sent every moment will be used for other tasks as long as the client is unblocked. On the other hand, a “solely dedicated” client, will just wait for the response even though it is unblocked. In the discussion about these two extreme cases, we will use the following symbols to represent the client average waiting time when different approaches are applied to different types of clients.

- **W<sub>CWS,busy</sub>** – busy clients’ average waiting time using CWS
- **W<sub>CWS,dc</sub>** – dedicated clients’ average waiting time using CWS
- **W<sub>Poll,busy</sub>** – busy clients’ average waiting time polling by the client
- **W<sub>Poll,dc</sub>** – dedicated clients’ average waiting time using polling

First, let us consider the clearinghouse approach. For a busy client, **W<sub>CWS,busy</sub>** = **R** + **n** * **L**. Considering after the first request, the busy client will not stop working on other tasks until each result arrives. For a dedicated client, **W<sub>CWS,dc</sub>** = **R** + **n** * (**SP** + **L**). This is because the client is constantly waiting for the server to process and send back each result; the time of sending back each result (**B**) overlaps with the server processing time (**SP**).

Next, we consider the polling approach. For a busy client that carries out polling in its own control flow, **W<sub>Poll,busy</sub>** = **P** + (**n** + **m**) * **R**,

where **P** is the polling interval and **m** is the number of missing polls. By a missing poll we mean a polling call finds out the result is not yet ready to pick up. If the client has a separated thread carrying out polling, then **W<sub>Poll,dc</sub>** = **P** + **R** + **n** * **SP**. To reduce the client waiting time, one would naturally attempt to decrease **P**. However, if **P** is very small compared to **SP**, then many missing polling calls will be made before getting each result. Thus, **m** will be very large. More importantly, a lot of polling messages will flood over the Internet, and hit the server.

Now, we can give a set of quantitative comparisons between the clearinghouse and the polling approaches.

- **W<sub>Poll,busy</sub>** − **W<sub>CWS,busy</sub>** = **P** + (**n** + **m**) * **R** − **R** = **P** + (**n** + **m** − 1) * **R**
- **W<sub>Poll,dc</sub>** − **W<sub>CWS,dc</sub>** = **P** + **R** − **R** = **P**
- **W<sub>Poll,dc</sub>** − **W<sub>CWS,busy</sub>** = **P** + **R** + **n** * **SP** − (**R** + **n** * **SP**) = **P**

It is clear that from the viewpoint of clients, the clearinghouse approach is better than the polling approach. Considering the working load of the server side and the bandwidth usage of the Internet, the clearinghouse approach can let the client call the server only once for each SRMR request. A polling client will hit the server **n** + **m** times, which
will spawn these many messages over the Internet. To repeat, \( m \) can be very large when \( P \) is small.

<table>
<thead>
<tr>
<th></th>
<th>Marshalling</th>
<th>Unmarshalling</th>
</tr>
</thead>
<tbody>
<tr>
<td>First iteration</td>
<td>63 ms</td>
<td>203 ms</td>
</tr>
<tr>
<td>Later iterations</td>
<td>5 ( \mu )s</td>
<td>6 ( \mu )s</td>
</tr>
</tbody>
</table>

Table 1. Marshalling/unmarshalling time

We also measured the time for the agent and the “remote” Web service components to convert the return objects into XML strings in our experiments. We used Sun Microsystem’s reference implementation of the JAXB API to marshal and unmarshal 1000 different request and response messages. The return objects were each around 5 KB. We found that the main computational cost was for the system to load the needed classes; the actual marshaling and unmarshaling took very little time as shown in Table 1. The CWS, the agent objects, and the “remote” Web services all ran on Sun Java Workstations with 2.39-GHz 64-bit AMD Opteron CPU and 2 GB memory.

A natural concern with the CWS approach is about the bottleneck effect caused by the CWS itself. This potential problem has been largely avoided because we let the client application directly invoke the Web service operation without going through the CWS. Only those callbacks from the Web services will be handled by the CWS first. Considering the percentage of the Web services having callbacks is usually very low among all the Web services, and each CWS handles a client network only, we believe the impact of the potential CWS bottleneck effect will not be common and significant. If the performance is worsened due to the CWS bottleneck, multiple CWS services can be deployed to mitigate the problem.

IX. Related Work

A framework for client to utilize Web services in various asynchronous ways was reported in [26], which suggested for each Web service call to use a thread to maintain the connection and to wait for the response from the server. Thus the client’s control flow will not be blocked. The thread will pass the result back to the application upon receiving the result from the server. 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 on maintaining a connection to the service provider is the major liability of this approach. If the connection is lost, the solution fails. Therefore, this solution does not support resumable clients. Also, the Web server would tend to be overwhelmed by too many concurrent connections, and would be vulnerable to denial-of-service attacks. In our approach, our agent object releases the connection to the server immediately after receiving server’s response to the request.

Deploying a listener Web service on the client side for each callback-client application would truly support callbacks [27]. 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. Another drawback of this approach is allowing these listeners to be accessible dynamically to the outside, which could pave the way for numerous security holes. Our framework has only one listener, namely the CWS. Having only one listener per enterprise allows single point of entry and thus reduces the security risks involved.

By binding the SOAP protocol to a specific transportation protocols that support asynchronous communications, Adams proposed an approach in which the underlying transport performs the needed asynchronous calls and the use of threads to perform the asynchrony [28]. The major drawback of this approach is that would force the developer to use some specific asynchronous protocols such as SMTP or JMS, and such a requirement may not be feasible for some environments. Another characteristic of this particular approach is the total transparency of the underlying callback to the client. While it makes its application easier to implement, 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 [29]. 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 can be a serious limitation to many applications. In our framework, we avoid these limitations. The users of our framework can utilize any underlying transport protocol, adding to the interoperability of the solution.

BEA developed a SOAP based solution, the “WS-Callback” protocol [3]. It defines “standard” headers in SOAP messages which the requestor can use to dynamically specify where to send the response. WS-Callback does not support SRMR because it lacks a built-in correlation mechanism. WS-Addressing, currently being proposed, allows a service request to pass a “reply-to” address of a callback listener to the operation call [4]. WS-Addressing can easily support SRMR, but cannot handle resumable or mobile clients. Therefore, WS-Addressing can be used to handle the response from the server to the CWS, but cannot replace our framework. As of the time of writing, WS-Addressing does not yet have a reference implementation, so it was not used in this work. WS-Callback and WS-Addressing cannot be used to call those pure-consumer applications that are not addressable at the level of Web services. Our framework uses application-level calls to all involved Web services without using any extensional WS- sub-standards. Our framework can be extended to support WS-Addressing when a reference implementation is available.

In the context of Web services, SRMR messaging is associated with asynchronous messaging. In the history, the WSDL 1.2 specification included two message exchange patterns, namely Out-Multi-In and In-Multi-Out [15]. For example, the Out-Multi-In pattern was specified as the following:

“\( \text{It consists of one or more messages, in order, as follows:} \)

- A message indicated by a message reference whose name is ‘A’ and direction is ‘out’.
- Zero or more messages indicated by a message reference whose name is ‘B’ and direction is ‘in’.
- An optional message indicated by a fault reference whose name is ‘C’ and direction is ‘in’.”
This piece of history clearly indicates that the SRMR message exchange pattern is commonly useful. However, the above two patterns were removed from WSDL 2.0 in order to avoid confusion with the multicast-capable patterns [30]. Consequently, when this type of interaction is needed, it will have to be implemented at the application level.

Finally, the “disconnected operation” is a well covered research topic [31]. Disconnected operations refer to those applications that continue working even if communications to the servers occasionally disconnect. A caching solution has been applied to Web services in [31]. With this solution, the response must at some point be received in order to be cached. If each request results in unique response, the cached information will not be useful to produce any response. Although our resumable client feature sounds like disconnected operations, they are fundamentally different. The caching solutions to disconnected operations in [31] are not applicable to resumable clients. While disconnected operations attempt to operate when disconnected, the resumable client is aware of the disconnection, and has a life cycle going through connecting, operating, shutting down, reconnecting and resuming at some later time. On the other hand, disconnected operations and resumable clients can mutually supplement, considering one deals with temporary loss of servers, the other handles temporary outage of clients.

X. Conclusion

It is very unlikely that the client-side accessibility problem with the pure-consumer applications could be solved by a pure standards approach because the client-side applications vary according to business needs rather than according to technological advances. On the other hand, client-side applications share the common need to handle communications with Web services, which can be complex and error-prone. Therefore, the suitable approach to reduction of application development efforts is the framework approach.

We have designed and implemented a feature-rich framework that assists client-side applications to utilize asynchronous Web services that deliver results by calling back in the context of enterprise secure networks. This framework has accomplished all the objectives we set forth to achieve, including supporting the Single-Request/Multiple-Response (SRMR) message exchange pattern, resumable clients, intra-enterprise user mobility and terminal mobility, and flexible interaction modes for client applications to communicate with asynchronous Web services. All of these features ensure reliable delivery. The case study and the two additional examples illustrated that our framework effectively minimized software engineers’ efforts in facilitating client applications’ use of Web services with callbacks.

In our approach, we focused solely on the design and implementation of the client-side framework. The server side design is left to the service developers. Callback is a well-know interaction pattern. The service designers can deploy numerous WS-* sub-standards such as WS-Callback [3] and new standards such as WS-Eventing [32] and the high-level standards such as WS-Coordination [33, 34]. To use our framework, the services need to serialize each response into an XML document, and to send it to the address provided within the call.

In the context of secure enterprise environments, we have addressed the aspect of accessibility, but did not intend to address the end-to-end security measures for Web services because there has been extensive research and therefore, results ready to be applied, such as the WS-Security family [35]. What our framework sought was a means to enable client applications to interact with remote Web services in the way of the callback pattern, without conflicting with any network-level security measure. The core of our framework, CWS, is just a Web service. Any Web service security measures can be added as a layer on top of our framework.

Acknowledgment

This work was supported in part by Louisiana Board of Regents Grant LEQSF(2003-04)-ENH-TR-86, and a research grant from Lockheed Martin Corporation in 2003. The authors particularly like to thank Dr. Marilyn T. Gaska from Lockheed Martin Federal Systems-Owego for her early guidance, discussions and contributions. The authors also like to thank former graduate students Eric Normand and Sriman Kuchimanchi for their early experiments.

References


Author Biographies

Michael E. Ruth received his Bachelor of Science in Computer Science at the University of New Orleans in 2002. In 2003, he was awarded the Crescent City Doctoral Scholarship. He received his Master’s of Science in Computer Science at the University of New Orleans in 2005. He is currently a Research Assistant and working towards his PhD in Engineering and Applied Science at the University of New Orleans. His current research interests include Web services, Distributed Systems, and Software Engineering.

Feng Lin was born in Xiamen, in the Fujian province of China. He completed his Bachelor of Science degree in Electronic Engineering at Tsinghua University in 2003. He is currently working towards a Master’s degree in Computer Science and is a Research Assistant at the University of New Orleans. His current research interests include Web services, Distributed Systems, and Software Engineering.

Shengyu Tu is a professor of Computer Science at University of New Orleans. He received his Ph.D. in Electrical Engineering and Computer Science at the University of Illinois at Chicago in 1991. His research interests include service-oriented computing, enterprise software integration, geographic information systems, static analysis of concurrent programs, and Petri net theory. He is currently serving as a guest editor of the theme issue on GIS Web services of IEEE Internet Computing.

ISSN 1738-6535 © Web Services Research Foundation, Seoul, Korea