Experiences in Deploying Services within the Axis Container

Beytullah Yildiz[1, 2], Shrideep Pallickara[1], Geoffrey Fox[1, 2, 3]
[1]*Community Grids Lab, Indiana University*
[2]*Computer Science Department, School of Informatics, Indiana University*
[3]*Physics Department, Collage of Art and Sciences, Indiana University*
{byildiz,spallick,gcf}@indiana.edu

Abstract

The Web Services framework, since it leverages the Service Oriented Architecture model, enables the development of applications that are loosely coupled and easier to manage. A Web Service is typically hosted within a Web Service Container. There are several choices for these containers depending on the platform and the language in which the applications would be developed. In this paper our focus is on applications built using the Java platform. Here, the most dominant Web Service container is the open-source Apache Axis Web Service container. In this paper we describe our experiences in deploying Web Services, specifically WS-ReliableMessaging, within this container. We enumerate the problems and limitations that we encountered within Axis, and our solutions to get around this problem. We also have a set of recommendations that would make this a more flexible container for sophisticated Web Service applications.

1. Introduction

Web Services provide a distributed computing environment to solve interoperability issues. It is an effort to provide a solution for seamless communications and interoperability. Perhaps what distinguishes Web Services from previous attempts is its leveraging of XML. Support from a large community of vendors/users is another advantage. Companies, universities and organizations are also committing several resources to this effort. This has hastened the maturity of the Web Services framework. Furthermore, this has resulted in several specifications targeting several application domains; with competing specifications within the same application domain.

Whereas Web Services are very promising, we encounter a very important side effect. They performed slower than previous approaches. The cost mainly results from SOAP message and its processing. SOAP is an XML base document and it is self-descriptive as a consequence. Therefore, it contains extra information and data. This increases the size of message and requires more bandwidth on transportation. Moreover, the XML parsing can be quite expensive.

Some specifications lay the foundation for building the Web Service framework. These include SOAP [1], WSDL [2] and UDDI [3]. Web services leverage XML extensively. SOAP is a widely accepted, XML-based format for messaging in Web Services. In order to process SOAP messages and provide a good environment for services; several containers are now available for different platforms and languages. Of interest to us in this paper is the open-source Apache Axis (version 1.2) [4], container for Java-based Web Service applications. Axis is currently the most dominant container within the Web Service community and has a plethora of applications developed around this container.

Axis basically provides three main interfaces viz. Remote Procedure Calls (RPC), document/wrapped and message style communications. In the RPC style, a Java object is serialized into XML and de-serialized back into a Java object at the target point. This is very useful if a Java program, which needs to be deployed, has been already implemented. Document and wrapped style are very similar to each other, but differ in their use of SOAP encoding. The data is encapsulated within a plain XML document. Although serialization and de-serialization operations are not required, binding is needed in this type of deployment. The Message style is a user-defined style and is typically very flexible. Since the message is already an XML document, serializers and deserializers are not needed. There are several scenarios where message style Web services have clear advantages.

In addition to the services themselves, several containers (including Axis) incorporate support for handlers or filters which facilitate incremental addition of capabilities at a service endpoint. An example of a handler is an encryption handler which encrypts messages originating from a client and an inverse-handler at the service side which performs the appropriate decryption. By setting up appropriate handlers (and inverse handlers) in the request and response flows originating from a service endpoint, the endpoint's capability is enhanced without the need for making changes to the application. One typically configures handlers through a deployment descriptor file that is part of the Web Service container. Finally, several handlers could be cascaded together to comprise a handler chain.

Although the Axis architecture provides very good functionalities, there are several areas where we see a need for improvement. We enumerate this below.

1. It is based on the request-response paradigm and does not support message injection in service side.
2. It does not have the ability to gracefully terminate processing related to a message within the handler chain associated with a service.
3. Handler chain has a static configuration.

Request and response paradigm will be discussed in section 2. We will elaborate the necessary mechanisms for a better container and provide test results for a solution in a current Web Service Container. Gracefully stopping a message, while passing through handler chain, is discussed in section 3. Finally, we will discuss static handler chain issues and suggestions in section 4.

2. Improving messaging

Axis container is mainly based on the request and response paradigm. Every message from a client is considered as a request which should have its accompanying response within a pre-defined period of time. This time is set internally by Axis itself and it is not possible to configure this (see Figure 1).

Despite the fact that the request and response paradigm have advantages in many scenarios, they do have limitations. We can classify these scenarios into two categories, those that need only one way messaging and those that require asynchronous messaging.

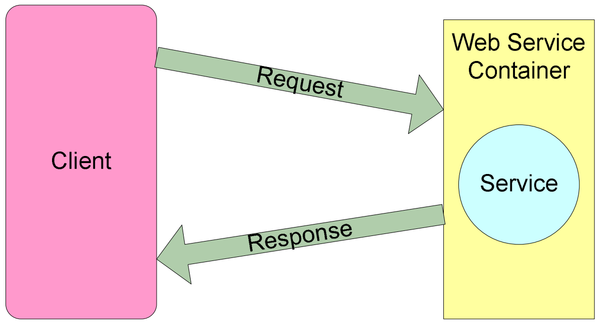

Figure 1. Simple request response paradigm

Axis does not perfectly support one-way messaging. Most Web Services use HTTP as a protocol for message transfers, which naturally provides request and response paradigm. The best way of doing one-way messaging in the current architecture is to send a dummy response message and discard the dummy message upon receipt.

Notification [5] is an example of one-way messaging; an entity may just need to inform another entity about an event. Acknowledgments in the Web Service Reliable Messaging (WSRM) [6] protocol are another example of notification; there obviously no need to acknowledge an acknowledgement.

Messages issued in the direction of client-to-service are a natural way of messaging in Web Services. On the other hand, a message in the opposite direction is very hard to accomplish. Service endpoints are deployed in addressable nodes. These addresses can be published in a registry. One of the main WS-specifications, UDDI, deals with address registry issues of service endpoints. Nevertheless, it is not possible to create a similar architecture to keep the client addresses. Hence, there is no mechanism that provides an environment to send a subsequent (possibly after a certain amount of time) response message back to a client. In order to successfully accomplish this task, Web Service architecture needs addressable clients which are kept in a registry. However, Axis does not keep any client addresses. Therefore, the server side components, handlers or endpoints, are not able to send a message initiated by them back to the clients.

Axis naturally supports synchronous communications. Both the client and the service have to be available during the interaction. A Client also needs to wait for a response after requesting a service. The client communication is thus based on blocking I/O.

There exist many scenarios where there is a clear need for asynchronous communications [7]. Client issues a request to a remote service and then continues its processing without waiting (or blocking) for a response. The service part lets the client know when the result is ready. For example, in reliable messaging, an acknowledgment can be sent back by bundling several of them together. The acknowledgment interval is specified so that the client is notified with a set of acknowledgments instead of sending each of them individually. This helps to increase network performance by decreasing the number of acknowledgments in transit between the endpoints.

We also need to come up with a solution in the current Axis container for our Service deployments, which requires one way and asynchronous messaging. The solution that we utilized is the one used commonly. Hence, it is important to show the impact of this approach.

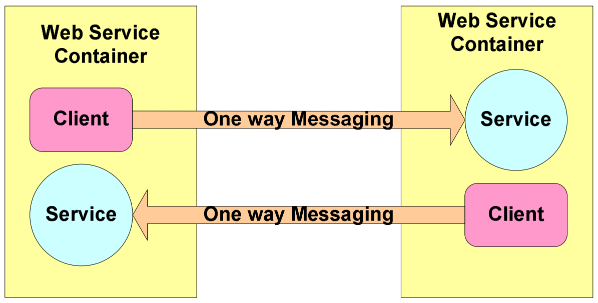

Figure 2. Making asynchronous messaging

The utilized approach is to bypass the aforementioned limitations by using a client where a message initiation is required. Since a client can not call another client, a new service endpoint needs to be created on the client side. We called these two nodes as sink and source. The source represents the client-side while sink represents the server-side. Both the sink and the source are deployed within the Axis/Tomcat container [8].

We gathered results from two test environments. We utilize two types of service request in these environments. The first type, which we call RPC, is a regular Remote Procedure Call (RPC) [9] of Axis. The second type utilizes one-way messaging and achieves the same result as the RPC call of the first type. One-way style messaging is utilized in both the directions viz. client-to-service and service-to-client. To achieve this messaging style, we also deploy the source within a container (see figure 2). The Source, at the client side, sends a message to the sink, at the service side, by using one way messaging. The service in the sink processes the message and passes the response to the client of the sink side and finally the client in the sink calls the service in the source. This mechanism basically provides asynchronous messaging. We are able to request a service from sink while the source can continue its processing of other tasks. The source is notified by another service call from the sink when the response is ready.

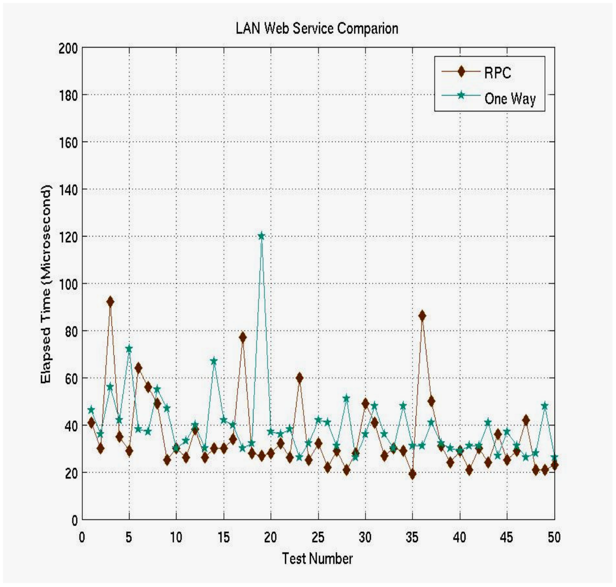

Figure 3. LAN Web Service

The first test is performed between two machines with the Indiana University, Local Area Network (LAN). One of the machines has Pentium 4 CPU operating at 2.80GHz with 1 GB memory. It utilizes Fedora 4 Linux operating system. The other machine, Sun Fire V880, has a Solaris 9 operating system which is equipped with 8 UltraSPARC III processors operating at 900 MHz with 16 GB Memory.

Table 1. LAN Web Service results

Mean	Mean	Standard Deviation	Standard Error
RPC	34.9216	16.2282	2.2724
One Way	39.4200	15.3691	2.1735

The second test is performed between Indiana University and University of Southern California, Wide Area Network (WAN). The first machine has a Pentium 4 CPU operating at 2.80GHz with 1 GB memory. It utilizes the Fedora 4 Linux operating system. The second machine has two Pentium III processors operating at 731.07 MHz with 512 MB of memory. It utilizes Red Hat Enterprise Linux as its operating system.

Although RPC style Web Services can be improved [10], the results of Local Area Network (see Table 1 and Figure 3) show that one way style messaging costs more than RPC style messaging. We get similar results in the Wide Area Network (see Table 2 and Figure 4). The overhead comes from a new service call initiated in the sink which is not the case for RPC. In RPC, the only service call happens in the source part, and the sink responds as soon as it processes the request. On the other hand, in one way messaging, we need to create a new service call in the sink side in addition to the one in the source. Moreover, in RPC style, we may utilize only one container whereas we have to use two of them in one way style. Another factor that needs to be taken into account is thread scheduling; this can causes spikes, as is shown in the figure 3 and 4. These spikes are higher in the one-way style messaging because of using a container in both the sink and the source side. However, the main contributor to the performance overheads is the creation of another call in the sink side.

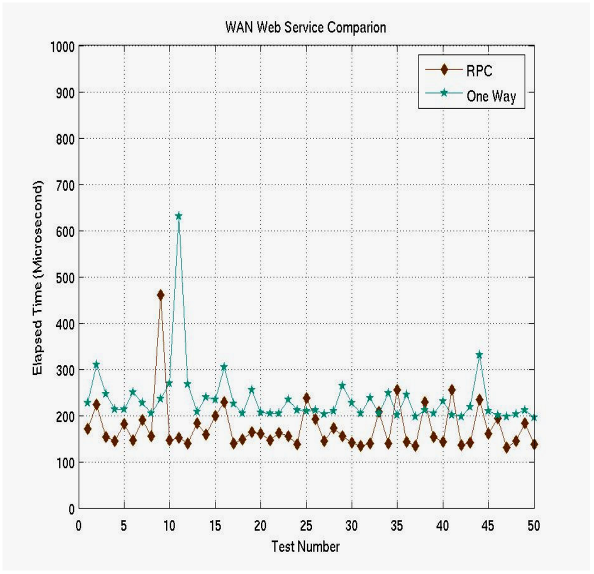

Figure 4. WAN Web Service

In spite of the fact that we had to utilize a bypass solution it seems that it is not efficient enough. A Web Service container should support fully asynchronous and one-way messaging without compromising too much on performance. This capability can be provided to a container by letting the message injection ability utilized in the service-side internally (see Figure 5). Additionally, performance improvement strategies can be applied for better throughput [11].

Table 2. WAN Web Service results

Mean	Mean	Standard Deviation	Standard Error
RPC	173.7400	53.7359	7.5994
One way	234	64.7274	9.1538

There are several scenarios that indicate the need for message initiation in the server side. Having reliable messaging between two endpoints requires several control message exchanges such as establishing sequences and acknowledgments. Although the server can send a response back as soon as it gets a request, it can not send the same massage more than once when it is required. A message may need to be retransmitted if the client has not received it (this is the case in WS-ReliableMessaging [12]).

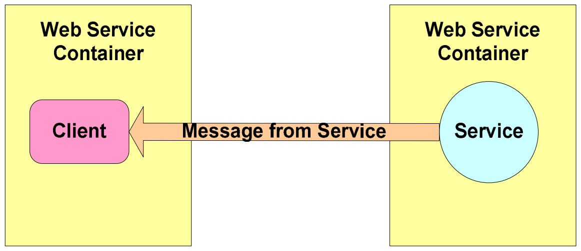

Figure 5. Message can be initiated by service

WS-Notification [13] and WS-Eventing [14] are the other examples where there is a need for message initiation in the service side. Here, a message may need to be sent to multiple end points that might have subscribed to the message. These specifications provide a solution for publish/subscribe mechanism [15] in Web Services. Since every subscriber which is interested in a topic must get the published messages, a new connection for each subscriber must be initiated at the server side

3. Terminating Message Propagation in Handler Chain

In our implementation, we wanted to be able to stop message propagation in order to eliminate unnecessary executions while the message is passing through handler chain (see Figure 6). A good example of this necessity is acknowledgment in reliable messaging. Only the reliable messaging handler needs to know whether the other endpoint has received the message. After getting an acknowledgement in the reliable messaging handler, there is no need to pass it to the endpoint because it is not a message that an end point needs to know about. This would be a very crucial performance issue if the endpoint gets a huge amount of acknowledgements. The reason for sending acknowledgment is to say that “I got the message you sent”. If acknowledgement has not been received from receiver, the retransmission process should be reinitiated. The important thing is that this is the job of reliable messaging handler not the service.

In Axis, the current architecture does not allow us to stop message propagation gracefully. An exception was thrown whenever we attempted to stop the message in a handler. Moreover, this exception propagates back through handler structure and back to the client. This contributes to network overheads. Another problem is that the completed tasks are rolled back if an exception is thrown during message propagation. Therefore, we wanted to access a mechanism that stops the message propagation and does not to cause any extra activities. We come up with the following solution.

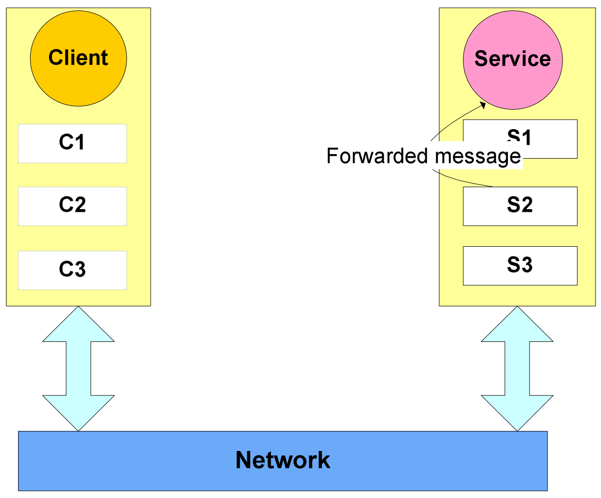

Figure 6. Stopping the message propagation

A message can be forwarded to a dummy task instead of blocking a message. We choose forwarding because we wanted to stop a propagation of a message without getting an exception. If we disposed the message in the blocker handler, we would get exception because of the reasons we mentioned earlier. Letting the message reach a dummy task in the service endpoint prevents this exception. On the other hand, there is a downside in this solution. Processing the dummy message can add to a message’s processing. However, within the Axis architecture we found this cost to be acceptable.

4. The Flexibility of Handler Structure

A handler is an additional functionality to the service endpoints. They can be cascaded to constitute a handler chain. The processing order within this chain is important. It can be either static or dynamic. Currently, containers utilize the static approach. However, a dynamic approach is much more powerful. The corresponding benefits will be clarified here.

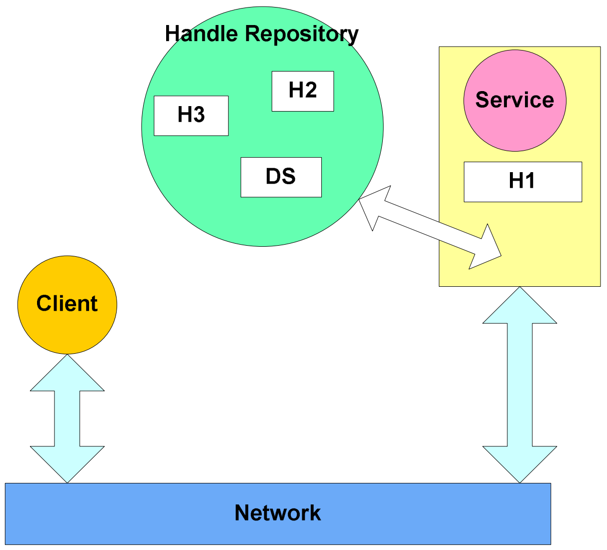

Figure 7. Flexible handler

As we mentioned, the Axis handler chain is currently static. The chain is setup when a service is being deployed. A static structure is generally easy to implement, but harder to customize. A new handler can not be added, just as an old one cannot be removed from the chain after deployment. The Axis architecture only allows for cloning the handler chain. The cloned chain can replace the running one. This is the way adding or removing a handler from chain in the Axis handler architecture.

Handler chains should be customizable on the fly. A Web Service needs to have the ability to select its handlers from the pool of handlers (see Figure 7). For instance, we have a service that sometimes receives signed messages and the verification of the signature is done by handler DS. If we would have the capability to insert DS to the current handler chain for only signed messages, the deployment would be much easier. On the other hand, a handler may need to be removed for specific messages. For example, we have a Web Service with two handlers, H1 and H2. The task of handler H1 is to increment a variable by 1 in SOAP message. The result would be inconsistent if we retransmitted a message by applying handler H1 second time. To prevent this inconsistency, we need to remove handler H1 from the chain that processes the retransmitted message. The second transition must have only handler H2.

5. Conclusion

Web Services are a promising technology to implement scalable [16] and interoperable distributed systems. Its potency is primarily a result of using XML-based messaging. Currently there is a lot of effort related to Web Service standardizations. While the standardization continues, containers, which are the hosts for services, have matured significantly to provide a better SOAP processing environment. Among the Web Service containers, Apache Axis is the most popular one. It is not only the most dominant but also provides the base for other containers. Moreover, efforts within Axis provide guidance to the Web Service community. This work identifies issues that will further contribute to its maturity. These issues are related to messaging, stopping propagation of messages and the flexibility of the handler architecture. We suggest that containers should support one way and/or asynchronous messaging. In addition, there should be a mechanism to ensure that a message can be gracefully stopped while traversing though the handler chain. Moreover, handler chain should employ the handlers dynamically. A flexible and dynamic handler structure will provide many advantages to Web Services. By considering these suggestions, the Axis Web Service container will provide an even better environment for service deployments.

6. References

[1] M. Gudgin, et al, "SOAP Version 1.2 Part 1: Messaging Framework," June 2003. http://www.w3.org/TR/ 2003/REC-soap12-part1-20030624/

[2] Web Services Description Language (WSDL) 1.1 http://www.w3.org/TR/wsdl

[3] Bellwood, T., Clement, L., and von Riegen, C. (eds) (2003), UDDI Version 3.0.1: UDDI Spec Technical Committee Specification., available from http://uddi.org/pubs/uddi-v3.0.1-20031014.htm.

[4] Apache Axis. http://ws.apache.org/axis.

[5] Shrideep Pallickara, Geoffrey Fox. "An Analysis of Notification Related Specifications for Web/Grid Applications," *itcc*, pp. 762-763, International Conference on Information Technology: Coding and Computing (ITCC'05) - Volume II, 2005.

[6] Web Services Reliable Messaging ftp://www6.software.ibm.com/software/developer/library/ws-reliablemessaging200502.pdf.

[7] Marco Brambilla, Stefano Ceri, Mario Passamani, Alberto Riccio. "Managing Asynchronous Web Services Interactions," *icws*, p. 80, IEEE International Conference on Web Services (ICWS'04), 2004.

[8] Apache Tomcat. http://jakarta.apache.org/tomcat/

[9] Andrew D. Birrell , Bruce Jay Nelson, “Implementing remote procedure calls”, ACM Transactions on Computer Systems (TOCS), v.2 n.1, February 1984 , p.39-59

[10] Satoshi Shirasuna, Hidemoto Nakada, Satoshi Shirasuna, Satoshi Sekiguchi. "Evaluating Web Services Based Implementations of GridRPC," *hpdc*, p. 237, 11th IEEE International Symposium on High Performance Distributed Computing (HPDC-11 '02), 2002.

[11] Shengru Tu, Maik Flanagin, Ying Wu, Mahdi Abdelguerfi, Eric Normand, Venkata Mahadevan, Jay Ratcliff, Kevin Shaw. "Design Strategies to Improve Performance of GIS Web Services," *itcc*, p. 444, International Conference on Information Technology: Coding and Computing (ITCC'04) Volume 2, 2004.

[12] Shrideep Pallickara, Geoffrey Fox, Beytullah Yildiz, Sangmi Lee Pallickara, Sima Patel and Damodar Yemme. “On the Costs for Reliable Messaging in Web/Grid Service Environments.” To appear in Proceedings of the 2005 IEEE International Conference on e-Science & Grid Computing. Melbourne, Australia.

[13] Web Services Notification (WS-Notification). IBM, Globus, Akamai et al. http://www-106.ibm.com/developerworks/library/specification/ws-notification/

[14] Web Services Eventing. Microsoft, IBM & BEA. http://ftpna2.bea.com/pub/downloads/WS-Eventing.pdf

[15] Patrick Th. Eugster , Pascal A. Felber , Rachid Guerraoui , Anne-Marie Kermarrec, “The many faces of publish/subscribe”, ACM Computing Surveys (CSUR), v.35 n.2, June 2003 , p.114-131

[16] Ken Birman. "Can Web Services Scale Up?" *Computer*, vol. 38, no. 10, pp. 107-110, October, 2005.