

L. P. H. S. Perera Undergraduate, Department of Computer Science Engineering, University of Moratuwa, Sri Lanka (hemapani@opensource.lk)

C. K. Herath Undergraduate, Department of Computer Science Engineering, University of Moratuwa, Sri Lanka (chathurah@yahoo.com)

D. Weeratunge Undergraduate, Department of Computer Science Engineering, University of Moratuwa, Sri Lanka

B. D. R. Priyanga Undergraduate, Department of Computer Science Engineering, University of Moratuwa, Sri Lanka

Contact Person

Chathura K. Herath

Tel: 0776-592914, -011-2414164

Address: 131/B, Welivita, Kaduwela, Sri Lanka.

email: chathurah@yahoo.com

Enterprise Web Services

Srinath Perera, Chathura Herath, Dasarath Weeratunge, Rajith Priyanga

Abstract

Web Services and the J2EE are two basic realizations of the Distributed Services. The two specifications, J2EE 1.4 Specification [1] and J2EE web Service [2] specifications integrate the Web Services in to the Java 2 Enterprise Edition. The EWS (Enterprise Web Service) is an effort to clinch this integration together with the security and the transaction aspects. This paper discusses the design experience of the EWS project.

Introduction

EWS is a software mainly intended for the integration of the web services to the J2EE platform based on the provisions laid down by the two specifications J2EE 1.4 [1] and the J2EE web Service specifications [2]. But these guide lines does not define how would the critical 'add on services' like Security and Transaction would behave when web services are integrated with J2EE platform. For web service integration for the J2EE platform to become usable at enterprise level, these add on services would be indispensable. Thus while supporting the integration of web services to the J2EE platform, EWS goes on to define and implement the Security and Transactions for the integrated J2EE web services. On this basis EWS can be broken down to three parts the J2EE Web Services [2] implementation, the EWS Security implementation and the EWS transaction implementation.

The realization of the J2EE Web Services[2] provides a the J2EE web Services an programming model which is similar to the J2EE component deployment programming model, based on the Web service module defined by the J2EE Web Services Specification [2] . For an instance with this programming model the Web Service Modules are portable among the J2EE containers [1] and once this module is created there is a deploy tool provide by the J2EE container that automates the rest of the

deployment process. The implementation of the J2EE Web Services Specification [2] is essentially implementing above deployment tool together with the mechanism to invoke the J2EE component implantation of the Web Service.

The security requirements of the J2EE Web Services Specification [2] are far-off from providing a secure infrastructure for enterprise level web services; on the other hand the absence of security can lead to breach of basic concepts. Considering an enterprise java bean deployed in an application server with usual J2EE declarative security and if the security support is not available for the enterprise web service then the developer has to configure the System to invoke the EJB from the web service disregarding the declarative security provided under J2EE security model or invoke only the EJBs methods that doesn't require authentication. The former violates the security requirements specified in the J2EE specification [1] and the EJB specification [7], while the latter will limit the availability and will again constrain the interoperability that was anticipated by enterprise web service. The EWS Security Module tries to provide the end to end security infrastructure for Enterprise Web Services, knowing that it requires authenticating a client through two different security domains; J2EE and Web Service domains.

The Requirements of the EWS part of transaction management is the ability to propagate a transaction from either web services or J2EE domain to the other if the need arises to do so. This could take place if a J2EE component calls a web service or a web service invokes a J2EE web service whose implementation is a J2EE component. Note that as far as J2EE web services are concerned it is adequate to consider how a transaction in the web services domain could be imported into J2EE. Yet for a complete integration of the two domains, it is essential that the reverse process should be addressed as well. Unfortunately, as at present, the release 1.1 of the J2EE web services specification [2] does not address either one of above requirements. Existing technologies, such as OTS [5], JTS [6] etc., are capable of propagating transactions among application components within a

particular domain; i.e. CORBA, J2EE. EWS moves this one step further by making it possible to propagate transactions between two heterogeneous application domains: namely J2EE and web services. Furthermore, the entire process is implicit and transparent to individual components in the same way as other contemporary intra-domain distributed transaction processing technologies.

This paper explains the design experience and the technology used in the EWS project that integrates the Web Services in to the J2EE platform. The paper starts by introducing the related tech technologies associated with the development of EWS. Then it explains the Architecture based on four sub areas; the invocation of a web Service inside a J2EE container, the deployment process of the Web Service to the J2EE container ,the implementation of the Security and the implementation of the transactions. Next the paper explains about the implementation specific details about the EWS. Finally the paper will conclude by briefly discussing the outcomes and new trends introduced by the project.

Background

The background briefly explains the technologies, Web Services, J2EE, Ws-Security, J2EE security, WS-Transaction and J2EE transaction that are necessary to understand this paper.

Web Services

Web Service is a distributed computing technology based on Service Oriented Architecture. It provides distributed services using XML based message format using Web protocols like HTTP/SMTP while SOAP [3] defines the most commonly used message format. The WSDL [4] defines how to define a Web Service independently of the programming language and the JAX-RPC [8] defines the programming model for implementing Web Services in the java programming language. There are set of Web Service specifications that defines 'add on services'. For instance WS-Security for Security and WS-AT for transactions inside the Web Service stack.

Development of a Web Service is done with the help of utility called SOAP engine. A

developer should start with a WSDL file that defines a web Service. The Developer may write it or may generate it from a Java Class or similar representation in other Programming language using a tool if one exists. Using a WSDL file the developer can generate a both the server side code that can be deployed in to a SOAP engine and the client side code that can be used to invoke the Web Service.

The generated client side code using the Client side SOAP engine converts the information about the method invocation in to XML according to the SOAP specification and sends it to the SOAP Engine that act as the Server. The XML message is send through one of the Web Protocols HTTP, SMTP, etc. The Server will convert the XML back in to the programming language representations and invoke the real implementation of the Web Service which is provided at the deploy time. What ever the result is converted back in to the XML and sent back to the Client. The Client converts the XML back in to the programming language representations and return it to the application.

Add on Services for the Web Services are implemented via an extensible framework called Handlers. Those Handlers can be created and configured at the deployment time of the Web Service. They can be invoked before or after the invocation of the Web Services both on the client and server ends.

J2EE is a platform for enterprise java applications define by the sun Microsystems for enterprise applications. J2EE applications are of two fold, the Web based applications and Enterprise Java Beans (EJB). EJBs provide the similar remote services as Web Services yet information about the invocation is sent via a binary protocol and is Java specific. The J2EE defines a J2EE container in which the J2EE applications reside. This Container provides services like deployment support based on the standard package format, the security support, the transaction support, the connector Architecture so on.

The J2EE has a well define programming model and J2EE developers may create J2EE applications by creating few java classes and the configuration files called Deployment Descriptors. All those should packaged in to specific modules (JAR, WAR, and EAR) and the J2EE container will accepts the module and deploy the J2EE application inside the container. This programming model for developing the J2EE application is named

J2EE Component programming Model in this paper.

The packaged J2EE application module is portable across the different implementations of the J2EE containers. Being the two technologies that are parallel in the distributed computing stack the integration of the two technologies allows both the technologies to benefit from the strong points of the other. The J2EE platform benefits from the wider availability, interoperability among different platforms and the programming languages. The Web Services benefit by adopting the mature add on services and the programming model of the J2EE platform.

Security

The importance of the security implementation has been duly addressed previously, but achieving it is made difficult due to the fact that enterprise web services involve two domains (i.e. Web Services and J2EE) and these two domains adopt their own security models. J2EE [1] provides a very comprehensive security infrastructure and the Web Service Security [9] has now been standardized and taken a form of its own.

WS-Security aims at enabling applications to construct secure SOAP message exchanges. [3] This specification provides a flexible set of mechanisms that can be used to construct a range of security protocols such as PKI, Kerberos, and SSL. WS-Security specification provides three security mechanisms; security token propagation, message integrity, and message confidentiality. These mechanisms by themselves do not sum up to a secure system. Instead they are the building blocks that can be used in conjunction with other Web service extensions and higher-level application-specific protocols to accommodate a wide variety of security services. Therefore it is the responsibility of the high level protocol to achieve the required level of security using the building blocks provided by WS-Security. WS-Security provides support for multiple security tokens, multiple trust domains, multiple signature formats, and multiple encryption technologies.

The J2EE security model is primarily based on the concepts of security roles and declarative security. The J2EE server allows the J2EE components to be secured under this

fashion and it is further explained in the J2EE specification. Such implementation provides tremendous flexibility, for example, foo() method in an EJB is accessible to only to the users in the right user role. During this work the server side security was achieved using this J2EE security infrastructure.

In the J2EE server before any method call on any component the server will ensure that the client doing the method call has duly authenticated itself to the J2EE server. If this client initially produced its credentials for login then the J2EE server will try to authenticate the client and on success it will associate an authenticated principal with that thread of execution so during further method calls the J2EE server will allow the client actions appropriately.

It was pointed out in the introduction that it is necessary to get client credentials using the WS-Security implementation and the client should be authenticated to the J2EE server. For this purpose the Java Authentication and Authorization Service is used. It implements a Java version of the standard Pluggable Authentication Module (PAM) framework and compatibly extends the Java 2 Platform's access control architecture to support user-based authorization.

Transaction

J2EE has a flat transaction model [7]; a flat transaction cannot have any child (nested) transactions. The API used for transaction management in J2EE is the Java Transaction API (JTA) [10]. A Bean Provider [7] can choose between using programmatic transaction demarcation in the enterprise bean code (bean-managed transaction demarcation) or declarative transaction demarcation performed automatically by the EJB container (container-managed transaction demarcation). With bean-managed transaction demarcation, enterprise bean code uses JTA to demarcate transactions. With container-managed transaction demarcation, the container demarcates transactions per instructions provided by the Application Assembler in the deployment descriptor.

When one application component invokes another component, transactional state or the transaction context of the calling thread is propagated to the called component along the invocation. Transaction context propagation in

J2EE is handled by the Java Transaction Service (JTS) [6]. JTS is a Java mapping of the CORBA Object Transaction Service (OTS).

Web services model transactions as a type of activity. An activity is a coordinated set of actions performed by a group of web services. Activities are coordinated by coordination services and are propagated among participant services by means of a coordination context. WS-Coordination [11] defines how activities may be created, propagated and coordinated to reach an outcome agreed upon by all participants. Creation and propagation of activities is independent of the type of activity. However, the means of reaching an agreement on the outcome of an activity is dependent on its type. For each type of activity, another set of specifications define the protocols that may be used in the process of instance termination and reaching collective agreement. For example, WS-Atomic Transaction stipulates how agreement is to be reached in the case of atomic transactions [12]. Apart from the traditional atomic transactions with ACID properties, web services use another more relaxed form of transactions. These are called Business Activities. The coordination of business activities is described under WS-Business Activity [13]. The flat transactions used in J2EE maps closely onto atomic transactions used in web services. The concept of business activities found in web services however, has no J2EE equivalent.

Architecture

The Architecture of the EWS has four considerations to address. They are how the invocation of the Web Service inside J2EE is handled, the deployment mechanism of a Web Service inside a J2EE platform and how the security and the transaction are handled.

The scope of the J2EE Web Service specification [2] is broad. However the most of the components that are required to realize the specification already exists. For an instance AXIS; a SOAP Engine, WSDL2Java and Java2WSDL, the code generation tools for SOAP engines already exist. The implementation of the EWS is assembling those existing components to execute the web service invocation and the Web Service Deployment processes, and implement the security and transaction. Web Service side of the EWS is

implemented based on the Apache Axis, the Apache implementation of the SOAP Engine. But the J2EE side is independent of a specific J2EE Container. The EWS implementation is optimized for the Geronimo but the EWS supports other EJB containers (e.g. JonAs and JBoss application servers).

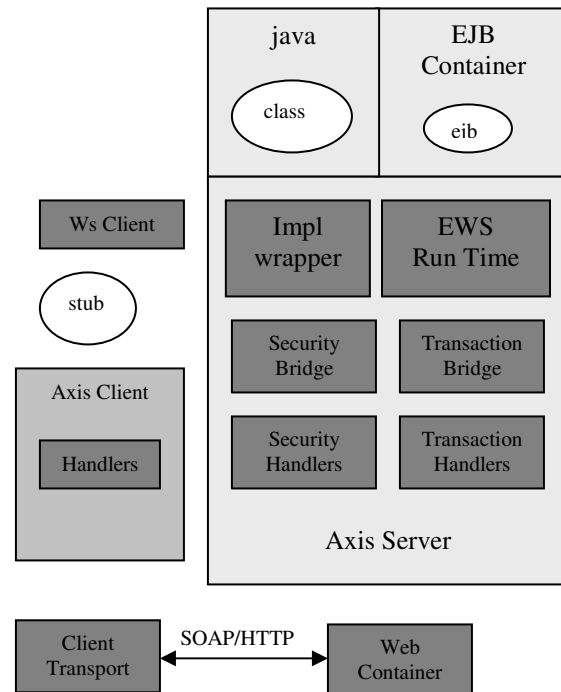


Figure: EWS Architecture.

J2EE Web Service Invocation

The First part of the EWS architecture is about what happens when the Web Service client invokes a Web service. The Web Service Client generates SOAP [3] message corresponds to the invocation and sends it via HTTP to the Web Container. Then the HTTP transport layer that provides by Axis accepts the SOAP Message over HTTP, parse the HTTP Headers and invoke the Axis Engine parsing the XML SOAP Message to the Axis Engine. The Axis Engine will find the configured transaction or security Headers and process them. The necessary information is converted in to the J2EE Domain and propagated in to the J2EE container.

The EWS has a java Class that is invoke by the Axis and this Class knows how to find the implementation EJB/POJO associated with the

Web Service. This Class is called the invocation code from this point onward. This class performs the invocation of POJO using the usual java invocation and the invocation of EJB uses Remote Interface/Local interface or the J2EE container internals to invoke the Web Service implementation as decided by the user.

Once the transaction and the security information is propagated in to the J2EE container Axis converts the SOAP request parameters to the java Objects and invokes the invocation code which would find the implementation EJB or the POJO that provides the Web Service implementation and invoke the implementation. Axis engine will convert any result back to the SOAP and send them back to the Client.

The Web Service deployment

The Second part of the EWS is the deployment process. The EWS project has a tool called WS4J2ee that accepted the standard Web Service Module explained by the J2EE Web Service specification and generates the artifacts that can be deployed in Axis and a J2EE Container. In effect those artifacts contain java invocation code for Axis and EJB/POJO that provides the Web Service implementation and the WSDL file. If the Web Service Deployment Descriptors has the security or the transaction services enabled, the handlers for those services are provided by the EWS project and the WS4J2ee will generate the code to configure the Axis to provide those Services.

The rest of the deployment is the J2EE container specific. We would be stuck in to what happens in the Apache Geronimo J2EE container. Deployment request for the J2EE Web Service is accepted by the J2EE container deployment module and the Deployment module generates the above explained artifacts with the deployment tool ws4j2ee using the packaged Web Service module as Input. The deployment tool will deploy the invocation code in Axis and the EJB inside the EJB Container if there is an EJB as implementation bean. From this point onwards the Web Service is available via SOAP/HTTP from the J2EE container. The WSDL associated with the Web Service is published in a JAXR registry for enable the Clients to Query and find the Web services.

EWS Security

The third part of the EWS architecture is the Security. At a very abstract point of view, an Enterprise Web Service invocation can be viewed as a remote method call that is initiated at the web service domain as a SOAP request and converted in to the EJB call in the J2EE domain. The Response will be started as an EJB call and ended as a web service SOAP response. Thus path of this information exchange is circular where half of it is in Web Service domain and the other half is in the J2EE domain.

The J2EE component, in this case an enterprise java bean, will reside in the J2EE server and it is exposed as a web service to the potential client. So the intuitive solution will be to get the credentials from the web service client and somehow transport them securely on top of SOAP/HTTP so that the Web Server can produce those credentials to the J2EE server and authenticate the web service client to the J2EE server. Since the SOAP/HTTP is purely web services, it is appropriate to use the WS Security implementation to securely deliver the credentials.

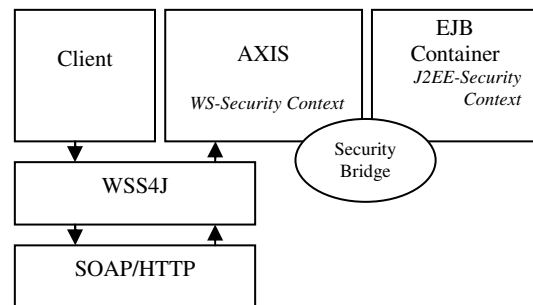


Figure: EWS Security Architecture

So when the web service client invoke the web service eventually, the EWS runtime will invoke the relevant EJB method and by the time the web service client has already authenticated itself to the J2EE server. As a result the invocation should proceed seamlessly unless of course the authentication fails or the authenticated principle doesn't possess sufficient clearance to call the method.

In abstract the two main subsections (i.e. the web services subsection and the J2EE subsection) have their own independent security contexts, WS-Security context and the J2EE security Context. The essence of the EWS Security is to use one security context to create another or to be specific to use the web service

security context to create a J2EE security context which is achieved by means of a bridge between two domains.

The Four basic security operations, the Authentication, Authorization, encryption and signing are supported by the EWS security Module.

The Authentication is implemented using the Web Service security implementation that propagates the Web Service Security context and a security bridge which convert the Propagated security context in to a J2EE security context.

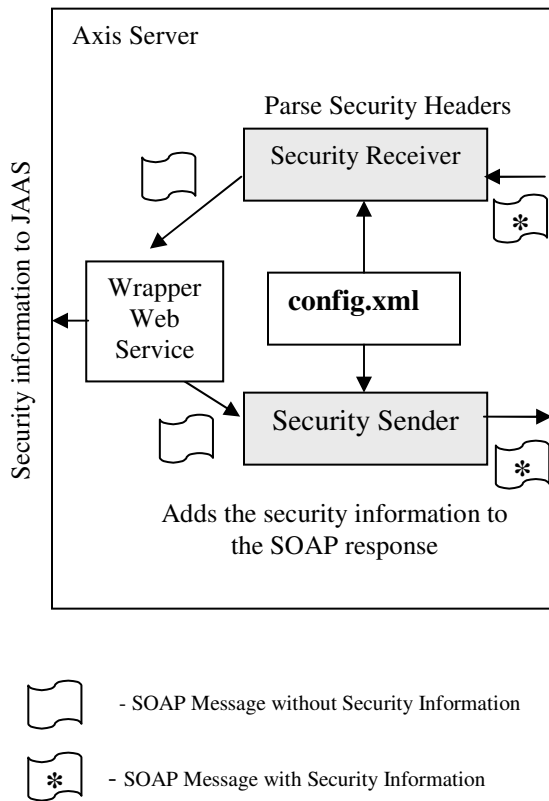


Figure: WS- Security work flow

The Authorization is realized using the EJB container which is responsible for enforcing access control on the enterprise bean method [7] by consulting the security policy (derived from the deployment descriptor) associated with the enterprise bean to determine the security roles that are permitted access to the method. For each role, the EJB container uses the security context associated with the call to determine whether the caller can be mapped to the role. If the mapping is successful, the call is authorized by the container to dispatches the control to the enterprise java bean method.

The confidentiality and the integrity are handled by the WS-Security layer and the communication between the Web Service and J2EE domains is considered secure.

During the design and In order to provide a credible security infrastructure to J2EE Web Services, EWS Security identified certain design considerations. First, the security provision should be loosely coupled with the service implementations, so that the service deplorer should be able to change their security policies by modifying the configuration files but not the service implementation classes. Second, the implementation of the security module should have a high degree of adaptability to new requirements and technologies since this is an evolving area of technology. And the last the implementation of security module should use the existing technologies and existing programming models as far as possible.

EWS Transactions

When one component invokes another, it is necessary to propagate transactional state of the first to the second. The transactional state must be exported from the caller's end; the states that exported should be imported from the called end.

J2EE transaction model is layered in two specifications JTA [10] and JTS [6]. Hence, the J2EE Web service [2] runtime may opt to interact with the J2EE transaction services at either on of those levels. The architecture described below maintains a one to one correspondence between activities and local transactions. If a transaction is imported into J2EE and is subsequently exported again, the exported transaction carries exactly the same coordination context as the imported transaction. Similarly when an exported transaction is imported subsequently, the same local transaction is reused. Further, when the same activity or a local transaction is imported or exported multiple times, it is associated with the same local transaction or activity on all occasions.

Importing a transaction

In the context of J2EE web services it is necessary to import an external transaction when an incoming web service request contains a

coordination context of type atomic transaction [12]. Under such circumstances, the J2EE Web Services J2EE [2] runtime must create and associate a local transaction to the thread that calls the service implementation beans.

JTA does not allow external transactions to be imported into J2EE. However, JCA [14] allows an Enterprise Information System (EIS) [1] to submit work to a J2EE server where it may also specify an external transaction under which work is to be performed. This puts an indirect requirement on J2EE transaction managers (or JTA implementations) to provide an API that allows external transactions to be imported even though JTA does not provide for it. But most open source transaction managers in fact do support such an API however no standard API exists. Though JTS handles transaction propagation, it only handles transaction propagation between J2EE domains and assumes that components communicate over Java RMI. In the context of J2EE web services, it is required to import transactions from the web services domain into J2EE where different protocols are used on each domain for transaction management.

The architecture that follows involves the following key components: an axis handler, a bridge, a transaction importer, a transaction terminator and the J2EE transaction manager. The transaction importer is a transaction manager specific component that allows transactions to be imported into J2EE. The transaction terminator encapsulates the JCA _ interface.

On the request flow, on the server side, the axis handler looks among the message headers for a coordination context of type atomic transaction. If it finds one, it calls the bridge to import the activity into J2EE. If the activity is already imported the bridge used it or else the transaction importer is called to create a new local transaction. The created new transaction is given a new Xid [15] and a timeout based on the Expires attribute [11] of the coordination context [11]. The bridge also registers for participation in the activity under Durable two phase commit (2PC) [12] protocol and the registration service endpoint is obtained from the coordination context [11].

The bridge maintains a correspondence between all imported activities and corresponding local transactions. This correspondence is used each time a new activity is imported to check whether it has already been imported. If the activity is already imported it is

not necessary to create a new local transaction. The previously created local transaction is reused instead. Hence, at any point there is only one local transaction corresponding to any one activity. In either case, the bridge returns a local transaction to the axis handler, which attaches it to the current thread by calling the transaction manager. This transaction is detached from the thread, when the axis handler is called again during the response flow.

When an imported activity is terminated, the coordination service carries out 2PC protocol and the bridge is treated as a durable participant. In responding to messages conveyed to it during this process, the bridge uses reference properties and its internal correspondence between imported activities and local transactions to locate the local transaction that corresponds to the activity being terminated. It carries out 2PC [14] on the local transaction through the transaction terminator in sync with 2PC being performed on the activity. Consequently, any operation performed against the local transaction, has a direct impact on the outcome of the activity. The outcome of the activity in turn governs the outcome of the local transaction and operations performed against it.

Exporting a transaction

Exporting transactional state is not a novel concept and is taken care of by JTS. However, tight coupling of J2EE with web services requires capabilities beyond JTS since it is necessary to propagate transactional state to web services in addition to J2EE components. The approach described below, handles transaction propagation on web service invocations made from J2EE. Under the scheme, the propagation of transactional state takes place implicitly and transparently to the bean provider.

It involves the following key components: an axis handler, a coordination service [11] and the J2EE transaction manager. During request flow, on the client side, the axis handler checks whether the executing thread has a transaction context through the transaction manager. If the executing thread is associated with a local transaction, the bridge is called to export the local transaction. The bridge first checks whether the local transaction is already associated with an activity. If it finds one the Corresponding coordination context is returned to the axis handler.

This may occur in two scenarios. Firstly, the local transaction may represent an imported activity. On the other hand, the transaction may have been exported earlier. If the local transaction is not associated with an activity, the bridge requests the coordination service to interpose in the local transaction. In response, the coordination service imports the local transaction and creates a new activity to represent it in the web services Domain.

The concept of importing an activity into a coordination service is described in WS-Coordination and has several motivations. For example, it allows an application to use an intermediary set of coordination services that it prefers to deal with directly due to performance, trust domain, or control reasons. In the context of exporting transactions from J2EE, the local J2EE transaction manager is unable to coordinate participants in the web services domain. Hence taking a cue from WS-Coordination, an alternate subordinate coordinator is created through interposition for coordinating those participants. However, interposition as described here is different from what is described under WS-Coordination in two ways. Firstly, the coordinator is requested to interpose in a J2EE transaction rather than an existing activity. Secondly, coordinator interposition results in the creation of a new activity.

Interposition results in the coordination service, returning a coordination context to the bridge. The bridge then updates its correspondence between activities and local transactions accordingly and passes the coordination context onto the axis handler to be placed in the soap request.

An exported transaction may be terminated as any other J2EE transaction. However, termination may also be triggered by the termination of the activity. In both cases, it could be easily handled from within the coordination service if the coordination service registers itself as a resource [1] in the local transaction.

Implementation of EWS

The EWS implementation composed of following four sections of code, A Code Generation tool, J2EE container specific deployment module that use the tool, and security and transaction implementations.

Code Generation Tool

The Code Generation tool consists of three types of classes, Contexts, Parsers, and Generators and the implementation classes are named accordingly following the pattern XXContext, XXGenerator and XXparser. The Context is run time representation of the Deployment Descriptor or a module. The parser will parse the actual representation of the input module and populate the Context associated with the module which would be used by the corresponding generators to generate the Deployment Descriptor, Java classes act.

The code Generation is done by Set of Generators, each having one or more Writers. A Generator represents a code module that takes care of the generating a module, for instance a module to create an EJB and each writer will generate a single file. All the code interacts with one Mediator and it is possible to remove or add new Code Generation Modules. Each object is representing by the interfaces and their actual representations are hidden behind the Factory classes. There is a Factory that Generate those factories and it is accepts in to the Ws4J2ee class (tools main class) as a parameter. Using that the behavior of the tool can be altered programmatically without changing the code of the original tool.

Deployment Module

The implementation of the deployment process is J2EE container specific. The EWS supports only the Geronimo deployment. But the other J2EE container users should follow the manual deployment.

Security

The realization of the EWS Security needs two things as explained by the architecture, the Web Service Security implementation and a security Bridge.

The Web Service Security implementation is based on the WSS4J project [16], an effort to implements WS-Security specification, which is currently being developed as an Apache WS-FX project named WSS4J. It has introduced a programming model based on axis handlers, which provides a set of server side request chain handlers that performs both the

SOAP message security handling and preparation of the security information explained in the architecture. One or more combinations of these handlers can be used to provide HTTP level authentication or any SOAP based security services defined in WS-Security specification [9]. The handlers will create and store the appropriate callback object that would be later used by the JASS to get the credentials using the deployment descriptors and the available information from the SOAP security headers.

The implementation of Security bridge was achieved using JAAS, the Java Authentication and Authorization Service [18] [19] using the called back object that is created by the WS-Security infrastructure. Nonetheless, configuring the JAAS modules and security domains are container specific and it is out of the scope of the project.

User authentication is the process by which a user proves his or her identity to the system. This authenticated identity is then used to perform authorization decisions for accessing J2EE application components. To add security to a J2EE Web Service, adding one of the above described handlers with the appropriate parameters, to the Deployment descriptor is sufficient. Then when the EWS tool generates the code, it will register the Handlers inside Axis.

Transaction

As explains by the Architecture the EWS transaction requires a transaction importer and a transaction exporter.

The implementation of transaction importer is transaction manager specific. However the rest of the transaction inflow design is portable. The crux of the problem is that JTA does not provide a standard way of importing transactions into a transaction manager. On one hand it allows transactions to be created but provides no way of carrying out 2PC [14] and on the other hand it does not expose Xid s [15] of local transactions so JCA may be used to terminate instances created through JTA via 2PC. Even though JCA allows an EIS to import transactions, JCA only provides necessary interfaces so that the EIS tier may request the application server to do work as part of an external transaction. Thus the present J2EE architecture does not presents standard

mechanism exists for importing an external transaction into J2EE.

On the contrary, the implementation of transaction outflow is entirely portable across different application servers and numerous JTA implementations since it leverages only on JTA. However, the requirement for a coordination service that supports interposition for J2EE transactions is beyond what is provided by WS-Coordination. Thus the design is not portable across coordination services. Nonetheless, since WS-Coordination provides for interposition of activities, if the need arises to use a particular coordination service, it could always be requested to interpose in the activity created by the first coordination service.

One problem that crops up with transaction outflow is how to handle timeouts. WS-Coordination recommends each coordination context carry an optional Expires attribute. This attribute is used by the transaction inflow mechanism to set a timeout when creating local transactions. However, when exporting local transactions, there is no way to find out when it was created and as a result, how much time remains before the instance timeouts. Thus the Expires attribute cannot be initialized to its correct value. A conservative solution is to set the Expires attribute to the maximum timeout possible.

Conclusions

A main goal of the Web Services to J2EE specification is to integrate the Web Services in to the J2EE platform while using the existing technologies as much as possible. The EWS build on top of the Axis the Apache Web Service implementation and the Geronimo, the Apache J2EE container achieved that goal. With the EWS implementations user can deploy the Web Services inside Geronimo in the same way they deploy an EJB following the J2EE development roles provider, assembler, Deployer.

As described in the sections above this work introduces the Security and Transaction implementation to the J2EE web services so that the EWS will become commercially viable infrastructure. The work related to EWS security and EWS transactions has provided the stepping stones for future standardizations in those area and this work can be very much guide the future course of the J2EE Web Services.

References

1. J2EE 1.4⁶ Specification, Sun Microsystems Inc.
2. J2EE Web services Specification, (JSR 109) Sun Microsystems Inc.
3. SOAP 1.2 Specifications
4. WSDL 1.1 Specification
5. Transaction Service Specification, Version 1.4, Object Management Group (OMG)
6. Java Transaction Service (JTS) Specification, Sun Microsystems Inc.
7. Enterprise JavaBeans Specification, Version 2.1, Sun Microsystems Inc.
8. Java API for XML Remote Procedure Calls (JAX-RPC).
9. Web Service Security Specification [Online] Available
10. Java Transaction API (JTA) Specification, Sun Microsystems Inc.
11. Web Services Coordination (WS-Coordination) Specification, September 2003, IBM, Microsoft and BEA System.
12. Web Services Atomic Transaction (WS-Atomic Transaction) Specification, September 2003, IBM, Microsoft and BEA System.
13. Web Services Business Activity (WS-Business Activity) Specification, September 2003, IBM, Microsoft and BEA System
14. Java 2 Enterprise Edition Connector Architecture, v1.5.
15. X/Open CAE Specification ñ Distributed Transaction Processing: The XA Specification, X/Open Document No. XO/CAE/91/300 or ISBN 1 872630 24 3.
16. WSS4J Project documentation. Apache Software Foundation [Online] Available:
17. Servlet 2.3 Specification. (2000), Java Community Press (JCP) [Online] Available:
18. Charlie Lai, Li Gong, Larry Koved, Anthony Nadalin, and Roland Schemers (December1999) User Authentication and Authorization in the Java ô Platform.
19. Java Authentication and Authorization Service (JAAS) 1.0 Developer's Guide [Online Available]