Lessons Learned from Implementing WS-Coordination and WS-AtomicTransaction

Ivan Silva Neto   Francisco Reverbel

Department of Computer Science
University of São Paulo
{ivanneto,reverbel}@ime.usp.br

Abstract

This paper presents the design and implementation of a transaction service that complies with the WS-Coordination and WS-AtomicTransaction standards. Such service builds upon XActor, a distributed transaction manager that supports an open-ended set of transports, and enhances it with full support for atomic transactions over Web services. The paper summarizes the main lessons we learned from implementing those standards. It also identifies weaknesses in the WS-AtomicTransaction specification and presents advice for implementors of any systems based upon WS-Coordination.

Keywords*: Web services, transactions, crash recovery, WS-Coordination, WS-AtomicTransaction.*

1. Introduction

Reliance on Web standards, industry endorsement, and ability to operate in heterogeneous and firewall-protected environments are some of the key reasons for the widespread use of Web services. Nevertheless, such attractive features are not enough to enable the development of enterprise-class applications, which usually do not tolerate data inconsistencies. To suit this class of applications, transactional support is needed [3].

A pair of recent WS standards, WS-Coordination (WS-C) [8] and WS-AtomicTransaction (WS-AT) [6], has added transactional capabilities to the Web services stack. This paper reports our experience designing and implementing a transaction service that complies with those standards. We implemented that service as a plug-in for XActor [11], a distributed transaction manager that affords transactional interactions over an open-ended set of transports. Our plug-in enhances XActor with full support for atomic transactions over Web services, including crash recovery capabilities. It cooperates with XActor to transparently handle all the complex interactions that take place between the parties involved in a distributed transaction.

This paper is organized as follows: Section 2 contains an overview about Web services transactions, Section 3 describes our transaction service implementation, Section 4 reviews related work, and Section 5 presents our concluding remarks.

2. Web services transactions

The WS-C specification defines an extensible coordinator that may be used to coordinate virtually any kind of activity: atomic transactions, long-running transactions, workflow processes, etc. That coordinator, however, is just an abstract part of a coordinator framework. As such, it is not able to coordinate any activity by itself. To do so, the coordinator has to be extended. WS-AT sits atop the WS-C specification and enables the coordinator to handle atomic transactions. The WS-BusinessActivity (WS-BA) [7] standard is another relevant extension to the WS-C coordinator. It enables such coordinator to handle long-running transactions.

2.1. WS-C/WS-AT port types

The WS-C coordinator exposes its functionality via two port types[1]: `ActivationCoordinator` and `RegistrationCoordinator` (Fig. 1). Each of these port types has a single operation. The first one has the operation `CreateCoordinationContext`, which creates a new coordinated activity (e.g., an atomic transaction). The second one has the `Register` operation, used by participants to join a coordinated activity.

WS-AT extends the WS-C coordinator with two other port types, named `CompletionCoordinator` and `Coordinator`. The former has operations `Commit` and `Rollback`, invoked by clients to tell the coordinator whether a transaction should be committed or rolled back. The latter has the operations `Prepared`, `ReadOnly`, `Aborted`, and `Replay`. These operations are invoked by the transaction participants at commit time, when the coordinator is driving the two-phase commit (2PC) [3] protocol.

[1]A port type is a logical grouping of operations. It is, therefore, a concept similar to an interface in Java or C#.

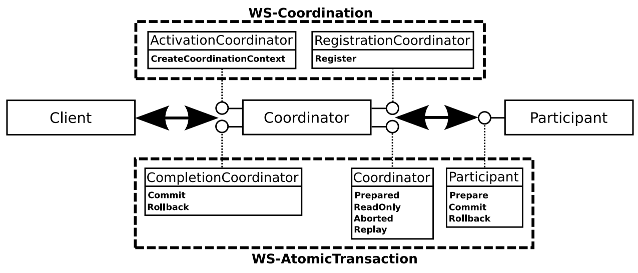


Figure 1. The WS-C/WS-AT port types.

WS-AT also defines a third port type named `Participant`, which has to be implemented by every transaction participant. This port type has the operations `Prepare`, `Commit`, and `Rollback`. All 2PC interactions between the parties involved in a distributed transaction over Web services take place through those port types.

2.2. Web services transactions in Java EE

In the Java EE scenario, the application server (AS) — or, more precisely, its transaction manager (TM) — has to implement the WS-C/WS-AT port types (Fig. 2). The purpose of those port types is to allow the communication, via SOAP messages, between the parties involved in a distributed transaction: (*i*) the client, responsible for the demarcation of the transaction boundaries, (*ii*) the root coordinator, responsible for determining the outcome of the distributed transaction, and (*iii*) the participants, i.e., the application servers that interact with transactional resource managers (RMs) such as database systems and message brokers.

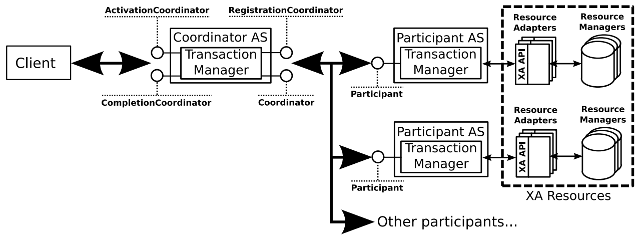


Figure 2. Web services transactions in a Java EE scenario.

The X/Open XA specification [16] is the relevant standard for interactions between a TM and transactional RMs. A TM running within an application server interacts with transactional RMs through resource adapters (JDBC drivers, JMS providers, etc.) that implement a Java mapping of the XA API. The RMs whose resource adapters implement that API are called *XA resources*.

2.3. WS-C/WS-AT transaction context

Besides a set of port types, the WS-C/WS-AT standards also define a transaction context that is injected into the header of every SOAP message sent out within the scope of a transaction. That context contains three fields: a globally unique transaction identifier, the transaction timeout, and a reference to the transaction coordinator. Hence, a Web service that receives a transaction context along with a SOAP request has all the information needed to join the transaction represented by the context.

3. The WS-C/WS-AT transaction service

Our WS-C/WS-AT service is built upon XActor[2], a transaction manager written in Java and aimed at server-side application containers such as Java EE servers and dependency injection frameworks (e.g., Spring [19]). XActor is extensible, in the sense that it can employ an open-ended set of transports to coordinate distributed transactions. A software layer named transactional remote method invocation plug-in, or simply *TRMI plug-in*, enhances the TM with support for a given transport. Before we started this work, XActor had already TRMI plug-ins for transactional interactions over IIOP and JBoss Remoting [4].

We implemented our WS-C/WS-AT service as a third TRMI plug-in, which encapsulates the SOAP/HTTP invocation mechanism. This plug-in is a dynamically deployable feature that enhances XActor with full support for atomic transactions over Web services, including crash recovery capabilities. Summarily, the plug-in comprises: (*i*) Web services that implement the WS-C/WS-AT port types, (*ii*) interceptors to propagate and import the transaction context, (*iii*) a software layer that encapsulates the SOAP/HTTP invocation mechanism and makes it available to XActor through well-known interfaces, and (*iv*) another software layer that extends the crash recovery mechanism of XActor. Each of these items will be discussed in the following sections.

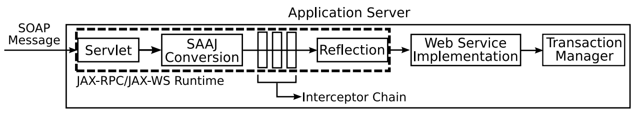
3.1. Implementation of port types

The Web services that implement the WS-C/WS-AT port types perform the role of adapters, converting the SOAP messages they receive into appropriate calls to XActor. For example, when the Web service that implements the `ActivationCoordinator` port type receives a `CreateCoordinationContext` message, it parses such message and then invokes the `begin` method of the TM, which starts a new transaction.

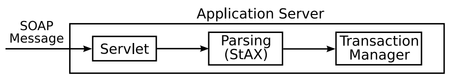
Performance issues. The JAX-RPC [13] and JAX-WS [15] APIs, both part of the Java EE 5 platform, are the most straightforward options for implementing the WS-C/WS-AT port types. However, for the sake of performance, we decided not to employ them. Both JAX-RPC and JAX-WS provide a great deal of flexibility, but such flexibility

[2]Project website: `http://xactor.sourceforge.net/`. All the source code for XActor, including the source for our plug-in, is available as free software at the project website.

comes at the price of performance. Fig. 3a shows the path a SOAP message would have to traverse if we had implemented the WS-C/WS-AT port types using JAX-RPC or JAX-WS Web services. In this approach, a servlet initially receives the SOAP message (which could be, for example, a `Commit` message from a client that wants to commit a distributed transaction). The message is converted into elements of the SOAP with Attachments API (SAAJ)[3] [14] and then traverses an interceptor chain. After that, reflective mechanisms are used to perform an XML-to-Java mapping and to invoke the suitable method of the class that implements the Web service. At last, the Web service implementation calls the TM.



(a) SOAP message processing in a JAX-RPC/JAX-WS runtime.



(b) Our approach for processing SOAP messages.

Figure 3. SOAP message processing.

A more efficient approach. Instead of using JAX-RPC or JAX-WS Web services, we implemented the WS-C/WS-AT port types using servlets that analyze the SOAP messages via an efficient API: the Streaming API for XML (StAX) [2]. A three-step process is used to process the SOAP messages targeted to the WS-C/WS-AT port types: the message is received by a servlet, analyzed by a StAX parser, and then the appropriate method of the TM is invoked (Fig. 3b). Compared to JAX-RPC or JAX-WS Web services, this process avoids the SAAJ conversion (and therefore the use of a DOM parser), the traversal of the interceptor chain, and the employment of reflective mechanisms.

Although Fig. 3b shows that the TM is invoked right after the message parsing, there is one extra level of indirection for the messages targeted to the WS-C port types. Since the coordinator defined in the WS-C standard is extensible, our implementation of this coordinator delegates the received requests to a repository of classes that implement the WS-C operations. Such classes are named coordination types. The repository dynamically selects the appropriate coordination type to handle a given request. The "atomic transaction coordination type" we implemented simply forwards the requests to the TM (Fig. 4). Our WS-C implementation allows the dynamic deployment of new coordinator functionalities. We could, for example, add WS-BA support (through a "business activity coordination type") to a running TM.

[3]A great number of SAAJ API interfaces extend DOM [18] interfaces. Therefore, the use of the SAAJ API requires a DOM-compatible XML parser.

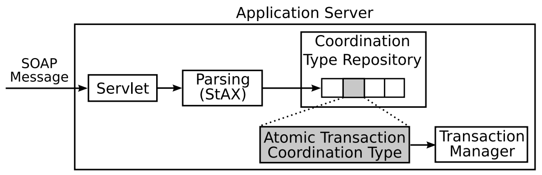


Figure 4. Dynamic selection of a coordination type.

Creation of SOAP messages. Also for performance reasons, instead of using an XML parsing API to generate the SOAP messages, our plug-in employs a template-based approach. The plug-in has a message template, stored as a string, for each message defined in WS-C/WS-AT. Whenever a new message needs to be created, the template is filled out with the message-specific data.

3.2. Propagation of the transaction context

Our plug-in propagates the transaction context implicitly: the context is transparently and automatically injected into all the SOAP messages sent within the scope of a transaction.

Interceptor usage. In Java EE environments, Web services are usually accessed through proxies that the application server binds into a naming service. The plug-in we developed equips the proxies bound into the naming service with an interceptor. This interceptor checks if a SOAP message is being sent within the boundaries of a transaction, and if so, the interceptor injects the transaction context into the outgoing SOAP message (Fig. 5). On the server side, another interceptor is employed. The role of such interceptor is to check if a transaction context comes along with an incoming SOAP message. If so, the interceptor extracts the context and imports it into the TM (Fig. 5).

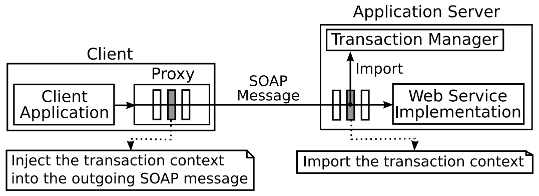

Application Server
Transaction Manager
Import
Client
Proxy
Client Application
SOAP Message
Web Service Implementation
Inject the transaction context into the outgoing SOAP message
Import the transaction context

Figure 5. Propagation of the transaction context.

Automatic registration with the coordinator. When the TM imports a transaction context, it creates a local branch of a distributed transaction. The application server automatically enlists with that branch any XA resources (e.g., databases, message queues) used to fulfill the SOAP request that conveyed the transaction context. If at least one

XA resource is used by the transaction branch, the TM that hosts the branch must register itself as a participant with the transaction coordinator, in order to receive the messages of the 2PC protocol.

A very convenient feature of XActor is that when it receives an enlistment request from the application server, it automatically and transparently registers itself as a participant with the transaction coordinator. Therefore, in an application server that runs XActor, all the enlistments are automatically handled: the application server automatically enlists XA resources in the transaction branch, and XActor automatically registers itself as a participant with the transaction coordinator.

Transaction-aware Web services. The automatic registration support of XActor affords our plug-in a very attractive feature: a Web service does not need to explicitly interact with the TM to register a participant with the transaction coordinator. The Web service only needs to be configured to use the transactional interceptors. Therefore, converting a non-transaction-aware Web service into a transaction-aware one is just a matter of changing a deployment descriptor. No changes in the source code are necessary. This is not the case in TMs that do not support automatic registration. Those TMs require changes in the code of the Web service to register a participant with the coordinator.

Propagation of transaction identifiers. The XA specification defines a structure named `Xid`, which uniquely identifies a distributed transaction. An `Xid` has three components, but only two of them are relevant for identifying a distributed transaction: the format identifier (`FormatId`), which is an integer, and the global transaction identifier (`GlobalId`), which is a byte array.

In a transaction spanning several TMs (or, equivalently, several application servers), all accesses to RMs should be tied to the same [`FormatId`, `GlobalId`] pair. For this reason, that pair needs to be propagated along with transactional SOAP calls between application servers. The [`FormatId`, `GlobalId`] pair should therefore correspond to the globally unique transaction identifier included into the transaction context. The WS-C/WS-AT standards, however, specify that the transaction identifier field of the transaction context is a URI. Moreover, those specifications do not define a standard way of encoding a [`FormatId`, `GlobalId`] pair as a URI.

Nevertheless, the WS-C/WS-AT transaction context is extensible, i.e., it has extension elements that may be used to convey additional information. Those elements may therefore be used to propagate the [`FormatId`, `GlobalId`] pair in an implementation-specific way. This is exactly what our plug-in does in order to propagate transaction identifiers across XActor instances.

The arrangement described above supports transactions spanning several XActor instances. Unfortunately, that solution is not applicable to multi-vendor scenarios, due to the lack of an agreed-upon way of encoding a [`FormatId`, `GlobalId`] pair as a URI. This may lead to deadlocks. For example, if two application servers access the same RM using different [`FormatId`, `GlobalId`] pairs, the RM will consider each access as part of a different transaction, even if both accesses are actually part of the same distributed transaction. A deadlock will occur if both application servers attempt to update the same piece of information (since the RM is not aware that both update requests are part of the same transaction).

3.3. Support for the SOAP/HTTP invocation mechanism

XActor supports distributed transactions in an extensible way. It knows two general kinds of transactional resources: XA resources, such as XA-enabled databases and message queues, and remote resources, which are accessible to the TM through a remote invocation mechanism. From the point of view of the coordinator, a remote resource represents a transaction branch that lives in another TM. The coordinator uses a remote invocation mechanism to communicate with the TM that hosts the foreign transaction branch. Our plug-in enhances XActor with support to remote resources accessible through the SOAP/HTTP invocation mechanism.

Stub wrappers. The XActor instance that coordinates a distributed transaction is unaware of the remote invocation mechanisms through which it communicates with other TMs. It interacts with those TMs only through an interface named `org.xactor.Resource`, which contains the operations called by the coordinator during the execution of the 2PC protocol. Our plug-in provides a stub wrapper[4] that implements the `Resource` interface and forwards the invocations, using SOAP messages, to a transaction participant (Fig. 6).

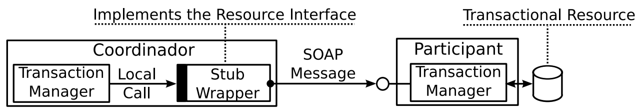


Figure 6. The stub wrapper that implements the `Resource` interface.

An instance of XActor that plays a participant role in a transaction is also unaware of the remote invocation mechanism through which it communicates with the transaction coordinator. A transaction participant interacts with its coordinating TM only through an interface named

[4]A stub wrapper encapsulates a protocol-specific stub (e.g., an IIOP stub or a SOAP/HTTP stub) that acts as a local proxy for a remote party.

`org.xactor.Coordinator`. Our plug-in provides a stub wrapper that implements the `Coordinator` interface and forwards the invocations, via SOAP messages, to the transaction coordinator.

3.4. Transaction recovery

To enable crash recovery, each TM involved in a distributed transaction has a transactional log, maintained in stable storage. At key points of the execution of the 2PC protocol, the coordinator and the participants write records on their respective transactional logs. Such records contain sufficient information to either ratify or abort the modifications made within the scope of the distributed transaction.

The coordinator records references to all the TMs that contain branches of a given distributed transaction (i.e., to all the transaction participants). The presence of those references in the log of the coordinator is important because if the coordinator crashes, it will still be able to contact, at recovery time, the TMs that hold the branches of the distributed transaction. The participants record in their log a reference to the coordinator. Such reference is essential for a crashed participant to reestablish communication with the coordinator at crash recovery time.

References to TMs. Since the WS-C/WS-AT standards define port types to enable inter-TM communication, the references to TMs are written in the transactional log as references to Web services that implement those port types. The WS-Addressing [17] standard defines an XML element, named *endpoint reference*, that fully describes a Web service endpoint. Our plug-in uses this standardized element to record references to remote TMs in the transactional log. Fig. 7 shows an example of endpoint reference that describes a transaction participant.

```
<wscoor:ParticipantProtocolService
 xmlns:wsa='http://schemas.xmlsoap.org/ws/2004/08/addressing'
 xmlns:wscoor='http://schemas.xmlsoap.org/ws/2004/10/wscoor'
 xmlns:xactcoor='http://xactor.sf.net/wscoor/2004/10'>
 <wsa:Address>http://10.0.0.1/xactor/Participant</wsa:Address>
 <wsa:ReferenceProperties>
  <xactcoor:ActivityID>3</xactcoor:ActivityID>
 </wsa:ReferenceProperties>
</wscoor:ParticipantProtocolService>
```

Figure 7. An endpoint reference.

The example has two nested elements, named `Address` and `ReferenceProperties`. The former holds the address of the Web service that represents the remote-accessible TM; the latter contains opaque information that is used to uniquely identify a transaction branch within the participant. The `ReferenceProperties` element is important because a participant may simultaneously host branches of several distributed transactions. Hence, when the coordinator sends a message to the participant, it has to specify, through the `ReferenceProperties` element, to which transaction branch that message is intended.

4. Related work

The CORBA Transaction Service (OTS) [9] is the relevant standard for distributed transactions in CORBA environments. It also plays a key role in the Java application server scenario, since the Java Transaction Service (JTS) [12] is a Java mapping of the OTS. Nevertheless, the OTS presents problems in interactions through firewalls and is not supported in Microsoft domains. The WS-C/WS-AT standards, on the other hand, use SOAP/HTTP to traverse firewalls and are supported by the Microsoft .NET platform.

Kandula [1] is an open-source implementation of the WS-C/WS-AT standards. It is developed under the Apache Software Foundation umbrella and builds upon the Axis SOAP stack. Kandula has two main shortcomings: it is tied to a specific SOAP stack and — more importantly — it does not support crash recovery. Our work does not have these deficiencies: it runs on a standard servlet container and does provide support for crash recovery.

JBoss Transactions (JBossTS) [5], formerly Arjuna Transaction Service [10], is an open-source transaction manager that bears similarities to XActor. The former has modules that implement the OTS and WS-C/WS-AT standards atop a transaction manager core; the latter implements the same standards as dynamically deployable plug-ins. The JBossTS modules that implement the WS-C/WS-AT standards currently have two shortcomings[5]: (*i*) its crash recovery mechanism for Web services transactions is not yet fully functional, and (*ii*) it requires that Web services explicitly register participants with the TM, which then registers itself with the coordinator. Our WS-C/WS-AT service does not have these shortcomings; furthermore, it offers additional features. One such feature allows transaction demarcation by stand-alone clients (i.e., clients that do not run on an application server). Another distinguishing feature — perhaps the most important one — is the support to distributed transactions that involve not only Web services, but also remote resources accessible through an open-ended set of transports. At the root of the latter point is the extensible architecture of the TM. XActor supports distributed transactions over any combination of the transports provided by its plug-ins. For example, with its IIOP and SOAP/HTTP plug-ins, XActor can coordinate distributed transactions that simultaneously involve transactional CORBA objects and transactional Web services. Moreover, the set of supported transports may grow at run time through the dynamic deployment of plug-in components.

5. Concluding remarks

This paper presented the design and implementation of a plug-in that enhances XActor with full support for atomic

[5]At the time of writing, the JBossTS team was addressing those issues.

transactions over Web services. Our future work plans include implementing other transaction models, such as the one defined by the WS-BA standard.

During the development of this work, we have identified weaknesses in the WS-AT standard. We have also drawn useful conclusions from our experience. These are the main lessons we learned from implementing WS-C and WS-AT:

- *Future revisions of WS-AT should support 1PC.* WS-AT does not currently support the one-phase commit (1PC) [3] optimization. Even if a distributed transaction involves only a single participant, WS-AT requires the execution of the full 2PC protocol. The lack of the 1PC in WS-AT is unfortunate, since it is an important and widely known optimization — both the OTS and XA standards support the 1PC.

- *Future revisions of WS-AT should address heuristic management.* WS-AT currently defines a single SOAP fault, named `InconsistentInternalState`. Transaction heuristics have to be conveyed within that fault message, in an implementation-specific format. Since all the heuristics take the form of an `InconsistentInternalState` fault, there is no standard way to differentiate them in multi-vendor environments.

- *Future revisions of WS-AT should standardize* `Xid` *URIs.* The WS-AT standard specifies that the transaction identifier is propagated as a URI, but it does not define a standard way of mapping [`FormatId`, `GlobalId`] pairs to URIs. The extension elements of the transaction context may be used to propagate [`FormatId`, `GlobalId`] pairs in implementation-specific ways. In multi-vendor scenarios, however, such arrangements lead to problems that range from suboptimal performance of transactions to deadlocks.

- *Transaction managers should be extensible.* The extensible architecture of XActor greatly simplified our work. To the best of our knowledge, no other transaction manager fully supports distributed transactions over an open-ended set of transports, which may even grow at run time. Supporting pluggable transports does not significantly raise the effort involved in building a transaction manager, nor does it impact the performance of the TM in a noticeable way [11]. There is therefore no reason for leaving that feature out from new TM projects.

- *Dependence on a SOAP stack is burdensome.* Tying the implementation of the WS-C/WS-AT port types to a SOAP stack is undesirable because changes in the stack may affect the behavior of the endpoints. Our work avoids dependence on specific frameworks by running within a standard servlet container and using thin software layers to integrate with SOAP stacks.

- *Performance-critical Web services require lightweight technologies.* Even though JAX-RPC and JAX-WS greatly simplify the development of Web services in the Java EE platform, they are unsuitable for the development of performance-critical Web services. We came to this conclusion after implementing a WS-C/WS-AT service prototype using JAX-RPC Web services and replacing that prototype by an implementation based on lighter technology (StAX).

The last two lessons concern not only WS-AT, but also WS-C. Those lessons should therefore be relevant to designers of other systems built atop WS-C, whose field of applicability goes well beyond transaction processing and comprises areas such as business processes and workflow management.

References

[1] Apache Kandula website, 2007. `http://ws.apache.org/kandula/`.

[2] BEA Systems. *Streaming API for XML Specification, v1.0*, October 2003.

[3] J. Gray and A. Reuter. *Transaction Processing: Concepts and Techniques*. Morgan Kaufmann, 1992.

[4] JBoss Remoting website, 2007. `http://labs.jboss.com/portal/jbossremoting/`.

[5] JBoss Transactions website, 2007. `http://www.jboss.com/products/transactions/`.

[6] OASIS. *Web Services Atomic Transaction 1.1*, April 2007.

[7] OASIS. *Web Services Business Activity 1.1*, April 2007.

[8] OASIS. *Web Services Coordination 1.1*, April 2007.

[9] Object Management Group. *CORBA Transaction Service Specification, version 1.4*, March 2003.

[10] G. D. Parrington, S. K. Shrivastava, S. M. Wheater, and M. C. Little. The Design and Implementation of Arjuna. *Computing Systems*, 8(3):255–308, 1995.

[11] F. Reverbel and I. Silva Neto. Dynamic support to transactional remote invocations over multiple transports. In *Proceedings of the 2008 ACM Symposium on Applied Computing*, pages 499–506, Fortaleza, Ceará, Brazil, March 2008.

[12] Sun Microsystems. *Java Transaction Service Specification, v1.0*, December 1999.

[13] Sun Microsystems. *Java API for XML-Based RPC Specification, v1.1*, October 2003.

[14] Sun Microsystems. *SOAP with Attachments API for Java Specification, v1.2*, October 2003.

[15] Sun Microsystems. *Java API for XML Web Services Specification, v2.0*, October 2005.

[16] The Open Group. *Distributed TP: The XA Specification*, February 1992.

[17] W3 Consortium. *Web Services Addressing 1.0 — Core*, May 2006.

[18] W3C Consortium. *Document Object Model (DOM) Level 3 Core Specification*, April 2004.

[19] C. Walls and R. Breidenbach. *Spring in Action, Second Edition*. Manning, 2007.