JBoss Transaction Service 4.5.0

Web Service Transactions Programmers Guide

JBXTS-PG-12/11/08

Legal Notices

The information contained in this documentation is subject to change without notice.

JBoss Inc. makes no warranty of any kind with regard to this material, including, but not limited to, the implied warranties of merchantability and fitness for a particular purpose. JBoss Inc. shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance, or use of this material.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Oracle® is a registered U.S. trademark and Oracle9™, Oracle9 Server™ Oracle9 Enterprise Edition™ are trademarks of Oracle Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Arjuna is a trademark of Hewlett-Packard Company and is used here under licence.

Copyright

Software Version

JBoss Transaction Service 4.5.0

Restricted Rights Legend

Use, duplication, or disclosure is subject to restrictions as set forth in contract subdivision (c)(1)(ii) of the Rights in Technical Data and Computer Software clause 52.227-FAR14.

Contents

Table Of Contents

About This Guide

What This Guide Contains

The Web Service Transactions Programmers Guide contains information on how to use JBoss Transaction Service 4.5.0. This guide provides information on how to develop service-based applications that use transaction technology to manage business processes. JBossTS provides a means of interacting with Web services within transactions, and constructing transaction-aware Web services, according to the WS-C, WS-Atomic Transaction and WS-Business Activity specifications, using common Web services platforms. While this guide discusses many of Web services standards like SOAP, WSDL and UDDI, it does not attempt to address all of their fundamental constructs. However, basic concepts are provided where necessary.

Audience

This guide is most relevant for application developers and Web service developers who are interested in building applications and Web services that are transaction-aware. This guide is also useful for system analysts and project managers that are unfamiliar with transactions as they pertain to Web services.

Prerequisites

JBossTS uses the Java programming language and this manual assumes that you are familiar with programming in Java. In addition, a fundamental level of understanding in the following areas will also be useful:

- A Working knowledge of Web services, including XML, SOAP, WSDL, and UDDI;
- A general understanding of transactions;
- A general understanding of WS-C, WS-Atomic Transaction and WS-Business Activity;

Note: This guide presents overview information for all of the above. However, to aid in understanding the Web Services component of JBossTS, the WS-C, WS-Atomic Transaction and WS-Business Activity specifications are discussed in great detail.

Organization

This guide contains the following chapters:

1. **Chapter 1, Introduction:** an overview of the features of the Web Service component of JBossTS.
2. **Chapter 2, Transactions overview:** a brief description of some basic transaction concepts and techniques relevant to understanding JBossTS.
3. **Chapter 3, WS-C, WS-Atomic Transaction and WS-Business Activity overview**: an overview of the Web services protocols supported by JBossTS.
4. **Chapter 4, Getting started**: how to get going with JBossTS and Web Services.
5. **Chapter 5, Participants**: a description of what a transactional participant is and how to write one.
6. **Chapter 6, The XTS API**: a detailed description of the API provided by JBossTS for use when building applications which use Web Services Transactions. This supplements the accompanying javadocs.
7. **Chapter 7, Stand alone coordination**: how to locate the coordinator service on a different host to the client or web services.
8. **Chapter 8, Participant crash recovery**: how to make transactional Web services resilient to crashes.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, "**Select File | Open**." indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: and **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or invalid test results.

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss Transaction Service 4.5.0 documentation set:

- **JBoss Transaction Service 4.5.0** *Release Notes*: Provides late-breaking information about JBoss Transaction Service 4.5.0**.**
- **JBoss Transaction Service 4.5.0** *Installation Guide*: This guide provides instructions for installing JBoss Transaction Service 4.5.0.
- **JBoss Transaction Service 4.5.0** *Failure Recovery Guide*: Provides guidance for administering the system.
- **JBoss Transaction Service 4.5.0** *Transactions API Guide*: Provides guidance for administering the system.
- **JBoss Transaction Service 4.5.0** *Transaction Core Programmers Guide*: Provides guidance for administering the system.
- **JBoss Transaction Service 4.5.0** *JTS Programmers Guide*: Provides guidance for administering the system.
- **JBoss Transaction Service 4.5.0** *Administration Guide*: Provides guidance for administering the system.

Contacting Us

Questions or comments about JBoss Transaction Service 4.5.0 should be directed to our support team.

Introduction

JBossTS Web Services Transactions overview

The XML Transaction Service component of JBossTS (shorthand referred to as XTS) supports the coordination of private and public Web services in a business transaction. Therefore, to understand XTS, you must be familiar with Web services, and also understand a little about transactions. This chapter introduces XTS and provides a brief overview of the technologies that form the Web services standard. Additionally, this chapter explores some of the fundamentals of transactioning technology and how it can be applied to Web services. Much of the content presented in this chapter is detailed throughout this guide; however, only overview information about Web services is provided. If you are new to creating Web services, please see consult your Web services platform documentation.

JBossTS provides as the XTS component a transaction solution for Web services. Using XTS, business partners can coordinate complex business transactions in a controlled and reliable manner. The JBossTS Web Services API supports a transactional coordination model based on the WS-Coordination, WS-Atomic Transaction and WS-Business Activity specifications. WS-Coordination (WS-C) is a generic coordination framework developed by IBM, Microsoft and BEA. WS-Atomic Transaction (WS-AT) and WS-Business Activity (WS-BA) together comprise the WS-Transaction (WS-T) transaction protocols that utilize this framework. JBoss Transaction Service 4.5.0. implements both version 1.0 and version 1.1 of these three specifications. Version 1.0 is available from http://www.ibm.com/developerworks/library/specification/ws-tx/. Version 1.1 is available from http://www.oasis-open.org/specs/index.php#wstransactionv1.1.

Note: The 1.0 and 1.1 specifications only differ in a small number of details. The rest of this document employs version 1.0 of these specifications when providing explanations and example code. On the few occasions where the modifications required to adapt these to the 1.1 specifications are not obvious an explanatory note will be provided.

Web services are modular, reusable software components that are created by exposing business functionality through a Web service interface. Web services communicate directly with other Web services using standards-based technologies such as SOAP and HTTP. These standards-based communication technologies allow Web services to be accessed by customers, suppliers, and trading partners, independent of hardware operation system or programming environment. The result is a vastly improved collaboration environment as compared to today's EDI and business-to-business (B2B) solutions—an environment where businesses can expose their current and future business applications as Web services that can be easily discovered and accessed by external partners.

Web services, by themselves, are not fault tolerant. In fact, some of the reasons that make it an attractive development solution are also the same reasons that service-based applications may have drawbacks:

- Application components that are exposed as Web services may be owned by third parties, which provides benefits in terms of cost of maintenance, but drawbacks in terms of having exclusive control over their behavior;
- Web services are usually remotely located which increases risk of failure due to increased network travel for invocations.

Applications that have high dependability requirements, must find a method of minimizing the effects of errors that may occur when an application consumes Web services. One method of safeguarding against such failures is to interact with an application's Web services within the context of a transaction. A transaction is simply a unit of work which is completed entirely, or in the case of failures is reversed to some agreed consistent state – normally to appear as if the work had never occurred in the first place. With XTS, transactions can span multiple Web services which mean that work performed across multiple enterprises can be managed with transactional support.

Managing Service-Based Processes

XTS allows you to create transactions that drive complex business processes spanning multiple Web services. Current Web services standards do not address the requirements for a high-level coordination of services since in today's Web services applications, which use single request/receive interactions, coordination is typically not a problem. However, for applications that engage multiple services among multiple business partners, coordinating and controlling the resulting interactions is essential. This becomes even more apparent when you realize that you generally have little in the way of formal guarantees when interacting with third-party Web services.

XTS provides the infrastructure for coordinating services during a business process. By organizing processes as transactions, business partners can collaborate on complex business interactions in a reliable manner, insuring the integrity of their data - usually represented by multiple changes to a database – but without the usual overheads and drawbacks of directly exposing traditional transaction-processing engines directly onto the web. The following example demonstrates how an application may manage service-based processes as transactions:

The application in question allows a user to plan a social evening. This application is responsible for reserving a table at a restaurant, and reserving tickets to a show. Both activities are paid for using a credit card. In this example, each service represents exposed Web services provided by different service providers. XTS is used to envelop the interactions between the theater and restaurant services into a single (potentially) long-running business transaction. The business transaction must insure that seats are reserved both at the restaurant and the theater. If one event fails the user has the ability to decline both events, thus returning both services back to their original state. If both events are successful, the user's credit card is charged and both seats are booked. As you may expect, the interaction between the services must be controlled in a reliable manner over a period of time. In addition, management must span several third-party services that are remotely deployed.

Caution: Without the backing of a transaction, an undesirable outcome may occur. For example, the user credit card may be charged, even though one or both of the bookings may have failed.

This simple example describes the situations where XTS excels at supporting business processes across multiple enterprises. This example is further refined throughout this guide, and appears as a standard demonstrator (including source code) with the XTS distribution.

Servlets

The WS-C, WS-Atomic Transaction and WS-Business Activity protocols are based on one-way interactions of entities rather than traditional synchronous request/response RPC style interactions. Entities (e.g. transaction participants) invoke operations on other entities (e.g. the transaction coordinator) in order to return responses to requests. What this means is that the programming model is based on peer-to-peer relationships, with the result that all services, whether they are participants, coordinators or clients, must have an active component that allows them to receive unsolicited messages.

In the current implementation of XTS, the active component is achieved through the use of Java servlet technology. Each endpoint that can be communicated with via SOAP/XML is represented as a servlet (and published within JNDI). Fortunately for the developer, this use of servlets occurs transparently. The only drawback is that (currently) clients must reside within a domain capable of hosting servlets, i.e., an application server. It is our intention that future versions of XTS will provide configurable deployment options, allowing servlets where required, but not mandating them.

Note: The 1.1 XTS implementation is implemented using JaxWS services.

SOAP

SOAP has emerged as the de-facto message format for XML-based communication in the Web services arena. It is a lightweight protocol that allows the user to define the content of a message and to provide hints as to how recipients should process that message.

SOAP messages can be divided into two main categories: Remote Procedure Call (RPC) and Document Exchange (DE). The primary difference between the two categories is that the SOAP specification defines encoding rules and conventions for RPC. The document exchange model allows the exchange of arbitrary XML documents - a key ingredient of B2B document exchange. XTS is based on the loosely coupled document-exchange style, yet it can support transactions spanning Web service that use either document-exchange or RPC.

Web Services Description Language (WSDL)

WSDL is an XML-based language used to define Web service interfaces. An application that consumes a Web service parses the service’s WSDL document to discover the location of the service, the operations that the service supports, the protocol bindings the service supports (SOAP, HTTP, etc), and how to access them (for each operation, WSDL describes the format that the client must follow).

Chapter 2

Transactions overview

Transactions have emerged as the dominant paradigm for coordinating interactions between parties in a distributed system, and in particular to manage applications that require concurrent access to shared data. Much of the JBossTS Web Service API is based on contemporary transaction APIs whose familiarity will enhance developer productivity and lessen the learning curve. While the following section provides the essential information that you should know before starting to use XTS for building transactional Web Services, it should not be treated as a definitive reference to all transactional technology.

A classic transaction is a unit of work that either completely succeeds, or fails with all partially completed work being undone. When a transaction is committed, all changes made by the associated requests are made durable, normally by committing the results of the work to a database. If a transaction should fail and is rolled back, all changes made by the associated work are undone. Transactions in distributed systems typically require the use of a transaction manager that is responsible for coordinating all of the participants that are part of the transaction.

The main components involved in using and defining transactional Web Services using XTS are illustrated in Figure 1.

- A Transaction Service: The Transaction Service captures the model of the underlying transaction protocol and coordinates parties affiliated with the transaction according to that model.
- A Transaction API: Provides an interface for transaction demarcation and the registration of participants.
- A Participant: The entity that cooperates with the transaction service on behalf of its associated business logic.
- The Context: Captures the necessary details of the transaction such that participants can enlist within its scope.

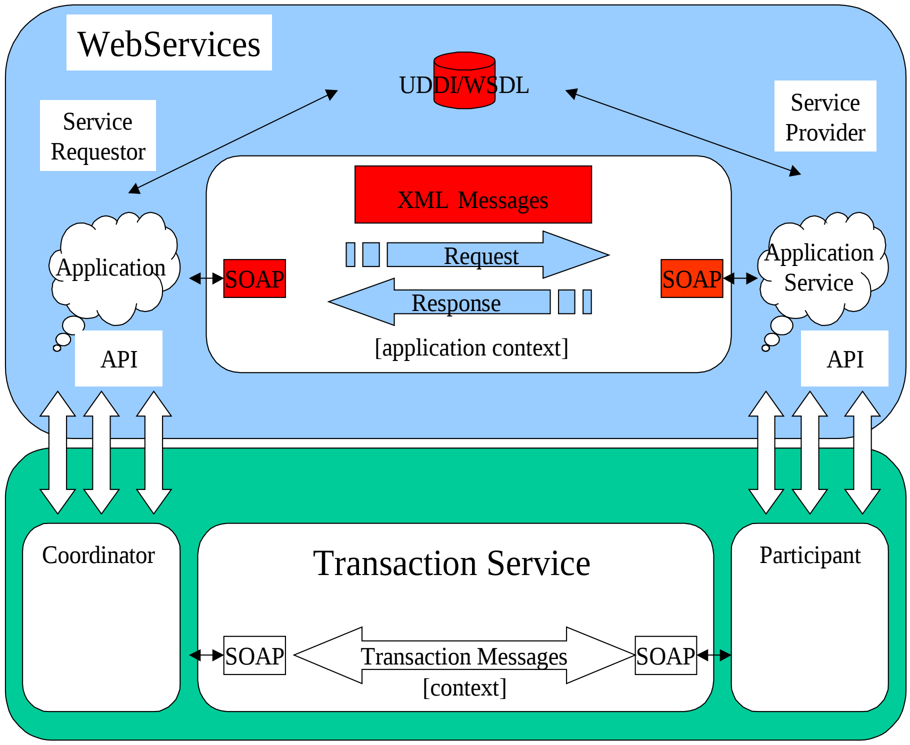

Figure 1 Web Services and XTS

The Coordinator

Associated with every transaction is a coordinator, which is responsible for governing the outcome of the transaction. The coordinator may be implemented as a separate service or may be co-located with the user for improved performance. Each coordinator is created by the transaction manager service, which is in effect a factory for those coordinators.

A coordinator communicates with enrolled participants to inform them of the desired termination requirements, i.e., whether they should accept (e.g., confirm) or reject (e.g., cancel) the work done within the scope of the given transaction. For example, whether to purchase the (provisionally reserved) flight tickets for the user or to release them. An application/client may wish to terminate a transaction in a number of different ways (e.g., confirm or cancel). However, although the coordinator will attempt to terminate in a manner consistent with that desired by the client, it is ultimately the interactions between the coordinator and the participants that will determine the actual final outcome.

A transaction manager is typically responsible for managing coordinators for many transactions. The initiator of the transaction (e.g., the client) communicates with a transaction manager and asks it to start a new transaction and associate a coordinator with the transaction. Once created, the context can be propagated to Web services in order for them to associate their work with the transaction.

The Transaction Context

In order for a transaction to span a number of services, certain information has to be shared between those services in order to propagate information about the transaction. This information is known as the Context. Using XTS, the context is automatically propagated and processed by transaction-aware components of an application. Though XTS removes most of the work associated with propagating contexts, it is still instructive to understand what information is captured in a context:

- A transaction identifier which guarantees global uniqueness for an individual transaction;
- The transaction coordinator location or endpoint address so participants can be enrolled.

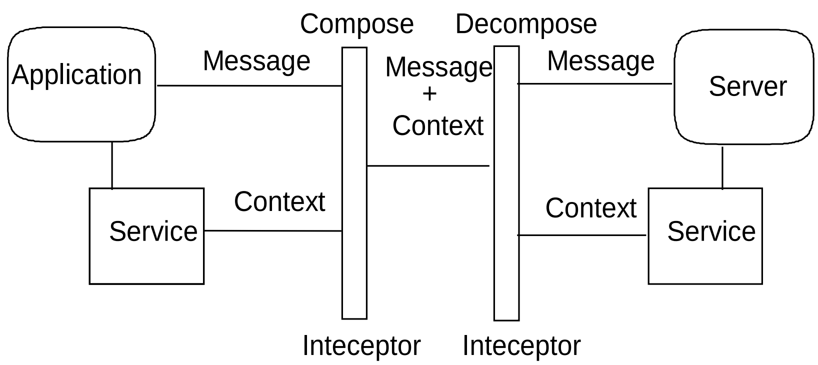

Figure 2 Web Services and Context Flow

As shown in Figure 2, whenever an application message is sent, the XTS Client API automatically creates a context and embeds it into the message. Similarly, any transaction-aware services are able to extract that context using the XTS service-side infrastructure and use it to perform work within the context of a particular transaction – even though that transaction was initiated elsewhere on the Web! The value of this approach is that the business logic contained within the client application and services are not peppered with transaction-processing code.

Participants

The coordinator cannot know the details of how every transactional service is implemented; in fact it is not necessary for it to do so in order to negotiate a transactional outcome. It treats each service taking part in a transaction as a *participant* and communicates with it according to some predefined *participant coordination models* appropriate to the type of transaction. When a service begins performing work within the scope of a transaction it enrolls itself with the coordinator as a participant, specifying the participant model it wishes to follow. So, the term participant merely refers a transactional service enrolled in a specific transaction using a specific participant model.

ACID Transactions

Traditionally, transaction processing systems support ACID properties. ACID is an acronym for Atomic, Consistent, Isolated, and Durable. A unit of work has traditionally been considered transactional only if the ACID properties are maintained:

- Atomicity: The transaction executes completely or not at all.
- Consistency: The effects of the transaction preserve the internal consistency of an underlying data structure.
- Isolated: The transaction runs as if it were running alone with no other transactions running and is not visible to other transactions.
- Durable: the transaction's results will not be lost in the event of a failure.

Two-Phase Commit

The classical two-phase commit approach is the bedrock of JBossTS (and more generally of Web Services transactions). Two-phase commit provides coordination of parties that are involved in a transaction. In general, the flow of a two-phase commit transaction is as follows:

- A transaction is started, and some work is performed.
- Once the work is finished, the two-phase commit begins.
- The coordinator (transaction manager) of the transaction asks each resource taking part in the transaction whether it is prepared to commit.
- If all resources respond positively, the coordinator instructs all work performed to be made durable (usually committed to a database).
- If not, all work performed is rolled back (undone) such that the underlying data structures are in their original states.

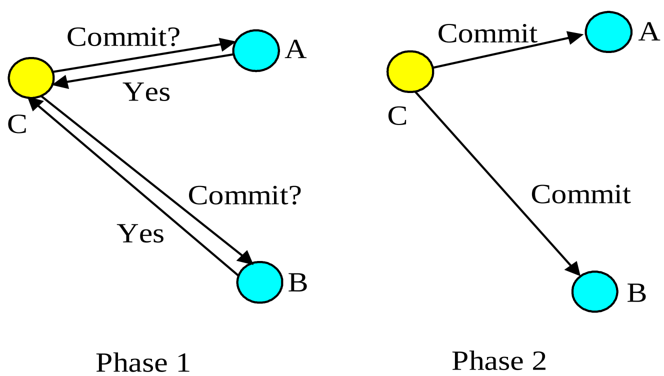

Figure 3 The Two-Phase Commit Protocol

> **Note:** During two-phase commit transactions, coordinators and resources keep track of activity in non-volatile data stores so that they can recover in the case of a failure.

The Synchronization Protocol

As well as the two-phase commit protocol, traditional transaction processing systems employ an additional protocol, often referred to as the synchronization protocol. If you recall the original ACID properties, then you'll remember that Durability is important in the case where state changes have to be available despite failures. What this means is that applications interact with a persistence store of some kind (e.g., a database) and this can impose a significant overhead – disk access is orders of magnitude slower than access to main computer memory.

One apparently obvious solution to this problem would be to cache the state in main memory and only operate on that for the duration of a transaction. Unfortunately you'd then need some way of being able to flush the state back to the persistent store before the transaction terminates, or risk losing the full ACID properties. This is what the synchronization protocol does, with Synchronization participants.

Synchronizations are informed that a transaction is about to commit, so they can, for example, flush cached state, which may be being used to improve performance of an application, to a durable representation prior to the transaction committing. They are then informed when the transaction has completed and in what state it completed.

- Synchronizations essentially turn the two-phase commit protocol into a four-phase protocol:
- Before the transaction starts the two-phase commit, all registered Synchronizations are informed. Any failure at this point will cause the transaction to roll back.
- The coordinator then conducts the normal two-phase commit protocol.
- Once the transaction has terminated, all registered Synchronizations are informed. However, this is a courtesy invocation because any failures at this stage are ignored: the transaction has terminated so there's nothing to affect.

Unlike the two-phase commit protocol, the synchronization protocol does not have the same failure requirements. For example, Synchronization participants don't need to make sure they can recover in the event of failures; this is because any failure before the two-phase commit protocol completes means the transaction will roll back, and failures after it has completed can't affect the data the Synchronization participants were managing.

Optimizations to the Protocol

There are several variants to the standard two-phase commit protocol that are worth knowing about because they can have an impact on performance and failure recovery. We shall briefly describe those that are the most common variants on the protocol:

- Presumed abort: if a transaction is going to roll back then the coordinator may simply record this information locally and tell all enlisted participants. Failure to contact a participant has no effect on the transaction outcome; the coordinator is effectively informing participants as a courtesy. Once all participants have been contacted the information about the transaction can be removed. If a subsequent request for the status of the transaction occurs there will be no information available and the requester can assume that the transaction has aborted (rolled back). This optimization has the benefit that no information about participants need be made persistent until the transaction has decided to commit (i.e. progressed to the end of the prepare phase), since any failure prior to this point will be assumed to be an abort of the transaction.
- One-phase: if there is only a single participant involved in the transaction, the coordinator need not drive it through the prepare phase. Thus, the participant will simply be told to commit and the coordinator need not record information about the decision since the outcome of the transaction is solely down to the participant.
- Read-only: when a participant is asked to prepare, it can indicate to the coordinator that no information or data that it controls has been modified during the transaction. Such a participant does not need to be informed about the outcome of the transaction since the

fate of the participant has no affect on the transaction. As such, a read-only participant can be omitted from the second phase of the commit protocol.

Note: WS-Atomic Transaction does not support the one-phase commit optimization.

Non-atomic Transactions and Heuristic Outcomes

In order to guarantee atomicity, the two-phase commit protocol is necessarily blocking. What this means is that as a result of failures, participants may remain blocked for an indefinite period of time even if failure recovery mechanisms exist. Some applications and participants simply cannot tolerate this blocking.

To break this blocking nature, participants that have got past the prepare phase are allowed to make autonomous decisions as to whether they commit or rollback: such a participant must record this decision in case it is eventually contacted to complete the original transaction. If the coordinator eventually informs the participant of the transaction outcome and it is the same as the choice the participant made, then there's no problem. However, if it is contrary, then a non-atomic outcome has obviously happened: a heuristic outcome.

How this heuristic outcome is reported to the application and resolved is usually the domain of complex, manually driven system administration tools, since in order to attempt an automatic resolution requires semantic information about the nature of participants involved in the transactions.

Precisely when a participant makes a heuristic decision is obviously implementation dependent. Likewise, the choice the participant makes (to commit or to roll back) will depend upon the implementation and possibly the application/environment in which it finds itself. The possible heuristic outcomes are:

- Heuristic rollback: the commit operation failed because some or all of the participants unilaterally rolled back the transaction.
- Heuristic commit: an attempted rollback operation failed because all of the participants unilaterally committed. This may happen if, for example, the coordinator was able to successfully prepare the transaction but then decided to roll it back (e.g., it could not update its log) but in the meanwhile the participants decided to commit.
- Heuristic mixed: some updates (participants) were committed while others were rolled back.
- Heuristic hazard: the disposition of some of the updates is unknown. For those which are known, they have either all been committed or all rolled back.

Heuristic decisions should be used with care and only in exceptional circumstances since there is the possibility that the decision will differ from that determined by the transaction service and will thus lead to a loss of integrity in the system. Having to perform resolution of heuristics is something you should try to avoid, either by working with services/participants that don't cause heuristics, or by using a transaction service that provides assistance in the resolution process.

A New Transaction Protocol

Many component technologies offer mechanisms for coordinating ACID transactions based on two-phase commit semantics (i.e., CORBA/OTS, JTS/JTA, MTS/MSDTC). ACID transactions are not suitable for all Web services transactions since:

- Classic ACID transactions are predicated on the idea that an organization that develops and deploys applications does so using their own infrastructure, typically an Intranet. Ownership meant transactions operated in a trusted and predictable manner. To assure ACIDity, potentially long-lived locks could be kept on underlying data structures during two-phase commit. Resources could be used for any period of time and released when the transaction was complete. In the Web services arena, these assumptions are no longer valid. One obvious reason is that the owners of data exposed through a Web service will refuse to allow their data to be locked for extended periods since to allow such locks invites denial-of-service.
- All application infrastructures are generally owned by a single party, systems using classical ACID transactions normally assume that participants in a transaction will obey the will of the transaction manager and only infrequently decide to make unilateral decisions which will hamper other participants in a transaction. On the contrary, Web services participating in a transaction can effectively decide to resign from the transaction at any time, and the consumer of the service generally has little in the way of quality of service guarantees to prevent this.

Addressing the Problems of Transactioning in Loosely Coupled Systems

Though extended transaction models which relax the ACID properties have been proposed over the years, WS-T provides a new transaction protocol to implement these concepts for the Web services architecture. XTS is designed to accommodate four underlying requirements inherent in any loosely coupled architecture like Web services:

- Ability to handle multiple successful outcomes to a transaction, with the ability to involve operations whose effects may not be isolated or durable;
- Coordination of autonomous parties whose relationships are governed by contracts rather than the dictates of a central design authority;
- Discontinuous service, where parties are anticipated to suffer outages during their lifetime, and coordinated work must be able to survive such outages;
- Interoperation using XML over multiple communication protocols – XTS chooses to use SOAP encoding carried over HTTP for the first release and other SOAP-friendly transports for future releases.

Chapter 3

WS-C, WS-Atomic Transaction and WS-Business Activity overview

Introduction

This section provides fundamental concepts associated with WS-C, WS-Atomic Transaction and WS-Business Activity. All of these concepts are defined in the WS-C, WS-Atomic Transaction and WS-Business Activity specifications. WS-C, WS-Atomic Transaction and WS-Business Activity principles are discussed throughout this guide.

Note: If you are well versed in the WS-C, WS-Atomic Transaction and WS-Business Activity specifications then you may want to just skim through this part of the manual.

WS-Coordination

In general terms, coordination is the act of one entity (known as the *coordinator*) disseminating information to a number of *participants* for some domain-specific reason. This reason could be in order to reach consensus on a decision like in a distributed transaction protocol, or simply to guarantee that all participants obtain a specific message, as occurs in a reliable multicast environment. When parties are being coordinated, information known as the *coordination context* is propagated to tie together operations which are logically part of the same coordinated work or *activity*. This context information may flow with normal application messages, or may be an explicit part of a message exchange and is specific to the type of coordination being performed.

The fundamental idea underpinning WS-Coordination is that there is a generic need for a coordination infrastructure in a Web services environment. The WS-Coordination specification defines a framework that allows different coordination protocols to be plugged-in to coordinate work between clients, services and participants, as shown in Figure 4.

The WS-Coordination specification talks in terms of activities, which are distributed units of work, involving one or more parties (which may be services, components, or even objects). At this level, an activity is minimally specified and is simply *created*, made to *run*, and then *completed*.

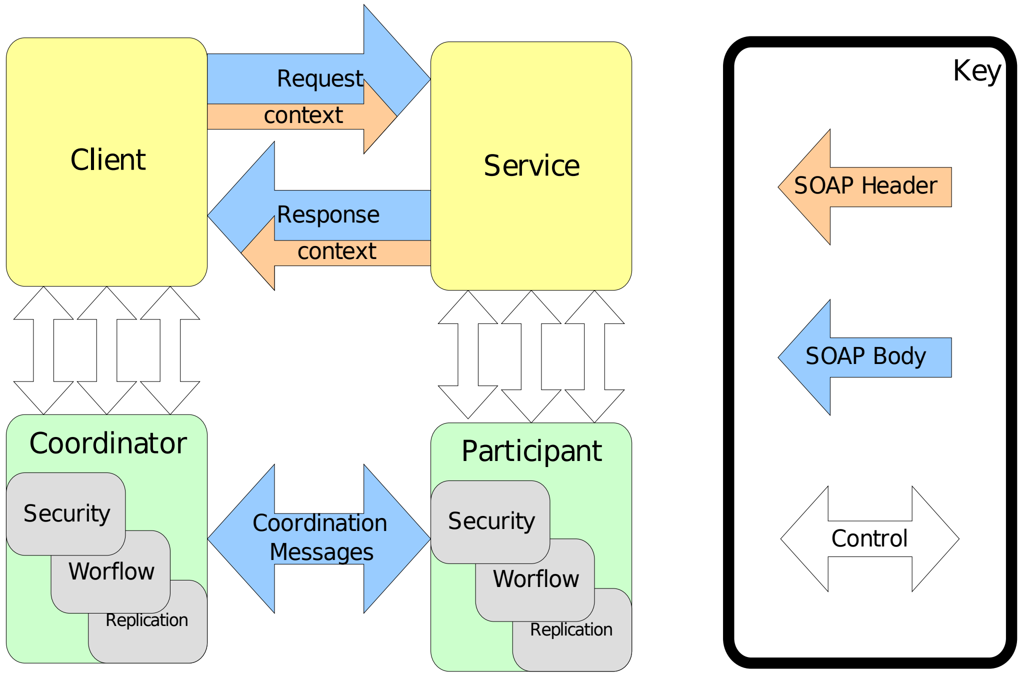

Figure 4 WS-C architecture.

Whatever coordination protocol is used, and in whatever domain it is deployed, the same generic requirements are present:

- Instantiation (or activation) of a new coordinator for the specific coordination protocol, for a particular application instance;
- Registration of participants with the coordinator, such that they will receive that coordinator's protocol messages during (some part of) the application's lifetime;
- Propagation of contextual information between Web services that comprise the application;
- An entity to drive the coordination protocol through to completion.

The first three of these points are directly the concern of WS-Coordination while the fourth is the responsibility of a third-party entity, usually the client application that controls the application as a whole. These four WS-Coordination roles and their interrelationships are shown in Figure 5.

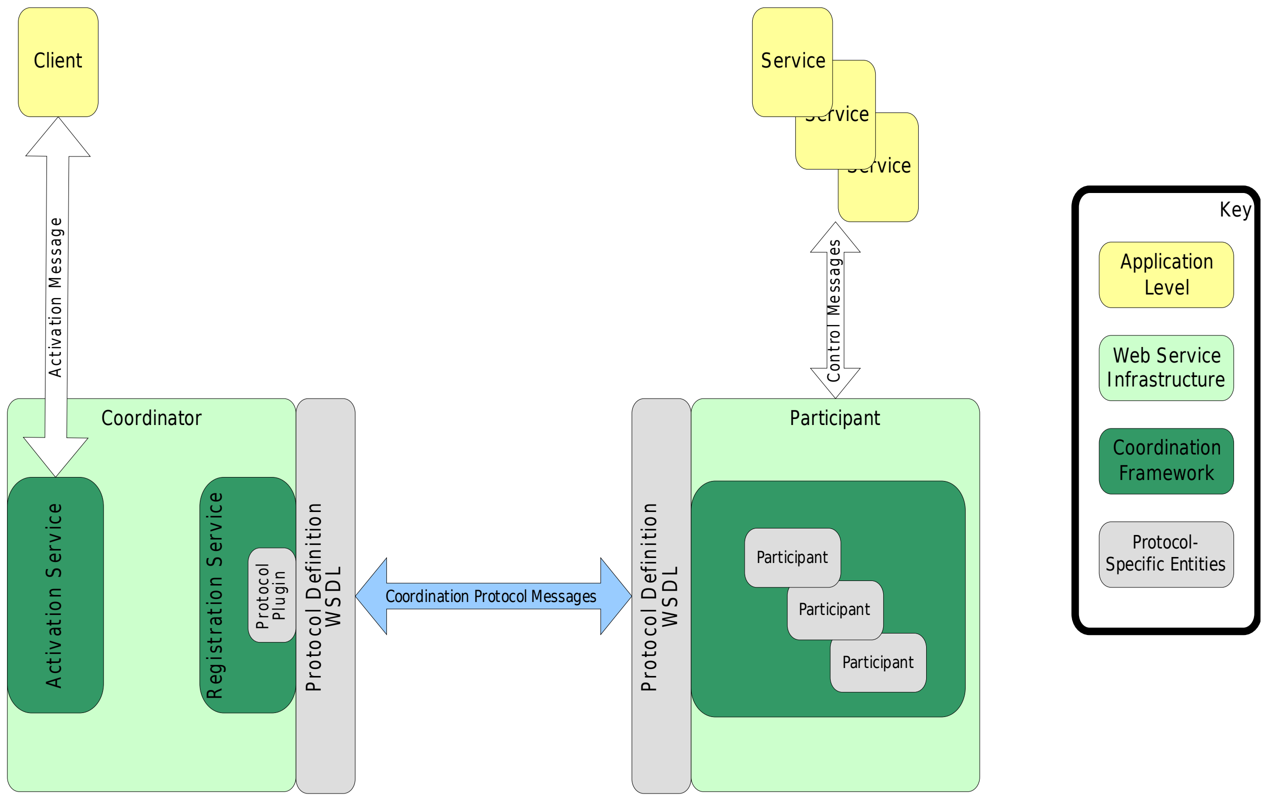

Figure 5 WS-C infrastructure

Activation

The WS-Coordination framework exposes an *Activation Service* which supports the creation of coordinators for specific coordination protocols and retrieval of associated contexts. The process of invoking an activation service is done asynchronously, and so the specification defines both the interface of the activation service itself, and that of the invoking service, so that the activation service can call back to deliver the results of the activation – namely a context that identifies the protocol type and coordinator location. These interfaces are presented in Figure 6, where the activation service has a one-way operation that expects to receive a `CreateCoordinationContext` message and correspondingly the service that sent the `CreateCoordinationContext` message expects to be called back with a `CreateCoordinationContextResponse` message, or informed of a problem via an `Error` message.

```
<!-- Activation Service portType Declaration -->
<wsdl:portType name="ActivationCoordinatorPortType">
  <wsdl:operation name="CreateCoordinationContext">
    <wsdl:input
     message="wscoor:CreateCoordinationContext"/>
  </wsdl:operation>
</wsdl:portType>

<!-- Activation Requester portType Declaration -->
<wsdl:portType name="ActivationRequesterPortType">
```

```
  <wsdl:operation
    name="CreateCoordinationContextResponse">
    <wsdl:input
     message="wscoor:CreateCoordinationContextResponse"/>
  </wsdl:operation>
  <wsdl:operation name="Error">
    <wsdl:input message="wscoor:Error"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 6 Activation Service WSDL Interfaces

> **Note:** The 1.1 Activation Coordinator service employs an RPC style message exchange avoiding the need to provide an Activation Requester service.

Registration

The context returned by the activation service includes information which exposes a *Registration Service*. This service allows participants to register to receive protocol messages from the coordinator associated with the context. Depending upon the coordination protocol there may be more than one choice of participant protocol available. Like the activation service, the registration service assumes asynchronous communication and so specifies WSDL for both registration service and registration requester, as shown in Figure 7.

```
<!-- Registration Service portType Declaration -->
<wsdl:portType name="RegistrationCoordinatorPortType">
  <wsdl:operation name="Register">
      <wsdl:input message="wscoor:Register"/>
  </wsdl:operation>
</wsdl:portType>

<!-- Registration Requester portType Declaration -->
<wsdl:portType name="RegistrationRequesterPortType">
  <wsdl:operation name="RegisterResponse">
      <wsdl:input message="wscoor:RegisterResponse"/>
  </wsdl:operation>
  <wsdl:operation name="Error">
      <wsdl:input message="wscoor:Error"/>
  </wsdl:operation>
</wsdl:portType>
```

Figure 7 Registration Service WSDL Interface

Once a participant is registered with a coordinator through the registration service, it receives coordination messages from the coordinator (for example, “prepare to complete” and "complete" messages if a two-phase protocol is used); where the coordinator's protocol supports it, participants can also send messages back to the coordinator.

Note: The 1.1 Registration Coordinator employs an RPC style message exchange avoiding the need to provide a Registration Requester service

Completion

The role of terminator is generally played by the client application, which at an appropriate point will ask the coordinator to perform its particular coordination function with any registered participants – to drive the protocol through to its completion. On completion, the client application may be informed of an outcome for the activity which may vary from simple succeeded/failed notification through to complete structured data detailing the activity's status.

WS-Transaction

In the past, making traditional transaction systems talk to one another was a holy grail that was rarely achieved. With the advent of Web services, there is an opportunity to leverage an unparalleled interoperability technology to splice together existing transaction processing systems that already form the backbone of enterprise level applications.

WS-Coordination Foundations

An important aspect of WS-Transaction that differentiates it from traditional transaction protocols is that a synchronous request/response model is not assumed. This model derives from the fact that WS-Transaction is, as shown in Figure 8, layered upon the WS-Coordination protocol whose own communication patterns are asynchronous by default.

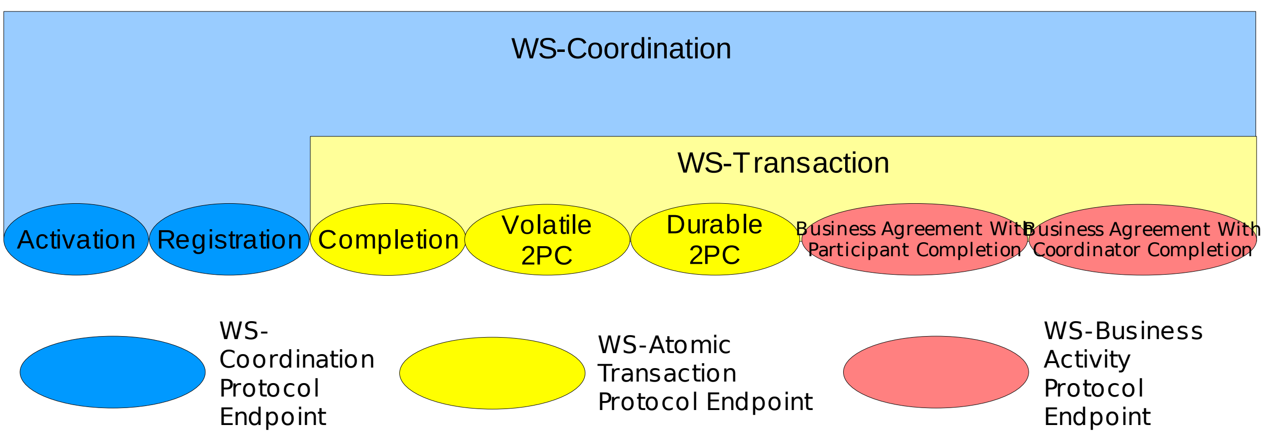

Figure 8 WS-Coordination WS-Atomic Transaction and WS-Business Activity

Web Services Coordination provides a generic framework for specific coordination protocols, like WS-Transaction, to be plugged in. Recall that WS-Coordination provides only context management – it allows contexts to be created and activities to be registered with those contexts. WS-Transaction leverages the context management framework provided by WS-Coordination in two ways. Firstly it extends the WS-Coordination context to create a transaction context. Secondly, it augments the activation and registration services with a number of additional services (`Completion`, `Volatile2PC`, `Durable2PC`, `BusinessAgreementWithParticipantCompletion`, and `BusinessAgreementWithCoordinatorCompletion`) and two protocol message sets (one for each of the transaction models supported in WS-Transaction) to build a fully-fledged transaction coordinator on top of the WS-Coordination protocol infrastructure.

WS-Transaction Architecture

WS-Transaction supports the notion of the service and participant as distinct roles, making the distinction between a transaction-aware service and the participants that act on behalf of the service during a transaction: transactional services deal with business-level protocols, while the participants handle the underlying WS-Transaction protocols, as shown in Figure 8.

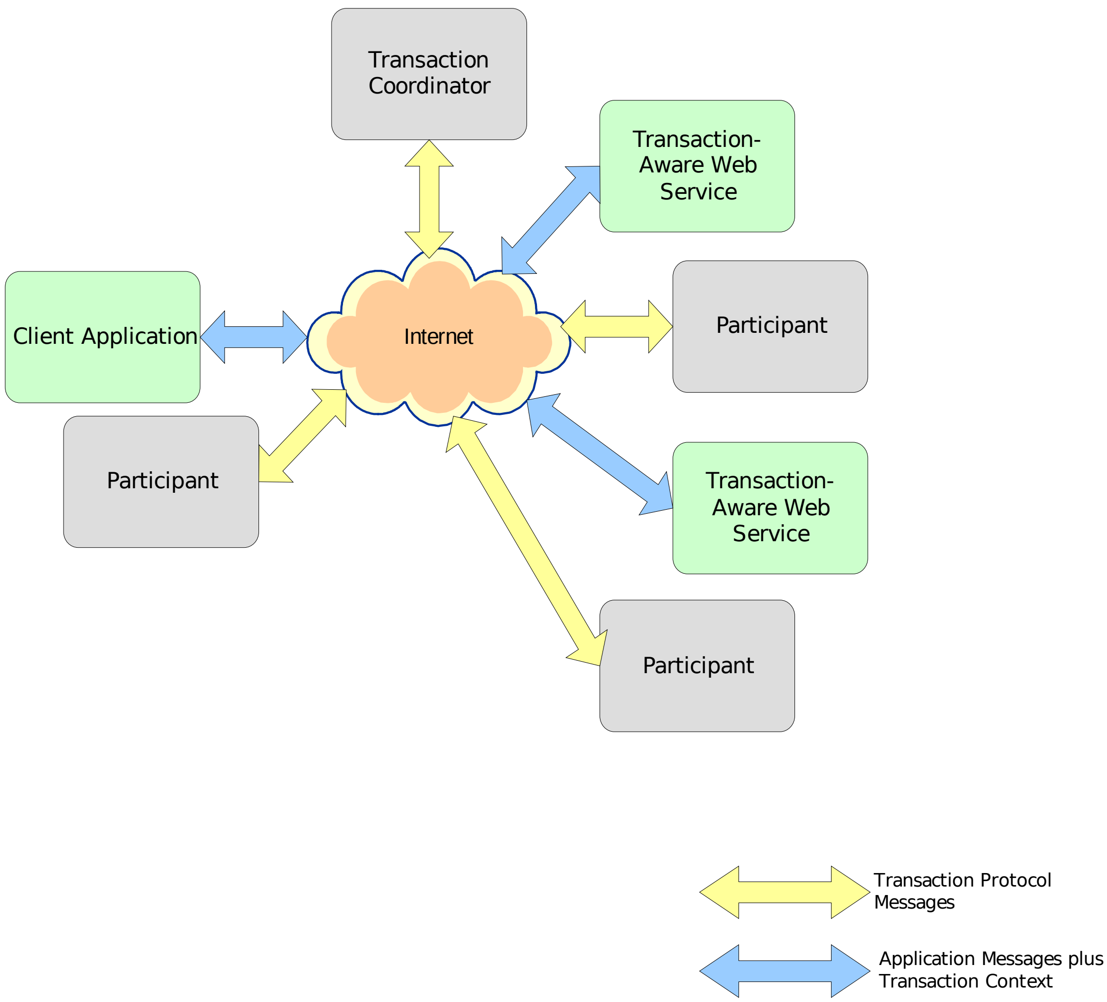

Figure 9 WS-Transaction Global View

A transaction-aware service encapsulates the business logic or work that is required to be conducted within the scope of a transaction. This work cannot be confirmed by the application unless the transaction also commits and so control is ultimately removed from the application and placed into the transaction's domain.

The participant is the entity that, under the dictates of the transaction coordinator, controls the outcome of the work performed by the transaction-aware Web service. In Figure 9 each service is shown with one associated participant that manages the transaction protocol messages on behalf of its service, while in Figure 10, there is a close-up view of a single service, and a client application with their associated participants.

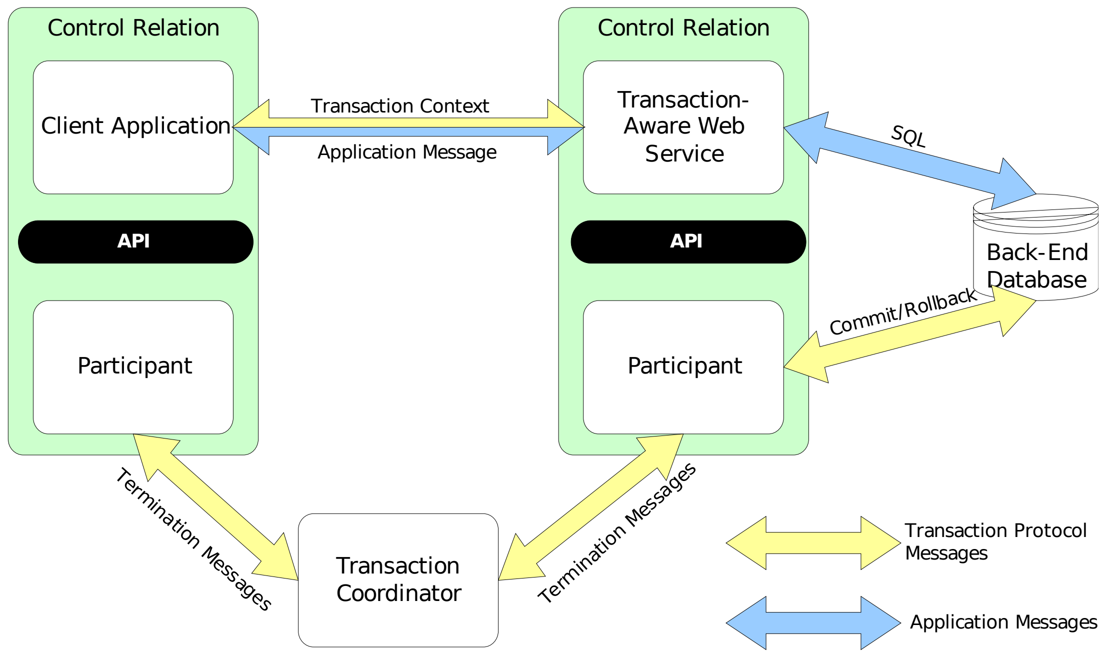

Figure 10 Transactional Service and Participant

The transaction-aware Web service and its participant both serve a shared transactional resource, and there is a control relationship between them through some API - which on the Java platform is JAXTX. In the example shown in Figure 10, it is assumed that the database is accessed through a transactional JDBC database driver, where SQL statements are sent to the database for processing via that driver, but where those statements will be tentative and only commit if the transaction does. In order to do this, the driver/database will associate a participant with the transaction which will inform the database of the transaction outcome. Since all transactional invocations on the Web service carry a transaction context, the participant working with the database is able to identify the work that the transactional service did within the scope of a specific transaction and either commit or rollback the work.

At the client end, things are less complex. Through its API, the client application registers a participant with the transaction through which it controls transaction termination.

WS-Transaction Models

Given that traditional transaction models are not appropriate for Web services, the following question must be posed, “what type of model or protocol *is* appropriate?” The answer to that question is that that no one specific protocol is likely to be sufficient, given the wide range of situations that Web service transactions are likely to be deployed within. Hence the WS-Transaction specification proposes two distinct models, where each supports the semantics of a particular kind of B2B interaction. The following sections shall discuss these two WS-Transaction models.

Note: The following discussion presents the interactions between the client, web service and the transaction coordinator in great detail for expository purposes only. Most of this activity happens automatically behind the scenes. The actual APIs used to initiate and complete a transaction and to register a participant and drive it through the commit or abort process are described in

Atomic Transactions (AT)

An atomic transaction or AT is similar to traditional ACID transactions and intended to support short-duration interactions where ACID semantics are appropriate. Within the scope of an AT, services typically enroll transaction-aware resources, such as databases and message queues, indirectly as participants under the control of the transaction. When the transaction terminates, the outcome decision of the AT is then propagated to each enlisted resource via the participant, and the appropriate commit or rollback actions are taken by each.

This protocol is very similar to those employed by traditional transaction systems that already form the backbone of an enterprise. It is assumed that all services (and associated participants) provide ACID semantics and that any use of atomic transactions occurs in environments and situations where this is appropriate: in a trusted domain, over short durations.

To begin an atomic transaction, the client application firstly locates a WS-Coordination Activation Coordinator web service that supports WS-Transaction. The client sends a WS-Coordination `CreateCoordinationContext` message to the service specifying `http://schemas.xmlsoap.org/ws/2004/10/wsat` as its coordination type and gets back an appropriate WS-Transaction context from the activation service. The response to the `CreateCoordinationContext` message, the transaction context, has its `CoordinationType` element set to the WS-Atomic Transaction namespace, `http://schemas.xmlsoap.org/ws/2004/10/wsat`, and also contains a reference to the atomic transaction coordinator endpoint (the WS-Coordination Registration Service) where participants can be enlisted.

Note: For the 1.1 specification the coordination type is supplied as `http://docs.oasis-open.org/ws-tx/wsat/2006/06`

The client will normally proceed to invoke Web services and then complete the transaction either committing all the changes made by the web services or, if something goes wrong, rolling them back. In order to be able to drive this completion activity the client must register itself as a participant for the `Completion` protocol by sending a `Register` message to the Registration Service whose endpoint was returned in the Coordination Context.

Once registered for `Completion`, the client application then interacts with Web services to accomplish its business-level work. With each invocation of a business Web service, the client inserts the transaction context into a SOAP header block, such that each invocation is implicitly scoped by the transaction. The toolkits that support WS-Atomic Transaction-aware Web services provide facilities to correlate contexts found in SOAP header blocks with back-end operations. This ensures that modifications made by the Web service are done within the scope of the same transaction as the client and subject to commit or rollback by the transaction coordinator.

Once all the necessary application level work has been completed, the client can terminate the transaction, with the intent of making any changes to the service state permanent. The completion participant instructs the coordinator either to try to commit or rollback the transaction. When the commit or rollback operation has completed, a status is returned to the participant to indicate the outcome of the transaction.

Although this description of the completion protocol makes it seem straightforward, it hides the fact that in order to resolve the transaction to an outcome several other participant protocols need to be followed.

The first of these protocols is the optional `Volatile2PC` (2PC is an abbreviation of the term two-phase commit). The `Volatile2PC` protocol is the WS-Atomic Transaction equivalent of the synchronization protocol we discussed earlier. It is typically executed where a Web service needs to flush volatile (cached) state, which may be being used to improve performance of an application, to a database prior to the transaction committing. Once flushed, the data will then be controlled by a two-phase aware participant.

When the completion participant initiates a commit operation all `Volatile2PC` participants are told that the transaction is about to complete (via the `prepare` message) and they can respond with either the `prepared`, `aborted` or `readonly` message; any failures at this stage will cause the transaction to rollback.

After `Volatile2PC prepare`, the next protocol to follow in WS-Atomic Transaction is `Durable2PC`. The `Durable2PC` protocol is at the very heart of WS-Atomic Transaction and is used to bring about the necessary consensus between participants in a transaction such that the transaction can safely be terminated.

The `Durable2PC` protocol is used to ensure atomicity between participants, and is based on the classic two-phase commit with presumed abort technique. During the first phase, when the coordinator sends the `prepare` message, a participant must make durable any state changes that occurred during the scope of the transaction, such that these changes can either be rolled back or committed later. That is, any original state must not be lost at this point as the atomic transaction could still roll back. If the participant cannot prepare then it must inform the coordinator (via the `aborted` message) and the transaction will ultimately roll back. If the participant is responsible for a service that did not do any work during the course of the transaction, or at least did not do any work that modified any state, it can return the `readonly` message and it will be omitted from the second phase of the commit protocol. Otherwise, the `prepared` message is sent by the participant.

Assuming no failures occurred during the first phase, in the second phase the coordinator sends the `commit` message to participants, who will make permanent the tentative work done by their associated services and respond by sending a `commited` message to the coordinator. If any failures occurred the coordinator sends the `rollback` message to all participants, who will discard tentative work done by their associated services, including deleting any state saved to persistent storage at prepare if they reached that stage. Participants respond to to a rollback by sending an `aborted` message to the coordinator.

Note: The WS-Atomic Transaction protocol specification does not include the one-phase commit optimization. A full two-phase commit will therefore be used even where there is only a single participant enlisted.

Figure 11[1] shows the state transitions of a WS-Atomic Transaction and the message exchanges between coordinator and participant; the coordinator generated messages are shown in the solid line, whereas the participant messages are shown by dashed lines.

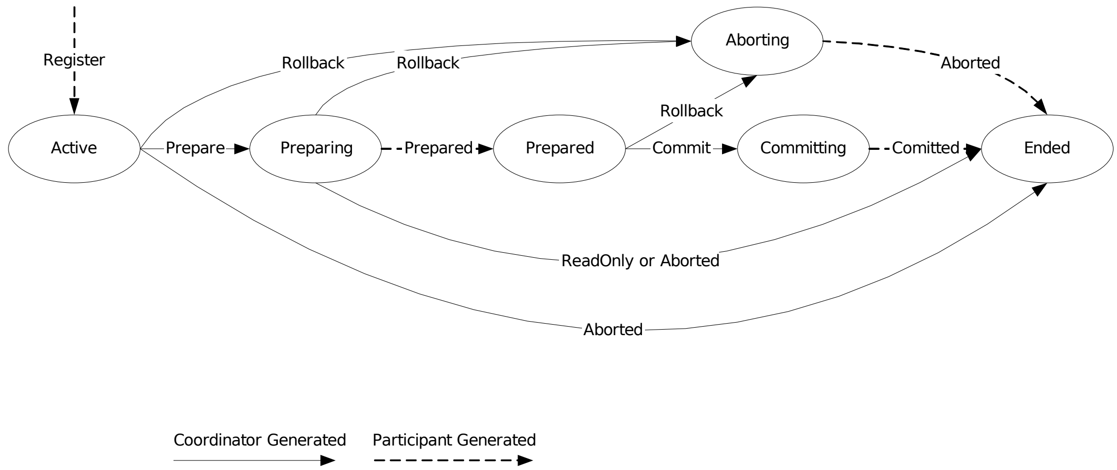

Figure 11 Two-Phase Commit Participant State Transitions

Once the 2PC protocol has finished, the `Completion` protocol that originally began the termination of the transaction can complete, and inform the client application whether the transaction was committed or rolled back. Additionally, the `Volatile2PC` protocol may complete.

Like the `prepare` phase of `Volatile2PC`, the final phase is optional and can be used to inform participants when the transaction has completed, typically so that they can release resources (e.g., put a database connection back into the pool of connections).

Any registered `Volatile2PC` participants are invoked after the transaction has terminated and are told the state in which the transaction completed (the coordinator sends either the `Committed` or `Aborted` message). Since the transaction has terminated, any failures of participants at this stage are ignored –it is essentially a courtesy, and has no bearing on the outcome of the transaction.

Finally, after having gone through each of the stages in an AT, it is possible to see the intricate interweaving of individual protocols that goes to make up the AT as a whole in Figure 12.

[1] Redrawn from the WS-Atomic Transaction specification.

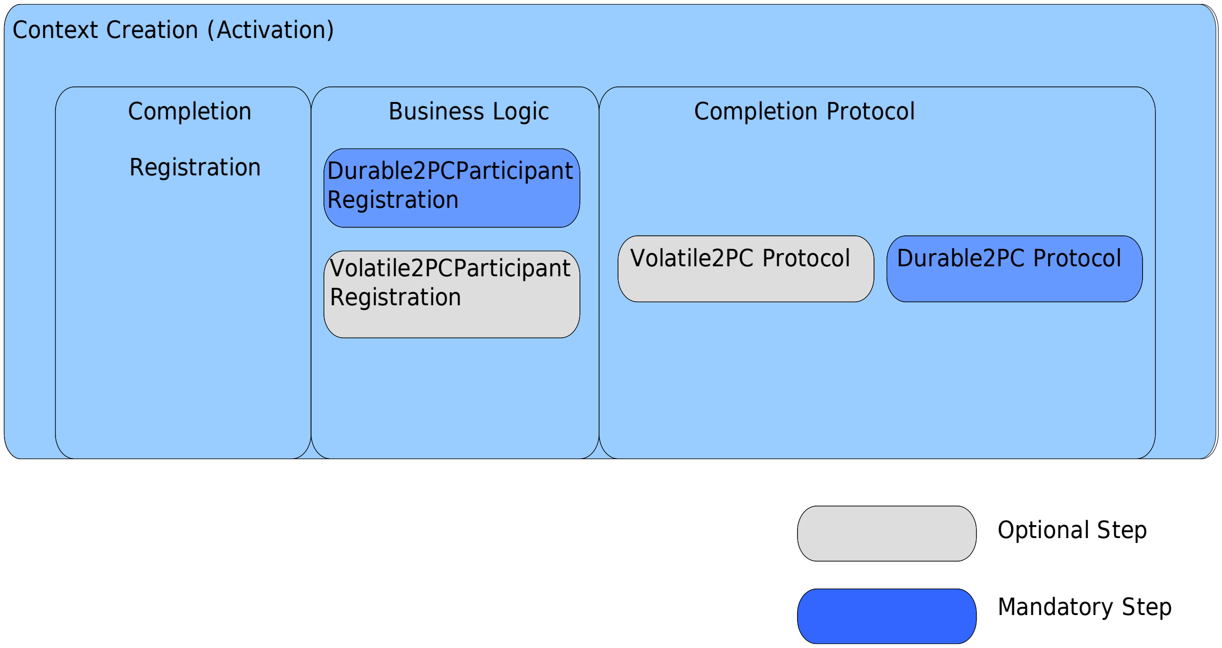

Figure 12 The AT Model

Business Activities (BA)

Most business-to-business applications require transactional support in order to guarantee consistent outcome and correct execution. These applications often involve long running computations, loosely coupled systems and components that do not share data, location, or administration and it is difficult to incorporate atomic transactions within such architectures. For example, an online bookshop may reserve books for an individual for a specific period of time, but if the individual does not purchase the books within that period they will be “put back onto the shelf” for others to buy. Furthermore, because it is not possible for anyone to have an infinite supply of stock, some online shops may appear to users to reserve items for them, but in fact may allow others to preempt that reservation (i.e., the same book may be “reserved” for multiple users concurrently); a user may subsequently find that the item is no longer available, or may have to be reordered specially for them.

A business activity or BA is designed specifically for these kinds of long-duration interactions, where exclusively locking resources is impossible or impractical. In this model services are requested to do work, and where those services have the ability to undo any work, they inform the BA such that if the BA later decides the cancel the work (i.e. if the business activity suffers a failure), it can instruct the service to execute its undo behavior. The key point for Business Activities is that how services do their work and provide compensation mechanisms is not the domain of the WS-Business Activity specification, but an implementation decision for the service provider.

The WS-Business Activity simply defines a protocol for Web services-based applications to enable existing business processing and workflow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

Unlike the WS-Atomic Transaction protocol model, where participants inform the coordinator of their state only when asked, a child activity within a business activity can specify its outcome to the coordinator directly without waiting for a request. A participant may choose to exit the activity or may notify a failure at any point. This feature is useful when tasks fail because the notification can be used to modify the goals and drive processing forward without having to meekly wait until the end of the transaction to identify failures – a well designed Business Activity should be proactive, if it is to perform well.

The Business Activity protocols employ a compensation-based transaction model. When a participant in a business activity has completed its work it may choose to *exit* the activity. This choice does not allow any subsequent rollback or undo of the changes it has made. Alternatively, the participant can *complete* its activity, signaling to the coordinator that the work it has done can be compensated if, at some later point, another participant notifies a failure to the coordinator. In this latter case, the coordinator will ask each non-exited participant to compensate for the failure, giving them the opportunity to execute whatever compensating action they consider appropriate (e.g. a participant might credit a bank account which was previously debited). In the event that all participants exit or complete without failure the coordinator notifies each completed participant that the activity has been closed.

Underpinning all of this are three fundamental assumptions:

- All state transitions are reliably recorded, including application state and coordination metadata (the record of sent and received messages);
- All request messages are acknowledged, so that problems are detected as early as possible. This avoids executing unnecessary tasks and can also detect a problem earlier when rectifying it is simpler and less expensive;
- As with atomic transactions, a response is defined as a separate operation and not as the output of the request. Message input-output implementations will typically have timeouts that are too short for some business activity responses. If the response is not received after a timeout, it is re-sent. This is repeated until a response is received. The request receiver discards all but one identical request received.

As with atomic transactions, the business activity model has multiple participant protocols: `BusinessAgreementWithParticipantCompletion` and `BusinessAgreementWithCoordinatorCompletion`. However, unlike the AT protocols which are driven from the coordinator down to participants, this protocol is driven much more from the participants upwards.

Under the `BusinessAgreementWithParticipantCompletion` protocol, a participant is initially created in the `Active` state; if it finishes the work it was created to do and no more participation is required within the scope of the BA (such as when the activity operates on immutable data), then the participant can unilaterally send an `exited` message to the coordinator. However, if the participant finishes and wishes to continue in the BA then it must be able to compensate for the work it has performed. In this case it sends a `completed` message to the coordinator and waits for the coordinator to notify the final outcome of the BA. This outcome will either be a `close` message, meaning the BA has completed successfully or a `compensate` message indicating that the coordinator requires the participant to reverse its work.

The `BusinessAgreementWithCoordinatorCompletion` protocol is identical to the `BusinessAgreementWithParticipantCompletion` protocol with the exception that the participant cannot autonomously decide to complete its participation in the business activity, even if it can be compensated. In this case the completion stage is driven by the client which created the business activity sending a `completed` message to the coordinator.

The coordinator sends a `complete` message to each participant indicating that no further requests will be sent to the service associated with the participant. The participant then acts as it does in the `BusinessAgreementWithParticipantCompletion` protocol.

The crux of the BA model compared to the AT model is that is allows the participation of services that cannot or will not lock resources for extended periods.

While the full ACID semantics are not maintained by a BA, consistency can still be maintained through compensation, though the task of writing correct compensating actions (and thus overall system consistency) is delegated to the developers of the services under control of the BA. Such compensations may use backward error recovery, but will typically employ forward recovery.

Figure 13[2] shows the state transitions of a WS-Business Activity BusinessAgreementWithParticipantCompletion participant and the message exchanges between coordinator and participant; the coordinator generated messages are shown in the solid line, whereas the participant messages are shown by dashed lines.

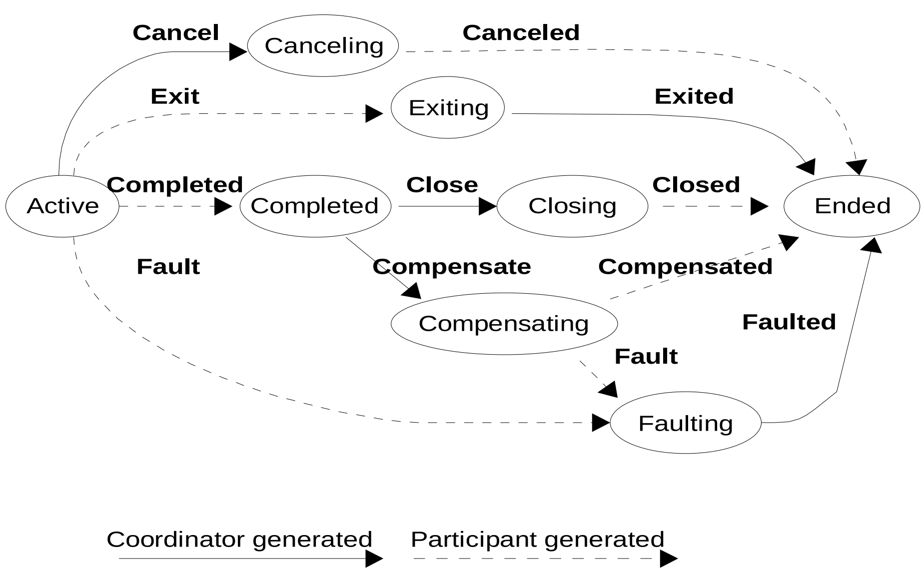

Figure 13 WS-BA 1.0 BusinessAgreementWithParticipantCompletion Participant State Transitions

Figure 14[3] shows the state transitions of a WS-Business Activity BusinessAgreementWithCoordinatorCompletion participant and the message exchanges between coordinator and participant; the coordinator generated messages are shown in the solid line, whereas the participant messages are shown by dashed lines.

Note: The WS-BA 1.1 specification has a more complex failure model: the Faulting state is replaced with a Failing state which is transitioned to from the Active,

[2] Redrawn from the WS-Business Activity specification.
[3] Redrawn from the WS-Business Activity specification.

Canceling and Compensating states if the participant sends a Fail message; a NotCompleting state is included which is transitioned to from the Active state if the participant sends a CannotComplete message. See the specification for full details.

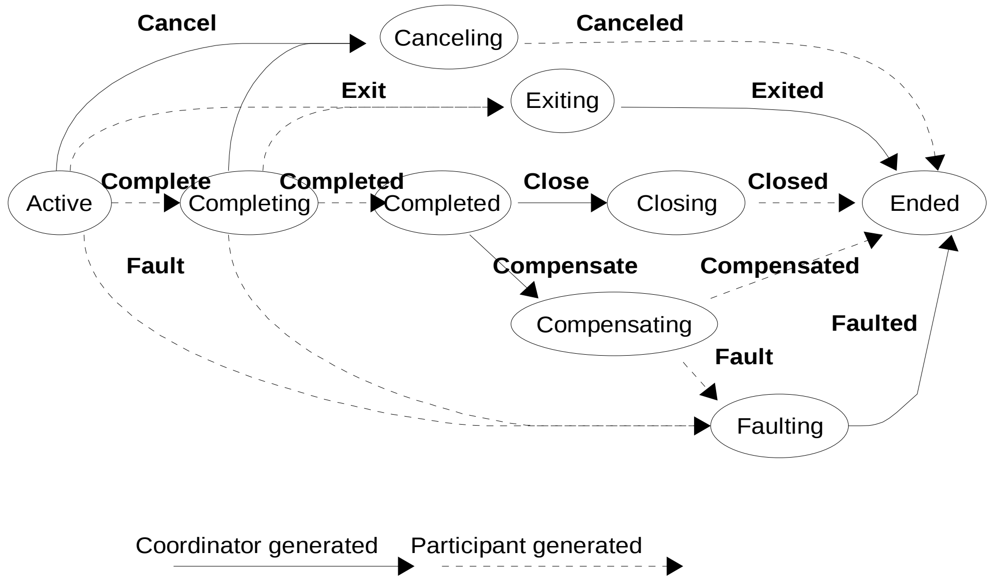

Figure 14 WS-BA 1.0 BusinessAgreementWithCoordinatorCompletion Participant State Transitions

Application Messages

Application messages are the requests and responses that are sent between parties that constitute the work of a business process. Any such messages are considered opaque by XTS, and there is no mandatory message format, protocol binding, or encoding style so the developer is free to use any appropriate Web services protocol. In XTS, the transaction context is propagated within the headers of SOAP messages.

Note: XTS provides out-of-box support for service developers building WS-T-aware services on the JBoss platform[4]. The provision of interceptors for automatic context handling at both client and service significantly simplifies the developer's workload, allowing the developer to concentrate on writing the business logic without having to worry about the transactional infrastructure getting in the way. The interceptors simply add and remove context elements to application messages without altering the semantics of those messages. Any service which understands what to do with a WS-C context can use it, services which do not understand the context (those services that are not WS-C, WS-Atomic Transaction and WS-Business Activity-aware) may ignore the context; the important point here is that XTS manages contexts without user intervention.

[4] Future versions of JBossTS will support other SOAP platforms.

WS-C, WS-Atomic Transaction and WS-Business Activity Messages

Although the application or service developer rarely sees or is interested in the messages exchanged by the transactional infrastructure (the transaction manager and any participants), it is useful to understand what kinds of exchanges occur so that the underlying model can be fitted in to an overall architecture.

In XTS, WS-Coordination, WS-Atomic Transaction and WS-Business Activity-specific messages are transported using SOAP messaging over HTTP. The types of messages that are propagated include instructions to perform standard transaction operations like begin, prepare.

> **Note:** XTS messages do not interfere in any way, shape, or form, with the application messages, and nor is there any requirement for an application to use the same transport as the transaction-specific messages. For example, it is quite reasonable for a client application to deliver its application-specific messages using SOAP RPC over SMTP even though under the covers the XTS messages are delivered using a different mechanism.

Summary

XTS provides a coordination infrastructure designed to allow transactions to run between enterprises across the Internet. That infrastructure is based on the WS-C, WS-Atomic Transaction and WS-Business Activity specifications. It supports two kinds of transactions: *atomic transactions* and *business activities*, which can be combined in arbitrary ways to map elegantly onto the transactional requirements of the underlying problem. The use of the whole infrastructure is simple due to the fact that its functionality is exposed through a simple transactioning API. Furthermore XTS provides all of the necessary plumbing to keep application and transactional aspects of an application separate, and to ensure that the transactionality of a system does not interfere with the functional aspects of the system itself.

Getting started

Creating Client Applications

There are two aspects to a client application using XTS. The first is the transaction declaration aspects. The second is the business logic, including, in particular, the invocation of web services.

Transaction declaration aspects are taken care of automatically with XTS's client API. This API provides simple transaction directives like begin, commit, and rollback which the client application can use to initialize, manage, and terminate transactions. Under the covers, this API invokes (via SOAP) operations on the various WS-C, WS-AT and WS-BA services required in order to create a coordinator and drive the transaction to completion.

User Transactions

A client creates and manages WS-AT transactions using the `UserTransactionFactory` and `UserTransaction` classes. These provide a very simple API which operates in a manner very similar to the JTA API. A WS-AT transaction is started and associated with the client thread by calling method `begin` of class `UserTransaction`. The transaction can be committed by calling method `commit` and rolled back by calling method `rollback`.

More complex transaction management such as suspension and resumption of transactions is supported by classes `TransactionManagerFactory` and `TransactionManager`.

Full details of the WS-AT APIs are provided in .

Business Activities

A client creates and manages Business Activities using the `UserBusinessActivityFactory` and `UserBusinessActivity` classes. A WS-BA activity is started and associated with the client thread by calling method `begin` of class `UserBusinessActivity`. A client can terminate a business activity by calling method `close` and cancel it by calling method `cancel`. If any of the web services invoked by the client register for the `BusinessActivityWithCoordinatorCompletion` protocol then the client can call method `completed` before calling `close` to notify the services that it will make no further service invocations in the current activity.

More complex business activity management such as suspension and resumption of business activities is supported by classes `BusinessActivityManagerFactory` and `BusinessActivityManager`.

Full details of the WS-AT APIs are provided in

Client Side Handler Configuration

XTS does not dictate an API that the client application must use to perform invocations on transactional Web services. The client is free to use whatever API it desires to send SOAP messages to the server and receive a SOAP response. The only requirements imposed on the client are that it must forward details of the current transaction to the server when invoking a web service and that it must process any responses from the server in the context of the correct transaction. In order to achieve this the client must insert details of the current XTS context into the headers of outgoing SOAP messages and must extract the context details from the headers of incoming messages and associate the context with the current thread. To make the user's life easier, the XTS software comes complete with "handlers" which can perform this task automatically. These handlers are designed to work with JAX-RPC and JAX-WS clients.

Note: If the user chooses to use a different SOAP client/server infrastructure for business service invocations, then the onus to perform header processing rests with them. XTS does not provide interceptors for anything other than JAX-RPC or JAX-WS for this release.

JAX-RPC Client Context Handlers

In order to register the JAX-RPC client-side context handler used by the client applications, a handler chain must be included in the definition of the `service-ref` in the client `web.xml` deployment descriptor. Please refer to the demo application `ddrpc/jboss/client-web-app.xml` for an example of how this can be achieved.

JAX-WS Client Context Handlers

In order to register the JAX-WS client-side context handler the client application uses the APIs provided by classes `javax.xml.ws.BindingProvider` and `javax.xml.ws.Binding` to install a handler chain on the service proxy used to invoke the remote endpoint. Please refer to the demo application client implementation `src/com/jboss/jbosstm/xts/demo/BasicClient.java` for an example of how this can be achieved.

Creating Transactional Web Services

There are two aspects to implementing a Web service using XTS. The first is the transaction management and the second is the business logic.

The bulk of the transaction management aspects are organized in a clear and easy-to-implement model by means of the XTS's participant API. This API provides a structured model for negotiation between the web service and the transaction coordinator. It allows the web service to manage its own, local transactional data in accordance with the needs of the business logic while still ensuring that its activities are in step with those of the client and other services involved in the transaction. Under the covers, this API invokes (via SOAP) operations on the various WS-C, WS-AT and WS-BA services required in order to drive the transaction to completion.

Participants

A participant is a software entity which is driven by the transaction manager on behalf of a Web service. When a web service wishes to participate in a particular transaction it must enrol a participant to act as a proxy for the service in subsequent negotiations with the coordinator. The participant implements an API appropriate to the type of transaction it is enrolled in and the participant model selected when it is enrolled. For example a Durable2PC participant as part of a WS-Atomic Transaction would implement the `Durable2PCParticipant` interface. The use of participants allows the transactional control management aspects of the Web service to be factored into the participant implementation separate from the the rest of the Web service's business logic and (private) transactional data management.

The creation of participants is non-trivial since they ultimately reflect the state of a Web service's back-end processing facilities which is a function of an enterprise's own IT infrastructure. The most that can be said about the implementation of a participant without getting into detail about the back-end systems it represents, or the details of the underlying transaction protocol is that implementations must implement one of the following interfaces, depending upon the protocol it will participate within: `com.arjuna.wst.Durable2PCParticipant`, `com.arjuna.wst.Volatile2PCParticipant`, or, `com.arjuna.wst.BusinessAgreementWithParticipantCompletionParticipant`, `com.arjuna.wst.BusinessAgreementWithCoordinatorCompletionParticipant`.

A full description of XTS's participant features is provided in .

Note: The 1.1 XTS implementation expects participants to implement the same APIs as the 1.0 implementation.

Service Side Handler Configuration

A transactional Web service must be ensure that service invocation is included in the appropriate transaction. This usually just affects the operation of the participants and has no impact on the operation of the rest of the Web service. Once again, XTS simplifies this task and decouples it from the business logic, in much the same way as was done for transactional clients. XTS provides a handler which detects and extracts the context details from the headers in incoming SOAP headers and associates the web service thread with the transaction. The handler clears this association when dispatching SOAP responses and writes the context into the outgoing message headers. This is shown in Figure 15.

JAX-RPC Service Context Handlers

In order to register the JAX-RPC server-side context handler with the deployed web services, a handler chain must be included in the web services deployment descriptor. Please refer to the demo application `ddrpc/jboss/webservices.xml` deployment descriptor for an example of how this can be achieved.

JAX-WS Service Context Handlers

In order to register the JAX-WS server-side context handler with the deployed web services, a handler chain must be installed on the Server Endpoint Implementation class. The `javax.jws.WebService` annotation attached to the endpoint implementation class identifies a handler configuration file deployed with the application. Please refer to the demo application configuration file `dd/jboss/context-handlers.xml` and the endpoint implementation classes in `src/com/jboss/jbosstm/xts/demo/services` for an example of how this can be achieved.

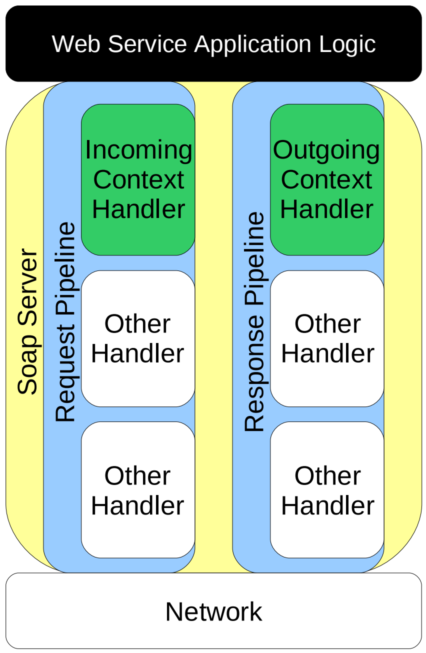

Figure 15 Context Handlers Registered with the SOAP Server

Implementing the Web Service Business Logic

The detail of the context management that the context processor performs is normally unimportant to the Web service application logic, and is orthogonal to any work performed by any other protocol-specific context handlers too. However back-end systems which the Web service application logic uses (such as databases) are often interested in the front-end transaction context such that any operations invoked within its scope can be mapped onto a back-end transaction context. This is typically achieved at the back-end by wrapping a database driver in a veneer which implements both the interface of the original driver and hooks into the service-side API to access the transaction context details. The general architecture for this pattern is shown in Figure 16.

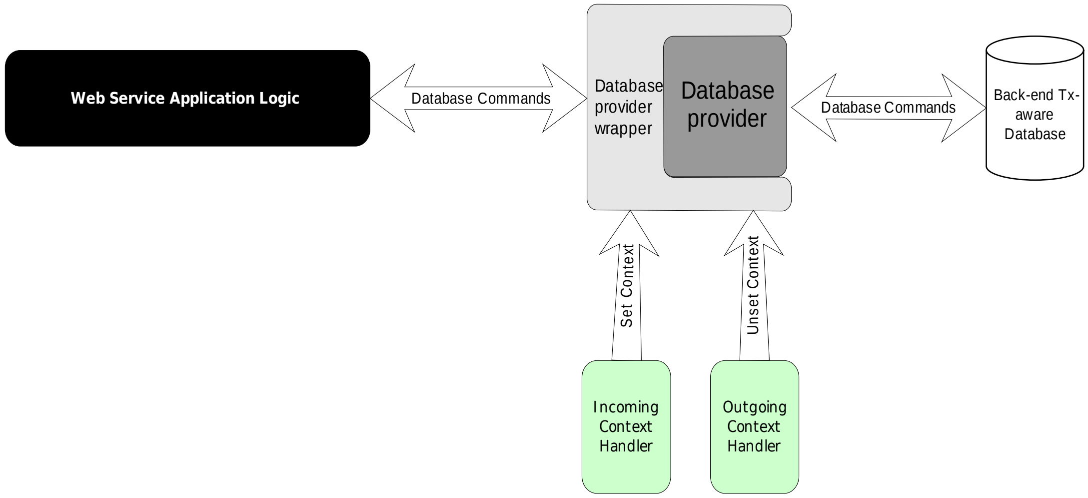

Figure 16 General Pattern for Back-End Integration, Service Side

The missing element from this is the commit protocol which finally allows back-end work to be made durable or not at the end of a transaction. This is covered in the participant chapter where the participant/back-end relation is explored further.

Summary

This chapter has provided a high-level overview of how to use each of the major software pieces of the Web Services transactions component of JBossTS. The Web Services transaction manager provided by JBossTS is the hub of the architecture and is the only piece of software that users' software does not bind to directly. XTS provides header processing infrastructure for dealing with Web Services transactions contexts for both users' client applications and Web services. For developing transaction participants, XTS provides a simple interface plus the necessary document handling code.

This chapter is meant as an overview only, and is unlikely to answer questions on more difficult and subtle aspects. For fuller explanations of the components, please refer to the appropriate chapter elsewhere in this document.

Participants

The Participant: an Overview

The participant is the entity that performs the work pertaining to transaction management on behalf of the business services involved in an application. The Web service (e.g. a theater booking system) contains some business logic for reserving a seat, inquiring availability etc, but it will need to be supported by something that maintains information in a durable manner. Typically this will be a database, but it could be a file system, NVRAM, etc. Now, although the service may talk to the back-end database directly, it cannot commit or undo any changes it (the services) makes, since these are ultimately under the control of the transaction that scoped the work. In order for the transaction to be able to exercise this control, it must have some contact with the database. In XTS this is accomplished by the participant, and the role played by the participant between the transaction and back-end transaction processing infrastructure is shown in Figure 17.

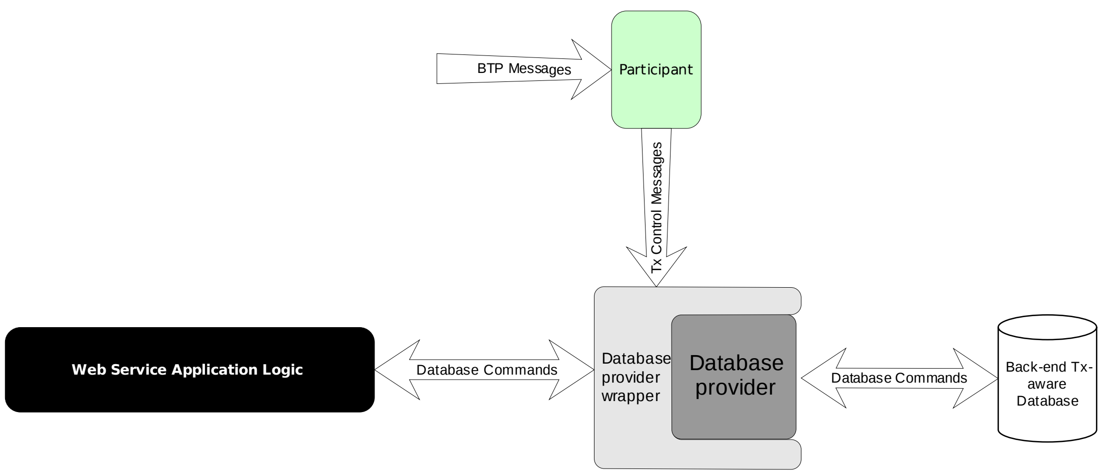

Figure 17 Transactions, Participants, and Back-End Transaction Control

Each participant in XTS is related to either the Atomic Transaction or Business Activity protocols. In the following sections we'll consider both protocols and their respective participants.

Atomic Transaction

All Atomic Transaction participants are instances of one of the following interfaces.

Durable2PCParticipant

This participant supports the WS-Atomic Transaction Durable2PC protocol with the following signatures, as per the `com.arjuna.wst.Durable2Participant` interface:

- *prepare*: the participant should perform any work necessary to allow it to either commit or rollback the work performed by the Web service under the scope of the transaction. The implementation is free to do whatever it needs to in order to fulfill the implicit contract between it and the coordinator. The participant is expected to indicate whether it can prepare or not by returning an instance of the `com.arjuna.wst.Vote`. Values are: `ReadOnly`, indicating the participant does not need to be informed of the transaction outcome as no state updates were made; `Prepared`, indicating the participant is prepared to commit or rollback depending on the final transaction outcome, and it has made sufficient state updates persistent to accomplish this; and `Aborted`, indicating the participant has aborted and the transaction should also attempt to do so.
- *commit*: the participant should make permanent the work that it controls. What it does will depend upon its implementation, e.g., commit the reservation of the theatre ticket. In the unlikely event that commit processing cannot complete the participant should throw a `SystemException`, potentially leading to a heuristic outcome for the transaction.
- *rollback*: the participant should undo the work that it controls. In the unlikely event that rollback processing cannot complete the participant should throw a `SystemException`, potentially leading to a heuristic outcome for the transaction.
- *unknown*: during recovery the participant can inquire as to the status of the transaction it was registered with. If that transaction is no longer available (has rolled back) then this operation will be invoked by the coordination service.
- *error*: during recovery the participant can inquire as to the status of the transaction it was registered with. If an error occurs (e.g., the transaction service is unavailable) then this operation will be invoked.

Volatile2PCParticipant

This participant supports the WS-Atomic Transaction Volatile2PC protocol with the following signatures, as per the `com.arjuna.wst.Volatile2Participant` interface:

- *prepare*: the participant should perform any work necessary to flush to persistent store any volatile data created by the Web service under the scope of the transaction. The implementation is free to do whatever it needs to in order to fulfill the implicit contract between it and the coordinator. The participant is expected to indicate whether it can prepare or not by returning an instance of the `com.arjuna.wst.Vote`. Values are: `ReadOnly`, indicating the participant does not need to be informed of the transaction outcome; `Prepared`, indicating the participant wishes to be notified of the final transaction outcome via a call to `commit` or `rollback`; and `Aborted`, indicating the participant has aborted and the transaction should also attempt to do so.
- *commit*: the participant should perform any cleanup activities required in response to a successful transaction commit. What it does will depend upon its implementation, e.g., it may decide to flush cached backup copies of data modified during the transaction. In the unlikely event that commit processing cannot complete the participant should throw a

`SystemException`. This will not affect the outcome of the transaction but will cause an error to be logged. Note that this method may not be called if a crash occurs during commit processing.

- *rollback*: the participant should perform any cleanup activities required in response to a transaction abort. In the unlikely event that rollback processing cannot complete the participant should throw a `SystemException`. This will not affect the outcome of the transaction but will cause an error to be logged. Note that this method may not be called if a crash occurs during commit processing.
- *unknown*: this method should never be called since volatile participants are not involved in recovery processing.
- *error*: this method should never be called since volatile participants are not involved in recovery processing.

Business Activity

All Business Activity participants are instances of the following interfaces.

BusinessAgreementWithParticipantCompletion

This participant supports the WS-T BusinessAgreementWithParticipantCompletion protocol with the following signatures, as per the `com.arjuna.wst.BusinessAgreementWithParticipantCompletionParticipant` interface:

- *close*: the transaction has completed successfully. The participant previously informed the coordinator that it was ready to complete.
- *cancel*: the transaction has canceled, and the participant should undo any work. The participant cannot have informed the coordinator that it has completed.
- *compensate*: the transaction has canceled. The participant previously informed the coordinator that it had finished work but could compensate later if required, so it is now requested to do so. In the unlikely event that commit processing cannot complete the participant should throw a `FaultedException`, potentially leading to a heuristic outcome for the transaction. If commit processing cannot complete because of a transient condition then the participant should throw a `SystemException`, in which case the compensation action may be retried or the transaction may finish with a heuristic outcome.
- *status*: return the status of the participant.
- *unknown*: if the participant inquires as to the status of the transaction it was registered with and that transaction is no longer available (has rolled back) then this operation will be invoked by the coordination service.
- *error*: if the participant inquired as to the status of the transaction it was registered with and an error occurs (e.g., the transaction service is unavailable) then this operation will be invoked.

BusinessAgreementWithCoordinatorCompletion

This participant supports the WS-T BusinessAgreementWithCoordinatorCompletion protocol with the following signatures, as per the `com.arjuna.wst.BusinessAgreementWithCoordinatorCompletionParticipant` interface:

- *close*: the transaction has completed successfully. The participant previously informed the coordinator that it was ready to complete.
- *cancel*: the transaction has canceled, and the participant should undo any work.
- *compensate*: the transaction has canceled. The participant previously informed the coordinator that it had finished work but could compensate later if required, so it is now requested to do so. In the unlikely event that commit processing cannot complete the participant should throw a `FaultedException`, potentially leading to a heuristic outcome for the transaction. If commit processing cannot complete because of a transient condition then the participant should throw a `SystemException`, in which case the compensation action may be retried or the transaction may finish with a heuristic outcome.
- *complete*: the coordinator is informing the participant that all work it needs to do within the scope of this business activity has been received.
- *status*: return the status of the participant.
- *unknown*: if the participant inquires as to the status of the transaction it was registered with and that transaction is no longer available (has rolled back) then this operation will be invoked by the coordination service.
- *error*: if the participant inquired as to the status of the transaction it was registered with and an error occurs (e.g., the transaction service is unavailable) then this operation will be invoked.

BAParticipantManager

In order for the Business Activity protocol to work correctly, the participants must be able to autonomously signal the coordinator that they have left the activity (exited) or are ready to complete and (if necessary) compensate (completed). Unlike the Atomic Transaction protocol, where all interactions between the coordinator and participants are instigated by the coordinator when the transaction terminates, this interaction pattern requires the participant to be able to talk to the coordinator at any time during the lifetime of the business activity.

As such, whenever a participant is registered with a business activity, it receives a handle on the coordinator. This handle is an instance of the `BAParticipantManager` interface, located in `com.arjuna.wst.BAParticipantManager`, with the following methods:

- *exit*: the participant has exited the business activity. The participant uses this to inform the coordinator that is has left the activity. It will not be informed when (and how) the business activity terminates.
- *completed*: the participant has completed it works, but wishes to continue in the business activity, so that it will eventually be told when (and how) the activity terminates. The participant may later be asked to compensate for the work it has done.

- *fault*: the participant encountered an error during normal activation and has compensated. This places the business activity into a mandatory cancel-only mode.

Note: The 1.1 XTS implementation provides class BAParticipantManager in package com.arjuna.wst11 whose API differs slightly in accordance with the differences in the 1.1 specification: method *fault* is renamed *fail*; method *cannotComplete* is provided to allow a participant to indicate that it is not able to complete the activity.

Participant Creation and Deployment

As has been shown, the participant provides the plumbing that drives the transactional aspects of the service. This section discusses the specifics of Participant programming and usage.

Implementing Participants

Implementing a participant is, in theory, a relatively straightforward task, though depending on the complexity of the transactional infrastructure that the participant is to manage, the actual size and complexity of a participant will vary. The participant interfaces can be found under `com.arjuna.wst`. Your implementation must implement one of these interfaces.

Deploying Participants

In order to allow Participants to be located remote from the Transaction Manager, XTS includes transparent message routing functionality. The Participant classes are not exposed directly as web services, but rather registered with a web service which receives messages from the Transaction Manager and maps them to appropriate method invocations on the relevant Participant instance. Transactional web services will typically enroll a new Participant instance of the desired type for each new transaction. A unique identifier must be provided at enrollment time and will be used to map transaction protocol messages to the appropriate participant instance. Note that Participant method invocations do not occur in a specific transaction context. Therefore, if your Participant implementation requires knowledge of the transaction context (e.g. to look up state information in a persistent store) then you should supply this to the Participant instance, typically as an argument to the constructor function. Sample Participant implementations and usage can be found in the demonstration applications included with XTS.

An application code which creates and enrolls Participants must be deployed along with the XTS services and libraries necessary for receiving and processing incoming messages from the Transaction Manager. When using the JBoss Application Server this merely involves dropping the XTS service archive (sar) into the server deploy directory. If you configure the application to use a coordinator located in some other container the client application will only exercise the participant components of your deployment. If you configure you client application to use a local coordinator then both the participant and transaction manager components will be exercised.

Note: In previous releases XTS applications could only be deployed if the appropriate XTS and Transaction Manager .jar, .war and configuration files were bundled with the application. Although this is still feasible as a way of deploying XTS applications to other containers it is not supported in this release.

The XTS API

Introduction

This chapter shows how to use the XTS API. This is of use both at the client-side where applications consume transactional Web services, and at the service/participant side where transactions need to be coordinated with back-end systems.

API for the Atomic Transaction protocol

Vote

During the two-phase commit protocol, a participant will be asked to vote on whether or not it can prepare to confirm the work that it controls. It must return one of the following subtypes of `com.arjuna.wst.Vote`:

- `Prepared`: the participant indicates that it can prepare if asked to by the coordinator. It will not have committed at this stage however, since it does not know what the final outcome will be.
- `Aborted`: the participant indicates that it cannot prepare and has in fact rolled back. The participant should not expect to get a second phase message.
- `ReadOnly`: the participant indicates that the work it controls has not made any changes to state that require it to be informed of the final outcome of the transaction. Essentially the participant is resigning from the transaction.

Thus a possible implementation of a 2PC participant's `prepare` method may resemble the following:

```
public Vote prepare () throws WrongStateException, SystemException
{
  // Some participant logic here

  if(/* some condition based on the outcome of the business logic */)
  {
    // Vote to confirm
    return new com.arjuna.wst.Prepared();
  }
  else if(/*another condition based on the outcome of the business logic*/)
  {
```

```
      // Resign
      return new com.arjuna.wst.ReadOnly();
    }
    else
    {
      // Vote to cancel
      return new com.arjuna.wst.Aborted();
    }
  }
```

Figure 18 API Example Showing Participant Voting

TXContext

A transaction is typically represented by some unique identifier and a reference to the coordinator which manages the transaction, e.g. a URL. XTS allows transactions to be nested such that a transaction hierarchy (tree) may be formed. Thus, a transaction context may be a set of transactions, with the top-most transaction the root parent (superior).

Note: The current XTS 1.0 and 1.1 implementations do not support nested transactions.

`com.arjuna.mw.wst.TxContext` is an opaque representation of a transaction context.

- valid: this indicates whether or not the contents are valid.
- equals: can be used to compare two instances.

Note: The 1.0 specification implementation locates the following four classes in the `com.arjuna.mw.wst` package. The 1.1 specification implementation locates them in the `com.arjuna.mw.wst11` package.

UserTransaction

`com.arjuna.mw.wst.UserTransaction` is the class that most users (e.g., clients or services) will employ. In particular, a client wishing to begin a new atomic transaction must first obtain a `UserTransaction` from the `UserTransactionFactory`. This class isolates the user from the underlying protocol-specific aspects of the XTS implementation. Importantly, a `UserTransaction` does not represent a specific transaction, but rather is responsible for providing access to an implicit per-thread transaction context; it is similar to the `UserTransaction` in the JTA specification. Therefore, all of the `UserTransaction` methods implicitly act on the current thread of control.

A new transaction is begun and associated with the invoking thread by using the `begin` method. If there is already a transaction associated with the thread then the `WrongStateException` exception is thrown. Upon success, this operation associates the newly created transaction with the current thread. An optional timeout parameter (measured in milliseconds) may be supplied as argument to the `begin` call.

The transaction is committed by the `commit` method. This will execute the Volatile2PC and Durable2PC protocols prior to returning. If there is no transaction associated with the invoking thread then `UnknownTransactionException` is thrown. If the transaction ultimately rolls back then the `TransactionRolledBackException` is thrown. When complete, this operation disassociates the transaction from the current thread such that it becomes associated with no transaction.

The `rollback` operation will terminate the transaction and return normally if it succeeded, while throwing an appropriate exception if it didn't. If there is no transaction associated with the invoking thread then `UnknownTransactionException` is thrown. When complete, this operation disassociates the transaction from the current thread such that it becomes associated with no transaction

UserTransactionFactory

A `UserTransaction` instance is obtained from a `UserTransactionFactory` by calling the `getUserTransaction` method.

TransactionManager

The `TransactionManager` interface defines how a service/container/participant (service-side user) typically will interact with the underlying transaction service implementation. As with `UserTransaction` a `TransactionManager` does not represent a specific transaction, but rather is responsible for providing access to an implicit per-thread transaction context.

The `currentTransaction` method returns a `TxContext` for the current transaction, or null if there is none. It can be used to identify whether a web service has been invoked from within an existing transaction. The returned value can also be used to enable multiple threads to execute within the scope of the same transaction. Calling this method does not disassociate the current thread from the transaction.

A thread of control may require periods of non-transactionality so that it may perform work that is not associated with a specific transaction. In order to do this it is necessary to disassociate the thread from any transactions. The `suspend` method accomplishes this, returning a `TxContext` instance, which is a handle on the transaction. The thread is then no longer associated with any transaction.

The `resume` method can be used to associate or re-associate a thread with a transaction via its `TxContext`. Prior to (re-)association, the thread is disassociated from any transaction with which it may be currently associated. If the `TxContext` is null, then the thread is associated with no transaction i.e. the result is the same as if `suspend` had been called. The `UnknownTransactionException` exception is thrown if the transaction that the `TxContext` refers to is invalid in the scope of the invoking thread.

In order to register and resign participants with a transaction, the container or participant must use:

- `enlistForVolatileTwoPhase`: enlist the specified participant with current transaction such that it will participate in the Volatile2PC protocol; a unique identifier for the participant is also required. If there is no transaction associated with the invoking thread then the `UnknownTransactionException` exception is thrown. If the coordinator already has a participant enrolled with the same identifier, then `AlreadyRegisteredException` will be thrown. If the transaction is not in a state where participants can be enrolled (e.g., it is terminating) then `WrongStateException` will be thrown.

- `enlistForDurableTwoPhase`: enlist the specified participant with current transaction such that it will participate in the 2PC protocol; a unique identifier for the participant is also required. If there is no transaction associated with the invoking thread then the `UnknownTransactionException` exception is thrown. If the coordinator already has a participant enrolled with the same identifier, then `AlreadyRegisteredException` will be thrown. If the transaction is not in a state where participants can be enrolled (e.g., it is terminating) then `WrongStateException` will be thrown.

TransactionManagerFactory

A `TransactionManager` instance is obtained from a `TransactionManagerFactory` by calling the `getTransactionManager` method.

API for the Business Activity protocol

> **Note:** The 1.0 specification implementation locates the following four classes in the `com.arjuna.mw.wst` package. The 1.1 specification implementation locates them in the `com.arjuna.mw.wst11` package.

UserBusinessActivity

`com.arjuna.wst.UserBusinessActivity` is the class that most users (e.g., clients and services) will employ. In particular, a client wishing to begin a new atomic transaction must first obtain a `UserBusinessActivity` from the `UserBusinessActivityFactory`. This class isolates them from the underlying protocol-specific aspects of the XTS implementation. Importantly, a `UserBusinessActivity` does not represent a specific business activity, but rather is responsible for providing access to an implicit per-thread activity. Therefore, all of the `UserBusinessActivity` methods implicitly act on the current thread of control.

A new business activity is begun and associated with the invoking thread by using the `begin` method. If there is already a business activity associated with the thread then the `WrongStateException` exception is thrown. Upon success, this operation associates the newly created activity with the current thread. An optional timeout parameter (measured in milliseconds) may be supplied as argument to the `begin` call.

The business activity is terminated successfully by the `close` method. This will execute the BusinessAgreementWithParticipantCompletion protocol prior to returning. If there is no activity associated with the invoking thread then `UnknownTransactionException` is thrown. If the activity ultimately cancels then the `TransactionRolledBackException` is thrown. When complete, this operation disassociates the business activity from the current thread such that it becomes associated with no activity.

The `cancel` operation will terminate the business activity and return normally if it succeeded, while throwing an appropriate exception if it didn't. If there is no activity associated with the invoking thread then `UnknownTransactionException` is thrown. Any participants that had previously completed will be informed to compensate for their work.

In some cases participants in a business activity may register for the BusinessAgreementWithCoordinatorCompletion protocol. This means that they require notification from the coordinator that all the work that they need to do within the scope of a business activity has been completed. In such cases, once the the client application is sure that it will not request further work to be done by these participants it must call the `complete` method before attempting either to `close` or `cancel` the activity.

UserBusinessActivityFactory

A `UserBusinessActivity` instance is obtained from a `UserBusinessActivityFactory` by calling the `getUserBusinessActivity` method.

BusinessActivityManager

The `BusinessActivityManager` interface defines how a service/container/participant's (service-side user) typically will interact with the underlying business activity service implementation. As with `UserBusinessActivity` a `BusinessActivityManager` does not represent a specific activity, but rather is responsible for providing access to an implicit per-thread activity.

The `currentTransaction` method returns the `TxContext` for the current business activity, or null if there is none. It can be used to identify whether a web service has been invoked from within an existing business activity. The returned value can also be used to enable multiple threads to execute within the scope of the same business activity. Calling this method does not disassociate the current thread from the activity.

A thread of control may require periods of non-transactionality so that it may perform work that is not associated with a specific activity. In order to do this it is necessary to disassociate the thread from any current business activity. The `suspend` method accomplishes this, returning a `TxContext` instance, which is a handle on the activity. The thread is then no longer associated with any activity.

The `resume` method can be used to associate or re-associate a thread with a business activity via its `TxContext`. Prior to (re-)association, the thread is disassociated from any business activity with which it may be currently associated. If the `TxContext` is null, then the thread is associated with no activity i.e. the result is the same as if `suspend` had been called. The `UnknownTransactionException` exception is thrown if the business activity that the `TxContext` refers to is invalid in the scope of the invoking thread.

In order to register and resign participants with a business activity, the container or participant must use:

- enlistForBusinessAgreementWithParticipantCompletion: enlist the specified participant with current business activity such that it will participate in the BusinessAgreementWithParticipantCompletion protocol; a unique identifier for the participant is also required. If there is no business activity associated with the invoking thread then the `UnknownTransactionException` exception is thrown. If the coordinator already has a participant enrolled with the same identifier, then `AlreadyRegisteredException` will be thrown. If the activity is not in a state where participants can be enrolled (e.g., it is terminating) then `WrongStateException` will be thrown.

- enlistForBusinessAgreementWithCoordinatorCompletion: enlist the specified participant with current activity such that it will participate in the BusinessAgreementWithCoordinatorCompletion protocol; a unique identifier for the participant is also required. If there is no business activity associated with the invoking thread then the `UnknownTransactionException` exception is thrown. If the coordinator already has a participant enrolled with the same identifier, then `AlreadyRegisteredException` will be thrown. If the activity is not in a state where participants can be enrolled (e.g., it is terminating) then `WrongStateException` will be thrown.

BusinessActivityManagerFactory

A `BusinessActivityManager` instance is obtained from a `BusinessActivityManagerFactory` by calling the `getBusinessActivityManager` method.

Stand-alone coordination

Introduction

The XTS service is deployed as a JBoss service archive (sar). The version of the service archive provided with this JBossTS release implements both the 1.0 and 1.1 versions of the WS-C, WS-AT and WS-BA services running side by side. Depending upon which API clients and transactional services are compiled with, the relevant WS servers are contacted and used to coordinate the transaction. It is possible to rebuild the XTS service archive to only start up one version of these services. See the service archive build script for for further details.

The release service archive is configured to obtain coordination contexts from the Activation Coordinator service running on the deployed host. This means that WS-AT transactions or WS-BA activities created by a locally deployed client application will be supplied with a context which identifies the Registration Service running on the client's machine. Hence, any web services invoked by the client will be coordinated by the Transaction Protocol services running on the client's host. This will be true whether the web services are running locally or on some other remote host. This configuration is known as *local* coordination.

It is possible to reconfigure this setting globally for all clients, causing context creation requests to be redirected to an Activation Coordinator Service running on some other, remote host. This normally implies that the rest of the coordination process is executed from the remote host and is referred to as *stand-alone* coordination.

There are various reasons for choosing to use a stand-alone coordinator. One reason might be efficiency: if a client application invokes web services on a remote JBoss AS server it might be more efficient to coordinate the transaction from the remote server since the protocol-specific messages between the coordinator and the participants would not have to travel over the network. Another reason might be reliability: if the coordinator service runs on its own dedicated host then there is no danger of failing applications or services bringing down the coordinator and causing failures for unrelated transactions. A third reason might be to use a coordination service provided by a third party vendor.

Configuring The Activation Coordinator

The simplest way to configure a stand-alone coordinator is to provide a command line switch when starting JBossAS using the -D option to specify a setting for a System property. There are several configuration options which are are obeyed in the following order of priority:

Absolute URL

An absolute URL for the Activation Coordinator service may be supplied by defining the following properties:

XTS 1.0

`org.jboss.jbossts.xts.coordinatorURL`

[http://*coord.host*:*coord.port*/ws-c10/soap/ActivationCoordinator]

XTS 1.1

`org.jboss.jbossts.xts11.coordinatorURL`

[http://*coord.host*:*coord.port*/ws-c11/ActivationService]

The value assigned to these URLs will depend upon the configuration of the remote coordinator host. The sample values written underneath the property names are appropriate for the case where the coordinator is another JBoss XTS service. Note that the italicised terms `coord.host` and `coord.port` should be substituted with the host name and port number of the JBoss instance running the Activation Coordinator service.

Coordinator Host, Port and Path

The separate components of the Coordinator URL may be independently specified by defining the following properties:

XTS 1.0

`org.jboss.jbossts.xts.coordinator.host`

[*server.bind.address*]

`org.jboss.jbossts.xts.coordinator.port`

[*jboss.web.bind.port*]

`org.jboss.jbossts.xts.coordinator.path`

[ws-c10/soap/ActivationCoordinator]

XTS 1.1

`org.jboss.jbossts.xts11.coordinator.host`

[*server.bind.address*]

`org.jboss.jbossts.xts11.coordinator.port`

[*jboss.web.bind.port*]

`org.jboss.jbossts.xts11.coordinator.path`

[ws-c11/ActivationService]

If any of these three components is set then the coordinator URL will be constructed using whichever of the component values is defined and substituting the defaults values specified below the property names for any undefined components. The italicised values *`server.bind.address`* and *`jboss.web.bind.port`* represent the server bind address and the web service listener port obtained either from the application server command line or the server configuration files.

Build Time Default Coordinator Configuration

It is possible to reset the default coordinator host name and port used by the XTS service archive by defining the properties `coordinator.hostname` and `coordinator`.port on the build command line when building the service archive. See the service archive build script in the sar directory of the XTS source code for further details.

It is also possible to redefine the default absolute URL employed by the XTS service by modifying the values specified in the WSTX configuration files included in the service archive. The 1.0 specification URL is defined in file wstx.xml and the 1.1 specification URL is defined in file wstx11.xml. The values can be redefined either by editing the service archive supplied in the sar directory of the XTS install tree or by modifying the source files located in the config directory of the XTS source tree.

Note: The XTS sar must be deployed on hosts running client applications and participant Web services as well as on the stand-alone coordinator host. Obviously, clients and participants need access to the libraries deployed with XTS and these are bundled into the sar. However, they also need to be able to receive transaction management messages dispatched from the coordinator host. The XTS sar provides implementations of client-side and participant-side listeners for this purpose. Although these services could be deployed separately from the coordinator-side service listeners and implementation they are all bundled into a single sar for simplicity.

Chapter 8

Participant crash recovery

Introduction

A transaction service is of limited use if it is not resilient to a system crash on the the participant host(s) or the host running the transaction coordination services. Crashes which happen before a transaction terminates or before a business activity has completed are relatively easy to deal with. The transaction service and participants can adopt a presumed abort policy. If the coordinator crashes then it can assume any transaction it does not know about is invalid and reject a participant request which mentions such a transaction. If the participant crashes then it can forget any provisional changes it has made and reject any request from the coordinator service to prepare a transaction or complete a business activity.

When a crash happens during a transaction commit operation or between completing and closing a business activity then things are more complicated. The transaction service must ensure as far as possible that participants arrive at a consistent outcome for the transaction. For a WS-AT transaction this means committing all provisional changes or rolling them all back to the state before the transaction was started. For a business activity this means getting all participants to close the activity or requiring them all to cancel the activity and run any required compensating actions. On the rare occasions where such a consensus cannot be arrived at the transaction service must guarantee to log and report transaction failures.

XTS currently includes support for automatic recovery of WS-AT transactions should the coordinator service crash. It also currently supports crash recovery for WS-AT transactions when a participant host crashes. Crash recovery for WS-BA participants is not currently implemented but will be provided in a future release of XTS.

> **Note:** Crash recovery is implemented for both the 1.0 and 1.1 WS-AT protocol implementations

WS-AT Coordinator Crash Recovery

The WS-AT coordination service implementation tracks the status of each participant in a transaction as the transaction progresses through its 2 phase commit. When all participants have been sent a prepare message and responded with a prepared message the coordinator writes a log record storing the details of each participant to indicate that the transaction is ready to complete. If the coordinator service crashes after this point has been reached it can still guarantee to complete the 2 phase commit. It merely has to check the log at reboot and send a commit message to all participants. Once all participants have responded to the commit with a committed message it can safely delete the log entry.

This is safe since the prepared messages returned by the participants imply that they are all able to roll forward their provisional changes and make them permanent. Also, the coordinator does not need to worry about any commit messages which may have been sent before the crash nor even care about resending messages if it crashes several times: the XTS participant implementation is resilient to redelivery of the commit messages. Assuming that the participant has implemented the recovery functions described below the coordinator can even guarantee to deliver commit messages if both it and one (or more) of the participant service hosts should simultaneously crash.

On the other hand, if the coordination service crashes before the prepare phase completes then the presumed abort protocol ensures that participants are rolled back. After restart the coordination service knows about all the transactions which could have entered the commit phase before the reboot because they will have an entry in the log. It also knows about any active transactions started after the reboot. If a participant was left waiting after sending its prepared message the XTS participant implementation will automatically resend the prepared message at regular intervals. When the coordinator detects a transaction which is not active and was not found in the log during reboot then it tells the participant to abort, ensuring that the web service gets a chance to roll back any provisional state changes it had made on behalf of the transaction.

There is a danger that a web service may decide to unilaterally roll forward or roll back provisional changes associated with a given participant because it decides it has waited too long for an outcome from the coordinator. For example, it might do this in order to release a resource lock. If so then the the web service should record this action and log a message to persistent storage. When the participant is eventually requested to commit or rollback it should throw an exception if its unilateral decision action does not match the requested action. The coordinator will detect the exception and log a message detailing the outcome as *heuristic*. It will also save the state of the transaction permanently in the transaction log where it can be inspected to identify the outcomes associated with all participants.

WS-AT Participant Crash Recovery

WS-AT participants associated with a transactional web service do not need to be involved in crash recovery if the Web service's host machine crashes before the participant is told to prepare. The coordinator will merely assume that the transaction has aborted. So, the Web service can discard any information associated with unprepared transactions when it reboots.

When a participant is told to prepare the Web service is expected to save to persistent storage the transactional state it needs either to roll forward or roll back the transaction. The details of what information it needs to save and how it organizes and manages the alternative versions of this information are specific to the business logic of the Web service. However, the participant must save this state before returning a `Prepared` vote from the prepare call. If it cannot save the required state or there is some other problem servicing the request made by the client it must return an `Aborted` vote.

The XTS participant services running on a Web service's host machine co-operate with the Web service implementation to implement participant crash recovery. These participant services are responsible for calling the participant's `prepare`, `commit` and `rollback` methods. The XTS implementation tracks the local state of every enlisted participant. If the `prepare` call returns a Prepared vote the XTS implementation ensures that the participant state is logged to the local transaction log before forwarding a prepared message to the coordinator.

A participant log record contains information identifying the participant, its transaction and its coordinator. This is enough to allow the rebooted XTS implementation to reinstate the participant as active and to continue communication with the coordinator, as though the participant had been enlisted and driven to the prepared state. However, if the log only contained this information at reboot there would still be one thing missing. The commit or rollback process cannot continue without a participant instance.

Full recovery requires the log record to contain information needed by the Web service which enlisted the participant. This information must allow it to recreate an equivalent participant which can continue the commit process to completion or rollback if some other Web service fails to prepare. This might be as simple as a String key which allows it to locate the data it made persistent before returning its `Prepared` vote. Or it may be as complex as a serialized object tree containing the original participant instance and, possibly, other objects created by the Web service.

If a participant instance implements the relevant interface the XTS implementation will append this *participant recovery state* to its log record before writing it to persistent storage. If a crash happens the participant recovery state is retrieved from the log and passed to the Web service which created it. The Web service must use this state to create a new participant which the XTS implementation will use to drive the transaction to completion. Log records are only deleted once the participant's commit or rollback method has been called. Note that if a crash happens just before or just after a commit method has been called it is possible that a commit or rollback method may be called twice.

WS-AT Participant Crash Recovery APIs

Saving Participant Recovery State

A participant may signal that it is capable of performing recovery processing simply by implementing interface `java.lang.Serializable`. Alternatively it may implement the following interface:

```
public interface PersistableATParticipant
{
  byte[] getRecoveryState() throws Exception;
}
```

If a participant implements interface `Serializable` then the XTS participant services implementation will use the serialization API to create a version of the participant which can be appended to the participant log entry. If it implements interface PersistableATParticipant then the XTS participant services implementation will call method `getRecoveryState` to obtain the state to be appended to the participant log entry.

If neither of these APIs is implemented then the XTS implementation will log a warning message and proceed without saving any recovery state. This means that in the event that the Web service's host machine crashes while the transaction is being committed the transaction will not be recovered and may well have a heuristic outcome logged on the host running the coordinator services.

Recovering Participants at Reboot

A Web service must register with the XTS implementation when it is deployed and unregister when it is undeployed in order for it to be able to take part in recovery processing. Registration is performed using class `XTSATRecoveryManager` defined in package `org.jboss.jbossts.xts.recovery.participant.at`:

```
public abstract class XTSATRecoveryManager {
    . . .
    public static XTSATRecoveryManager getRecoveryManager() ;
    public void registerRecoveryModule(XTSATRecoveryModule module);
    public abstract void unregisterRecoveryModule(XTSATRecoveryModule module)
        throws NoSuchElementException;
    . . .
}
```

The Web service must provide an implementation of interface `XTSATRecoveryModule` in package `org.jboss.jbossts.xts.recovery.participant.at` as argument to the regsiter and unregister calls. This instance is responsible for identifying saved participant recovery records and recreating new, recovered participant instances:

```
public interface XTSATRecoveryModule
{
    public Durable2PCParticipant
    deserialize(String id, ObjectInputStream stream) throws Exception;
    public Durable2PCParticipant
    recreate(String id, byte[] recoveryState) throws Exception;
}
```

If a participant's recovery state was saved using serialization then the recovery module's deserialize method will be called to allow it to recreate the participant. Normally, all this requires is for the recovery module to read, cast and return an object from the supplied input stream. If a participant's recovery state was saved using the `PersistableATParticipant` interface then the recovery module's recreate method will be called to allow it to recreate the participant from the byte array it provided at the point where the state was saved.

The XTS implementation does not know which participants belong to which recovery modules. A module is only expected to return a participant instance if it can identify that the recovery state belongs to its Web service. If the participant was created by some other Web service then it should return null. The participant id supplied as argument to the `deserialize` or `recreate` calls is the id which was used by the Web service when the original participant was enlisted in the transaction. Web services which wish to participate in recovery processing should ensure that the participant ids they employ contain a unique identifier string which will not be employed by any other services. If a module recognizes a participant id as belonging to its Web service but cannot recreate the participant it may throw an exception. For example this may be because the service cannot associate the participant with any business logic-specific transactional information.

A module must be registered by the application even when it relies upon serialization to create the participant recovery state saved by the XTS implementation. The deserialization operation must employ a class loader which is capable of loading Web service-specific classes. The XTS implementation achieves this by devolving responsibility for the deserialize operation to the recovery module.

Index