Transactions XTS Administration And Development Guide

Using the XTS Module of JBoss Transactions to provide Web Services Transactions

by Andrew Red Hat Dinn, Kevin Red Hat Connor, and Mark Red Hat Little

1. Document Conventions

This manual uses several conventions to highlight certain words and phrases and draw attention to specific pieces of information.

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts* [https://fedorahosted.org/liberation-fonts/] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

Four typographic conventions are used to call attention to specific words and phrases. These conventions, and the circumstances they apply to, are as follows.

`Mono-spaced Bold`

Used to highlight system input, including shell commands, file names and paths. Also used to highlight keycaps and key combinations. For example:

> To see the contents of the file `my_next_bestselling_novel` in your current working directory, enter the `cat my_next_bestselling_novel` command at the shell prompt and press **Enter** to execute the command.

The above includes a file name, a shell command and a keycap, all presented in mono-spaced bold and all distinguishable thanks to context.

Key combinations can be distinguished from keycaps by the hyphen connecting each part of a key combination. For example:

> Press **Enter** to execute the command.
>
> Press **Ctrl**+**Alt**+**F2** to switch to the first virtual terminal. Press **Ctrl**+**Alt**+**F1** to return to your X-Windows session.

The first paragraph highlights the particular keycap to press. The second highlights two key combinations (each a set of three keycaps with each set pressed simultaneously).

If source code is discussed, class names, methods, functions, variable names and returned values mentioned within a paragraph will be presented as above, in `mono-spaced bold`. For example:

> File-related classes include `filesystem` for file systems, `file` for files, and `dir` for directories. Each class has its own associated set of permissions.

Proportional Bold

This denotes words or phrases encountered on a system, including application names; dialog box text; labeled buttons; check-box and radio button labels; menu titles and sub-menu titles. For example:

Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click **Close** to switch the primary mouse button from the left to the right (making the mouse suitable for use in the left hand).

To insert a special character into a **gedit** file, choose **Applications** → **Accessories** → **Character Map** from the main menu bar. Next, choose **Search** → **Find…** from the **Character Map** menu bar, type the name of the character in the **Search** field and click **Next**. The character you sought will be highlighted in the **Character Table**. Double-click this highlighted character to place it in the **Text to copy** field and then click the **Copy** button. Now switch back to your document and choose **Edit** → **Paste** from the **gedit** menu bar.

The above text includes application names; system-wide menu names and items; application-specific menu names; and buttons and text found within a GUI interface, all presented in proportional bold and all distinguishable by context.

Mono-spaced Bold Italic or ***Proportional Bold Italic***

Whether mono-spaced bold or proportional bold, the addition of italics indicates replaceable or variable text. Italics denotes text you do not input literally or displayed text that changes depending on circumstance. For example:

To connect to a remote machine using ssh, type `ssh` *`username@domain.name`* at a shell prompt. If the remote machine is `example.com` and your username on that machine is john, type `ssh john@example.com`.

The `mount -o remount` *`file-system`* command remounts the named file system. For example, to remount the `/home` file system, the command is `mount -o remount /home`.

To see the version of a currently installed package, use the `rpm -q` *`package`* command. It will return a result as follows: *`package-version-release`*.

Note the words in bold italics above — username, domain.name, file-system, package, version and release. Each word is a placeholder, either for text you enter when issuing a command or for text displayed by the system.

Aside from standard usage for presenting the title of a work, italics denotes the first use of a new and important term. For example:

Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

Terminal output and source code listings are set off visually from the surrounding text.

Output sent to a terminal is set in `mono-spaced roman` and presented thus:

```
books        Desktop   documentation  drafts  mss    photos   stuff  svn
books_tests  Desktop1  downloads      images  notes  scripts  svgs
```

Source-code listings are also set in `mono-spaced roman` but add syntax highlighting as follows:

```
package org.jboss.book.jca.ex1;

import javax.naming.InitialContext;

public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      System.out.println("Created Echo");

      System.out.println("Echo.echo('Hello') = " + echo.echo("Hello"));
   }
}
```

1.3. Notes and Warnings

Finally, we use three visual styles to draw attention to information that might otherwise be overlooked.

Note

Notes are tips, shortcuts or alternative approaches to the task at hand. Ignoring a note should have no negative consequences, but you might miss out on a trick that makes your life easier.

Important

Important boxes detail things that are easily missed: configuration changes that only apply to the current session, or services that need restarting before an update will apply. Ignoring a box labeled 'Important' will not cause data loss but may cause irritation and frustration.

Warning

Warnings should not be ignored. Ignoring warnings will most likely cause data loss.

2. We Need Feedback!

You should over ride this by creating your own local Feedback.xml file.

About This Guide

The XTS Development Guide explains how to add resilience to distributed business processes based on web services, making them reliable in the event of system or network failures. It covers installation, administration, and development of transactional web services.

The JBoss Application Server implements Web Services Transactions standards using *XTS (XML Transaction Service)*. XTS supports development and deployment of transaction-aware web services. It also enables web service clients to create and manage web service transactions from which transactional web services can be invoked. XTS ensures that the client and web services achieve consistent outcomes even if the systems on which they are running crash or temporarily lose network connectivity.

XTS is compliant with the *WS-Coordination*, *WS-Atomic Transaction*, and *WS-Business Activity* specifications. The implementation supports web services and clients which are based on the JaxWS standard. XTS is itself implemented using services based on JaxWS. While this guide discusses many Web Services standards like SOAP and WSDL, it does not attempt to address all of their fundamental constructs. However, basic concepts are provided where necessary.

1.1. 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. It is also useful for system analysts and project managers who are unfamiliar with transactions as they pertain to Web services.

1.2. Prerequisites

JBoss Transaction Service uses the Java programming language and this manual assumes that you are familiar with programming in Java. Additional helpful skills are outlined in *Prerequisite Skills for XTS Developers*.

Prerequisite Skills for XTS Developers

- A Working knowledge of Web Services, including XML, SOAP, and WSDL
- A general understanding of transactions
- A general understanding of WS-Coordination, WS-Atomic Transaction and WS-Business Activity protocols

This guide presents overview information for all of the above. However, to aid in understanding the Web Services component of JBoss Transaction Service, the WS-C[1], WS-Atomic Transaction[2], and WS-Business Activity[3] specifications are discussed in great detail.

[1] *http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-Coordination.pdf*
[2] *http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-AtomicTransaction.pdf*
[3] *http://download.boulder.ibm.com/ibmdl/pub/software/dw/specs/ws-tx/WS-BusinessActivity.pdf*

Introduction

The *XML Transaction Service (XTS)* component of JBoss Transaction Service 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 something 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.

JBoss Transaction Service provides the XTS component as a transaction solution for Web Services. Using XTS, business partners can coordinate complex business transactions in a controlled and reliable manner. The XTS API supports a transactional coordination model based on the *WS-Coordination*, *WS-Atomic Transaction*, and *WS-Business Activity* specifications.

Protocols Included in XTS

- 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 implements versions 1.0, 1.1, and 1.2 of these three specifications. Version specifications are available from *http://www.oasis-open.org/specs/*.

Note

The 1.0, 1.1, and 1.2 specifications only differ in a small number of details. The rest of this document employs version 1.1 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 is 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 enable customers, suppliers, and trading partners to access Web Services, independent of hardware operating 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 the Web Services model is an attractive development solution are also the same reasons that service-based applications may have drawbacks.

Properties of Web Services

- 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, increasing risk of failure due to increased network travel for invocations.

Applications that have high dependability requirements need 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 a unit of work which is completed entirely, or in the case of failures is reversed to some agreed consistent state. The goal, in the event of a failure, is normally to appear as if the work had never occurred in the first place. With XTS, transactions can span multiple Web Services, meaning that work performed across multiple enterprises can be managed with transactional support.

2.1. 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. This is because 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. *An Evening On the Town* demonstrates how an application may manage service-based processes as transactions:

An Evening On the Town. 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.

Without the backing of a transaction, an undesirable outcome may occur. For example, the user credit card may be charged, even if one or both of the bookings fail.

An Evening On the Town 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.

2.2. Servlets

The WS-Coordination, 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. One group of entities, called transaction participants, invoke operations on other entities, such as the transaction coordinator, in order to return responses to requests. 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 XTS, the active component is achieved through deployment of JaxWS endpoints. Each XTS endpoint that is reachable through SOAP/XML is published via JaxWS, without developer intevention. The only requirement is that transactional client applications and transactional web services must reside within a domain capable of hosting JaxWS endpoints, such as an application server. JBoss Application Server can provide this functionality.

Note

The XTS 1.0 protocol implementation is based on servlets.

2.3. 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.

2.4. Web Services Description Language (WDSL)

Web Services Description Language (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.

Transactions Overview

Note

This chapter deals with the theory of transactional Web Services. If you are familiar with these principles, consider this chapter a reference.

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 JBoss Transaction Service 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 transaction is a unit of work that encapsulates multiple database actions such that that either all the encapsulated actions fail or all succeed.

Transactions ensure data integrity when an application interacts with multiple datasources.

The main components involved in using and defining transactional Web Services using XTS are illustrated in *Figure 3.1, "Components Involved in an XTS Transaction"*.

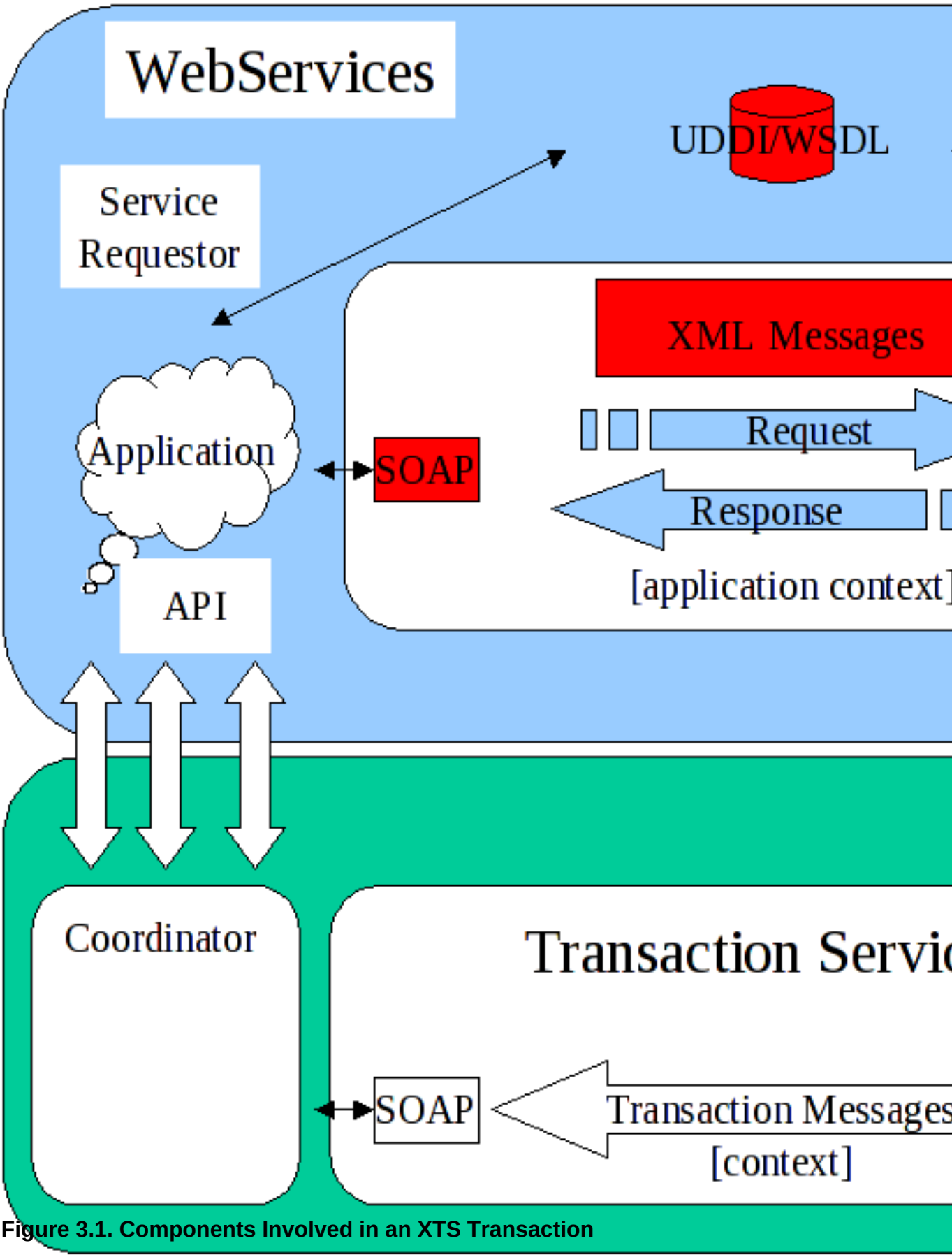

Figure 3.1. Components Involved in an XTS Transaction

3.1. The Coordinator

Every transaction is associated with a coordinator, which is responsible for governing the outcome of the transaction. When a client begins a Web Service transaction it posts a `create` request to a coordination service, which creates the coordinator and returns its details to the client. This service may be located in its own container or may be colocated with the application client or with one of the transactional web services for improved performance. The coordination service is typically responsible for managing many transactions in parallel, so each coordinator is identified by a unique transaction identifier.

The coordinator is responsible for ensuring that the web services invoked by the client arrive at a consistent outcome. When the client asks the coordinator to complete the transaction, the coordinator ensures that each web service is ready to confirm any provisional changes it has made within the scope of the transaction. It then asks them all to make their changes permanent. If any of the web services indicates a problem at the confirmation stage, the coordinator ensures that all web services reject their provisional changes, reverting to the state before the transaction started. The coordinator also reverts all changes if the client asks it to cancel the transaction.

The negotiation between the coordinator and the web services is organized to ensure that all services will make their changes permanent, or all of them will revert to the previous state, even if the coordinator or one of the web services crashes part of the way through the transaction."

3.2. The Transaction Context

In order for a transaction to span a number of services, certain information has to be shared between those services, to propagate information about the transaction. This information is known as the *Context*. The coordination service hands a context back to the application client when it begins a transaction. This context is passed as an extra, hidden parameter whenever the client invokes a transactional web service. The XTS implementation saves and propagates this context automatically with only minimal involvement required on the part of the client. However, it is still helpful to understand what information is captured in a context. This information is listed in *Contents of a Context*.

Contents of a Context

Transaction Identifier
: Guarantees global uniqueness for an individual transaction.

Transaction Coordinator Location
: The endpoint address participants contact to enroll.

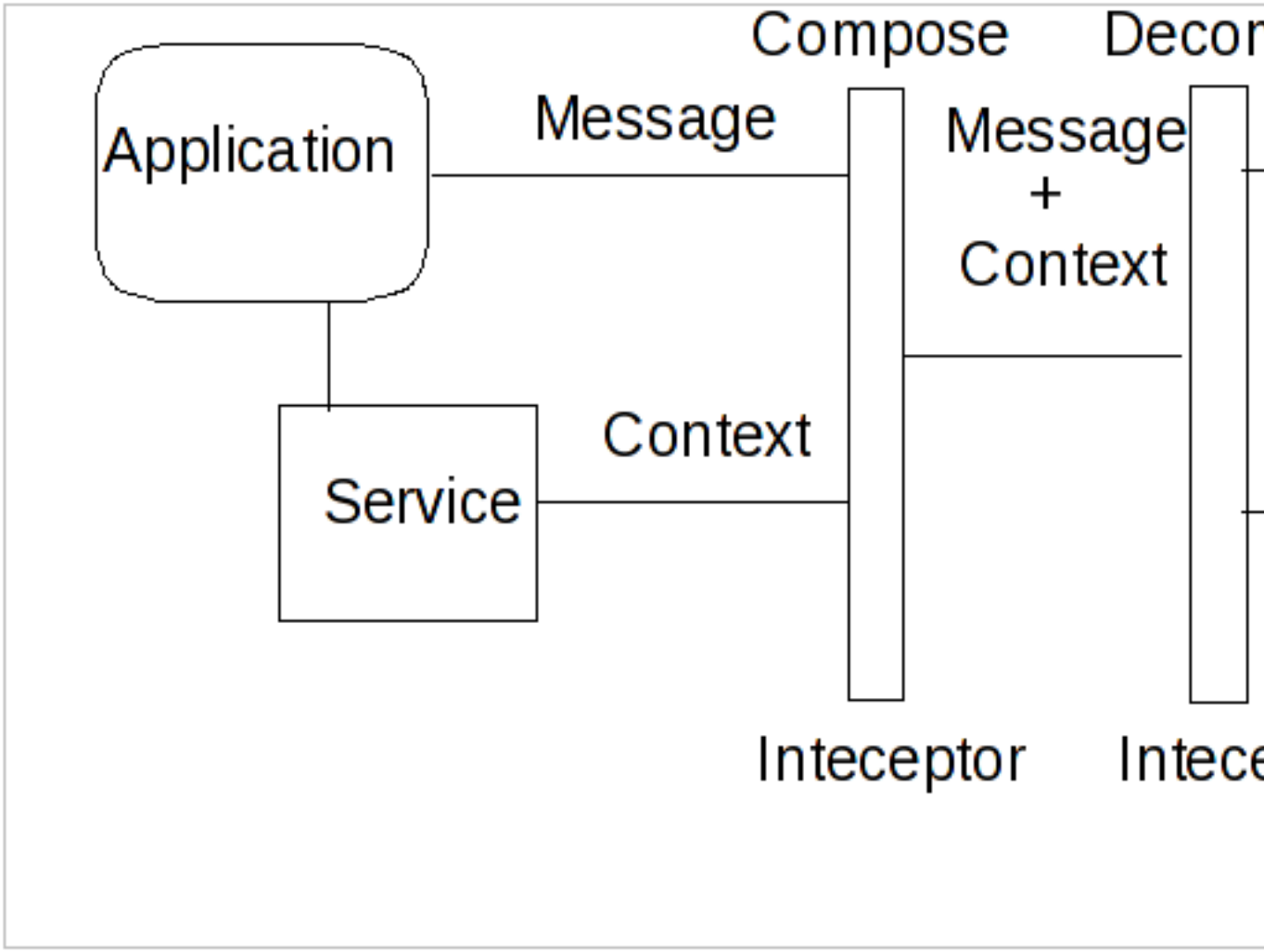

Figure 3.2. Web Services and Context Flow

3.3. Participants

The coordinator cannot know the details of how every transactional service is implemented. In fact this knowledge is not even necessary for it 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 web service receives its first service request in some given transaction, it enrolls with the coordinator as a participant, specifying the participant model it wishes to follow. The context contains a URL for the endpoint of the coordination service which handles enrollment requests. So, the term participant merely refers a transactional service enrolled in a specific transaction using a specific participant model.

3.4. ACID Transactions

Traditionally, transaction processing systems support *ACID* properties. ACID is an acronym for *A*tomic, *C*onsistent, *I*solated, and *D*urable. A unit of work has traditionally been considered transactional only if the ACID properties are maintained, as describe in *ACID Properties*.

ACID Properties

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 are not lost in the event of a failure.

3.5. Two Phase Commit

The classical two-phase commit approach is the bedrock of JBoss Transaction Service, and more generally of Web Services transactions. Two-phase commit provides coordination of parties that are involved in a transaction. The general flow of a two-phase commit transaction is described in *Figure 3.3, “Two-Phase Commit Overview”*.

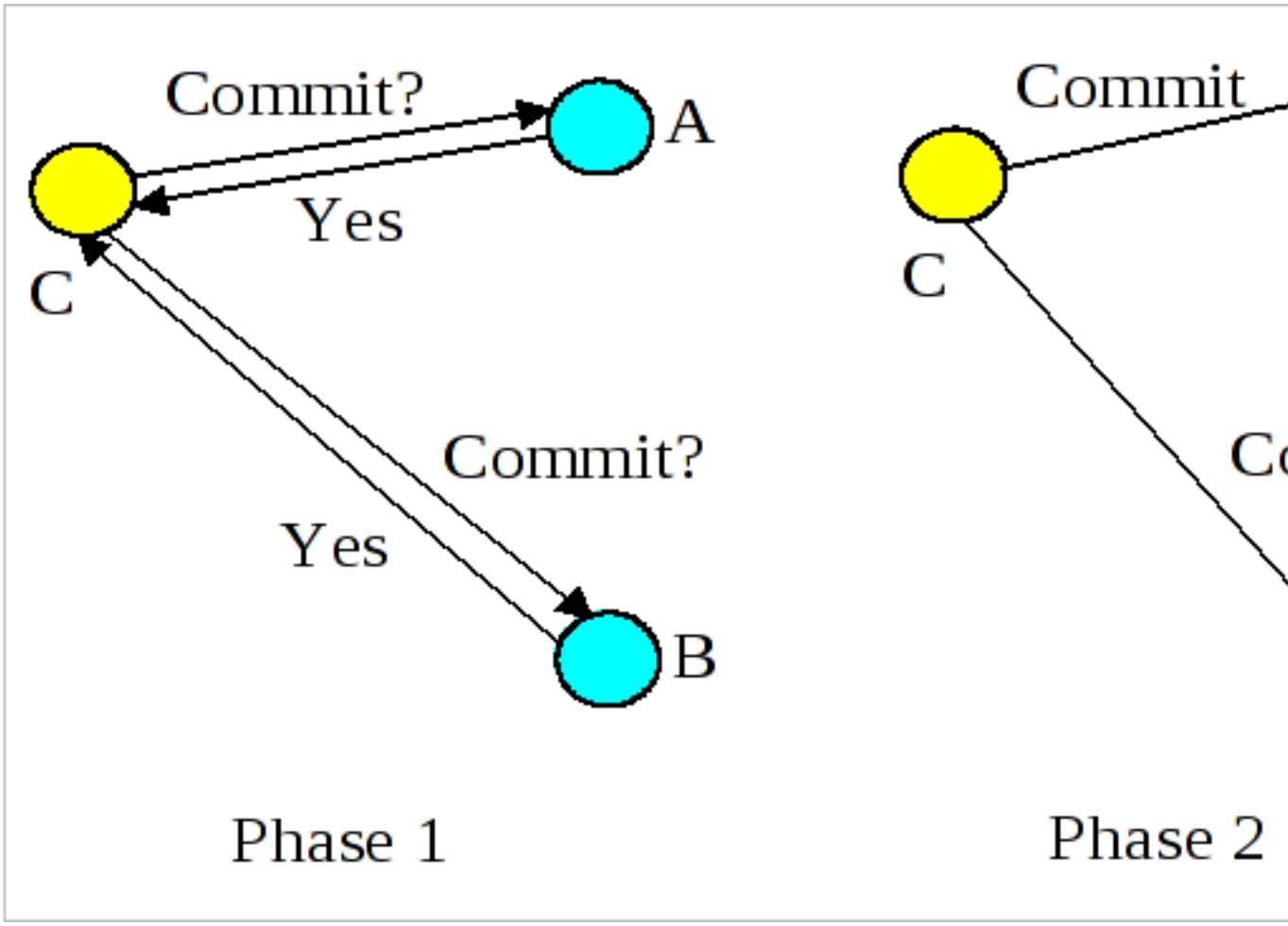

Figure 3.3. Two-Phase Commit Overview

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.

3.6. The Synchronization Protocol

Besides the two-phase commit protocol, traditional transaction processing systems employ an additional protocol, often referred to as the *synchronization protocol*. With the original ACID properties, Durability is important when state changes need to be available despite failures. Applications interact with a persistence store of some kind, such as a database, and this interaction can impose a significant overhead, because disk access is much slower to access than main computer memory.

One solution to the problem disk access time is to cache the state in main memory and only operate on the cache for the duration of a transaction. Unfortunately, this solution needs a way 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. At that point, they can flush cached state, which might be used to improve performance of an application, to a durable representation prior to the transaction committing. The synchronizations are then informed about when the transaction completes and its completion state.

Procedure 3.1. The "Four Phase Protocol" Created By Synchronizations

Synchronizations essentially turn the two-phase commit protocol into a four-phase protocol:

1. **Step 1**

   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.

2. **Steps 2 and 3**

   The coordinator then conducts the normal two-phase commit protocol.

3. **Step 4**

   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.

The synchronization protocol does not have the same failure requirements as the traditional two-phase commit protocol. For example, Synchronization participants do not need the ability to recover in the event of failures, because any failure before the two-phase commit protocol completes cause the transaction to roll back, and failures after it completes have no effect on the data which the Synchronization participants are responsible for.

3.7. 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. *Table 3.1, "Variants to the Two-Phase Commit Protocol"* gives more information about each one.

Table 3.1. Variants to the Two-Phase Commit Protocol

Variant	Description
Presumed Abort	If a transaction is going to roll back, the coordinator may record this information locally and tell all enlisted participants. Failure to contact a participant has no effect on

Variant	Description
	the transaction outcome. The coordinator is informing participants only 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, no information will be available and the requester can assume that the transaction has aborted. This optimization has the benefit that no information about participants need be made persistent until the transaction has progressed to the end of the `prepare` phase and decided to commit, since any failure prior to this point is assumed to be an abort of the transaction.
One-Phase	If only a single participant is involved in the transaction, the coordinator does not need to drive it through the `prepare` phase. Thus, the participant is told to commit, and the coordinator does not need to record information about the decision, since the outcome of the transaction is the responsibility of 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. Therefore, a read-only participant can be omitted from the second phase of the commit protocol.

Note

The WS-Atomic Transaction protocol does not support the one-phase commit optimization.

3.8. Non-Atomic Transactions and Heuristic Outcomes

In order to guarantee atomicity, the two-phase commit protocol is *blocking*. 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 cannot tolerate this blocking.

To break this blocking nature, participants that are past the `prepare` phase are allowed to make autonomous decisions about whether to commit or rollback. Such a participant must record its decision, so that it can complete the original transaction if it eventually gets a request to do so. If the coordinator eventually informs the participant of the transaction outcome, and it is the same as the choice the participant made, no conflict exists. If the decisions of the participant and coordinator are different, the situation is referred to as a non-atomic outcome, and more specifically as a *heuristic outcome*.

Resolving and reporting heuristic outcomes to the application is usually the domain of complex, manually driven system administration tools, because attempting an automatic resolution requires semantic information about the nature of participants involved in the transactions.

Precisely when a participant makes a heuristic decision depends on the specific implementation. Likewise, the choice the participant makes about whether to commit or to roll back depends upon the implementation, and possibly the application and the environment in which it finds itself. The possible heuristic outcomes are discussed in *Table 3.2, "Heuristic Outcomes"*.

Table 3.2. Heuristic Outcomes

Outcome	Description
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. One situation where this might happen is if the coordinator is able to successfully `prepare` the transaction, but then decides to roll it back because its transaction log could not be updated. While the coordinator is making its decision, the participants decides to commit.
Heuristic Mixed	Some participants commit ed, 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 the decision may possibly differ from that determined by the transaction service. This type of difference can lead to a loss of integrity in the system. Try to avoid needing to perform resolution of heuristics, either by working with services and participants that do not cause heuristics, or by using a transaction service that provides assistance in the resolution process.

3.9. Interposition

Interposition is a scoping mechanism which allows coordination of a transaction to be delegated across a hierarchy of coordinators. See *Figure 3.4, “Interpositions”* for a graphical representation of this concept.

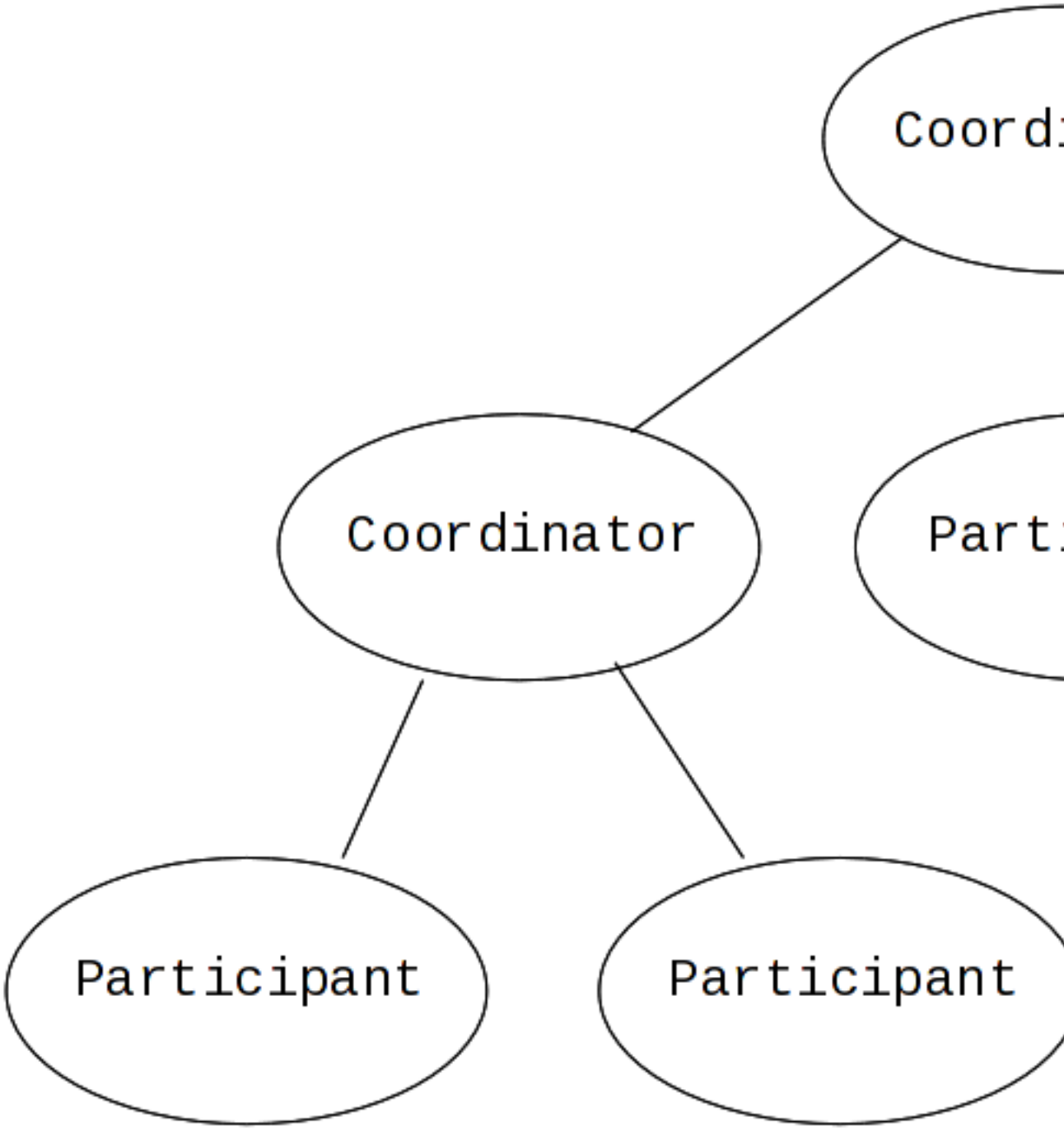

Figure 3.4. Interpositions

Interposition is particularly useful for Web Services transactions, as a way of limiting the amount of network traffic required for coordination. For example, if communications between the top-level coordinator and a web service are slow because of network traffic or distance, the web service might benefit from executing in a subordinate transaction which employs a local coordinator

service. In *Figure 3.4, "Interpositions"*,to `prepare`, the top-level coordinator only needs to send one `prepare` message to the subordinate coordinator, and receive one `prepared` or `aborted` reply. The subordinate coordinator forwards a `prepare` locally to each participant and combines the results to decide whether to send a single `prepared` or `aborted` reply.

3.10. A New Transaction Protocol

Many component technologies offer mechanisms for coordinating ACID transactions based on two-phase commit semantics. Some of these are CORBA/OTS, JTS/JTA, and MTS/MSDTC. ACID transactions are not suitable for all Web Services transactions, as explained in *Reasons ACID is Not Suitable for Web Services*.

Reasons ACID is Not Suitable for Web Services

- Classic ACID transactions assume that an organization that develops and deploys applications owns the entire infrastructure for the applications. This infrastructure has traditionally taken the form of an Intranet. Ownership implies that transactions operate in a trusted and predictable manner. To assure ACIDity, potentially long-lived locks can be kept on underlying data structures during two-phase commit. Resources can be used for any period of time and released when the transaction is complete.

  In Web Services, these assumptions are no longer valid. One obvious reason is that the owners of data exposed through a Web service refuse to allow their data to be locked for extended periods, since allowing such locks invites denial-of-service attacks.

- 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 directives of the transaction manager and only infrequently make unilateral decisions which harm other participants in a transaction.

  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.

3.10.1. Transaction in Loosely Coupled Systems

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. These requirements are discussed in *Requirements of Web Services*.

Requirements of Web Services

- Ability to handle multiple successful outcomes to a transaction, and 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 expected to suffer outages during their lifetimes, and coordinated work must be able to survive such outages.
- Interoperation using XML over multiple communication protocols. XTS uses SOAP encoding carried over HTTP.

Overview of Protocols Used by XTS

This section discusses fundamental concepts associated with the WS-Coordination, WS-Atomic Transaction and WS-Business Activity protocols, as defined in each protocol's specification. Foundational information about these protocols is important to understanding the remaining material covered in this guide.

Note

If you are familiar with the WS-Coordination, WS-Atomic Transaction, and WS-Business Activity specifications you may only need to skim this chapter.

4.1. 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 to reach consensus on a decision by a distributed transaction protocol, or to guarantee that all participants obtain a specific message, such as 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. It is specific to the type of coordination being performed.

The fundamental idea underpinning *WS-Coordination (WS-C)* is that a coordination infrastructure is needed in a Web Services environment. The WS-C 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 4.1, “WS-C Architecture”*.

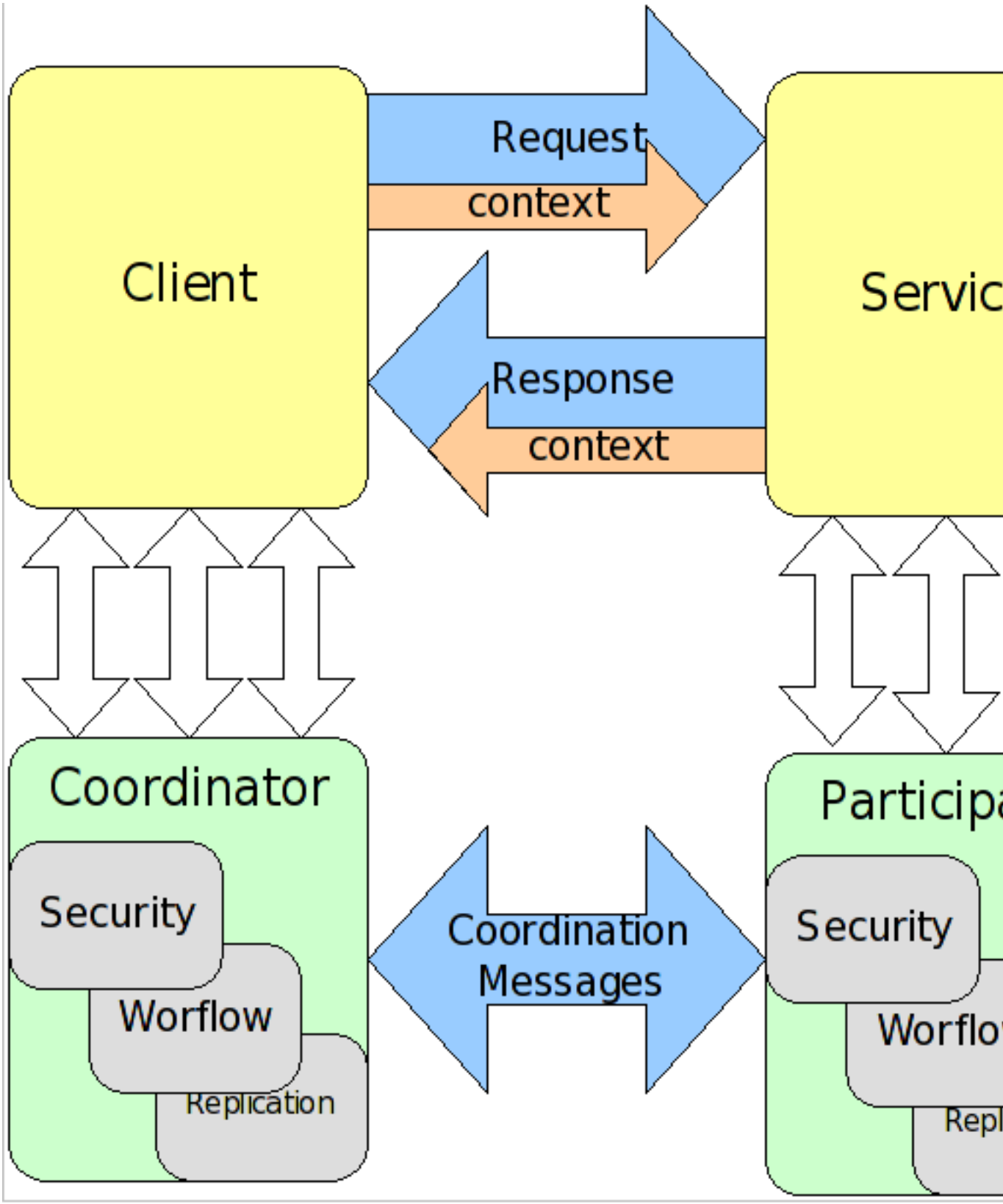

Figure 4.1. WS-C Architecture

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

Generic Requirements for WS-C

- 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 the points in *Generic Requirements for WS-C* are the direct responsibility of WS-C, while the fourth is the responsibility of a third-party entity. The third-party entity is usually the client component of the overall application. These four WS-C roles and their relationships are shown in *Figure 4.2, "Four Roles in WS-C"*.

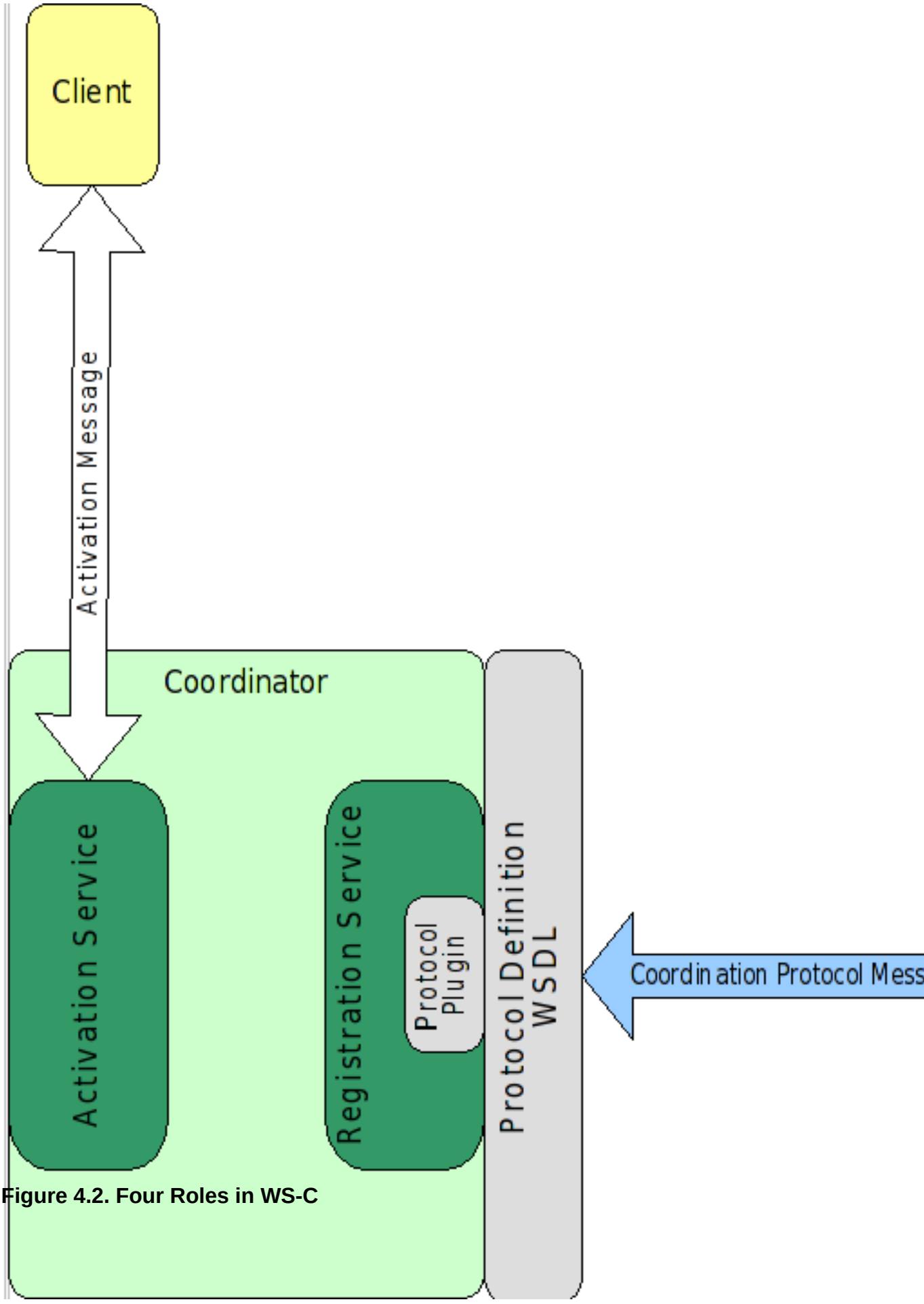

Figure 4.2. Four Roles in WS-C

4.1.1. Activation

The WS-C framework exposes an Activation Service which supports the creation of coordinators for specific coordination protocols and retrieval of associated contexts. Activation services are invoked synchronously using an RPC style exchange. So, the service WSDL defines a single port declaring a `CreateCoordinationContext` operation. This operation takes an input specfying the details of the transaction to be created, including the type of coordination required, timeout, and other relevant information. It returns an output containing the details of the newly-created transaction context: the transaction identifier, coordination type, and registration service URL.

Example 4.1.

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

Note

The 1.0 Activation Coordinator service employs an asynchronous message exchange comprised of two one-way messages, so an Activation Requester service is also necessary.

4.1.2. Registration

The context returned by the activation service includes the URL of a Registration Service. When a web service receieves a service request accompanied by a transaction context, it contacts the Registration Service to enroll as a participant in the transaction. The registration request includes a participant protocol defining the role the web service wishes to take in the transaction. Depending upon the coordination protocol, more than one choice of participant protocol may be available.

Like the activation service, the registration service assumes synchronous communication. Thus, the service WSDL exposes a single port declaring a `Register` operation. This operation takes an input specifying the details of the participant which is to be registered, including the participant protocol type. It returns a corresponding output response.

Example 4.2. Registration ServiceWSDL Interface

```
<!-- Registration Service portType Declaration -->
```

```
<wsdl:portType name="RegistrationCoordinatorPortType">
  <wsdl:operation name="Register">
    <wsdl:input message="wscoor:Register"/>
    <wsdl:output message="wscoor:RegisterResponse"/>
  </wsdl:operation>
</wsdl:portType>
```

Once a participant is registered with a coordinator through the registration service, it receives coordination messages from the coordinator. Typical messages include such things as “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 1.0 Registration Coordinator service employs an asynchronous message exchange comprised of two one way messages, so a Registration Requester service is also necessary

4.1.3. Completion

The role of terminator is generally filled by the client application. At an appropriate point, the client asks the coordinator to perform its particular coordination function with any registered participants, to drive the protocol through to its completion. After completion, the client application may be informed of an outcome for the activity. This outcome may take any form along the spectrum from simple success or failure notification, to complex structured data detailing the activity’s status.

4.2. WS-Transaction

WS-Transaction (WS-T) comprises the pair of transaction coordination protocols, *WS-Atomic Transaction (WS-AT)* and *WS-Business Activity (WS-BA)*, which utilize the coordination framework provided by *WS-Coordination (WS-C)*.

WS-Transactions was developed to unify existing traditional transaction processing systems, allowing them to communicate reliably with one another without changes to the systems' own function.

4.2.1. WS-Transaction Foundations

WS-Transaction is layered upon the WS-Coordination protocol, as shown in as shown in *Figure 4.3, “WS-Coordination, WS-Transaction, and WS-Business Activity”*.

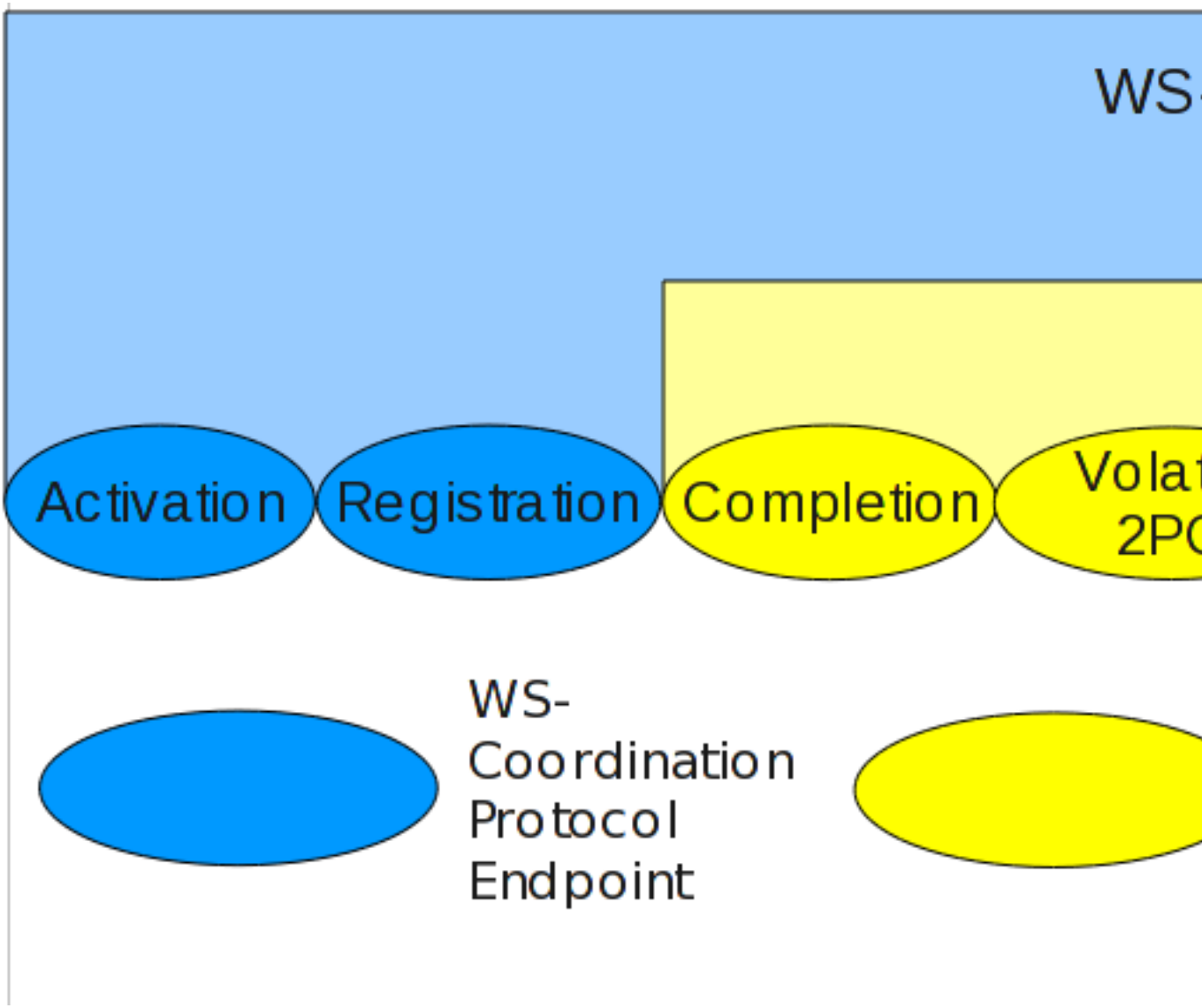

Figure 4.3. WS-Coordination, WS-Transaction, and WS-Business Activity

WS-C provides a generic framework for specific coordination protocols, like WS-Transaction, used in a modular fashion. WS-C provides only context management, allowing contexts to be created and activities to be registered with those contexts. WS-Transaction leverages the context management framework provided by WS-C in two ways.

1. It extends the WS-C context to create a transaction context.

2. 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-C protocol infrastructure.

3. An important aspect of WS-Transaction that differs from traditional transaction protocols is that a synchronous request/response model is not assumed. Sequences of one way messages are used to implement communications between the client/participant and the coordination services appropriate to the transaction's coordination and participant protocols. This is significant because it means that the client and participant containers must deploy XTS service endpoints to receive messages from the coordinator service.

   This requirement is visible in the details of the `Register` and `RegisterResponse` messages declared in the Registration Service WSDL in *Example 4.2, "Registration ServiceWSDL Interface"*. The `Register` message contains the URL of an endpoint in the client or web service container. This URL is used when a WS-Transaction coordination service wishes to dispatch a message to the clinet or web service. Similarly, the `RegisterResponse` message contains a URL iendtifying an endpoint for the protocol-specific WS-Transaction coordination service for which the client/web service is registered, allowing messages to be addressed to the transaction coordinator.

4.2.2. WS-Transaction Architecture

WS-Transaction distnguishes the transaction-aware web service in its role executing business-logic, from the web service acting as a participant in the transaction, communicating with and responding to its transaction coordinator. Transaction-aware web services deal with application clients using business-level protocols, while the participant handles the underlying WS-Transaction protocols, as shown in *Figure 4.4, "WS-Transaction Global View"*.

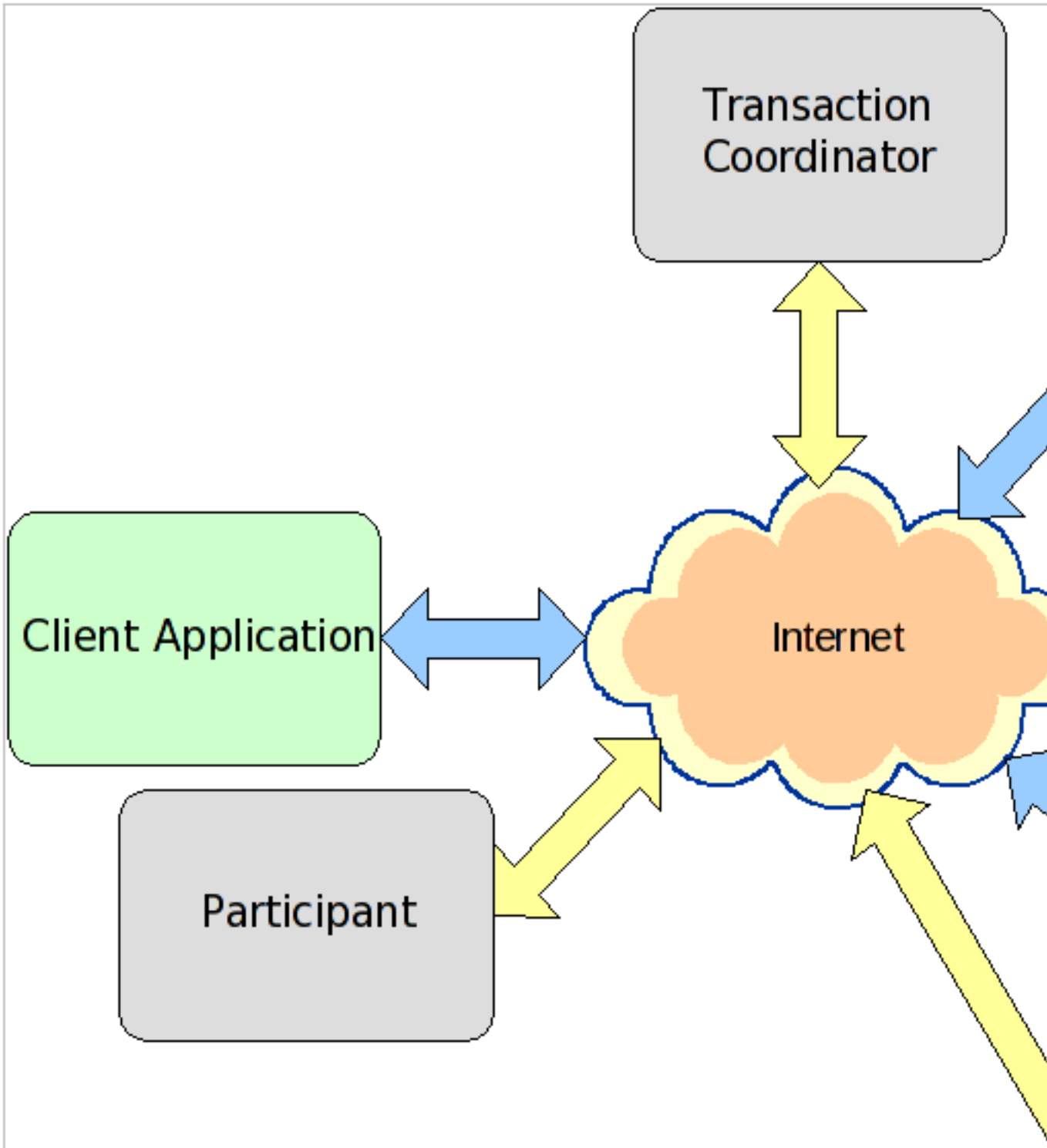

Figure 4.4. WS-Transaction Global View

A transaction-aware web service encapsulates the business logic or work that needs to be conducted within the scope of a transaction. This work cannot be confirmed by the application unless the transaction also commits. Thus, control is ultimately removed from the application and given to the transaction.

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 4.4, "WS-Transaction Global View"*, each web service is shown with one associated participant that manages the transaction protocol messages on behalf of its web service. *Figure 4.5, "WS-Transaction Web Services and Participants"*, however, shows a close-up view of a single web service, and a client application with their associated participants.

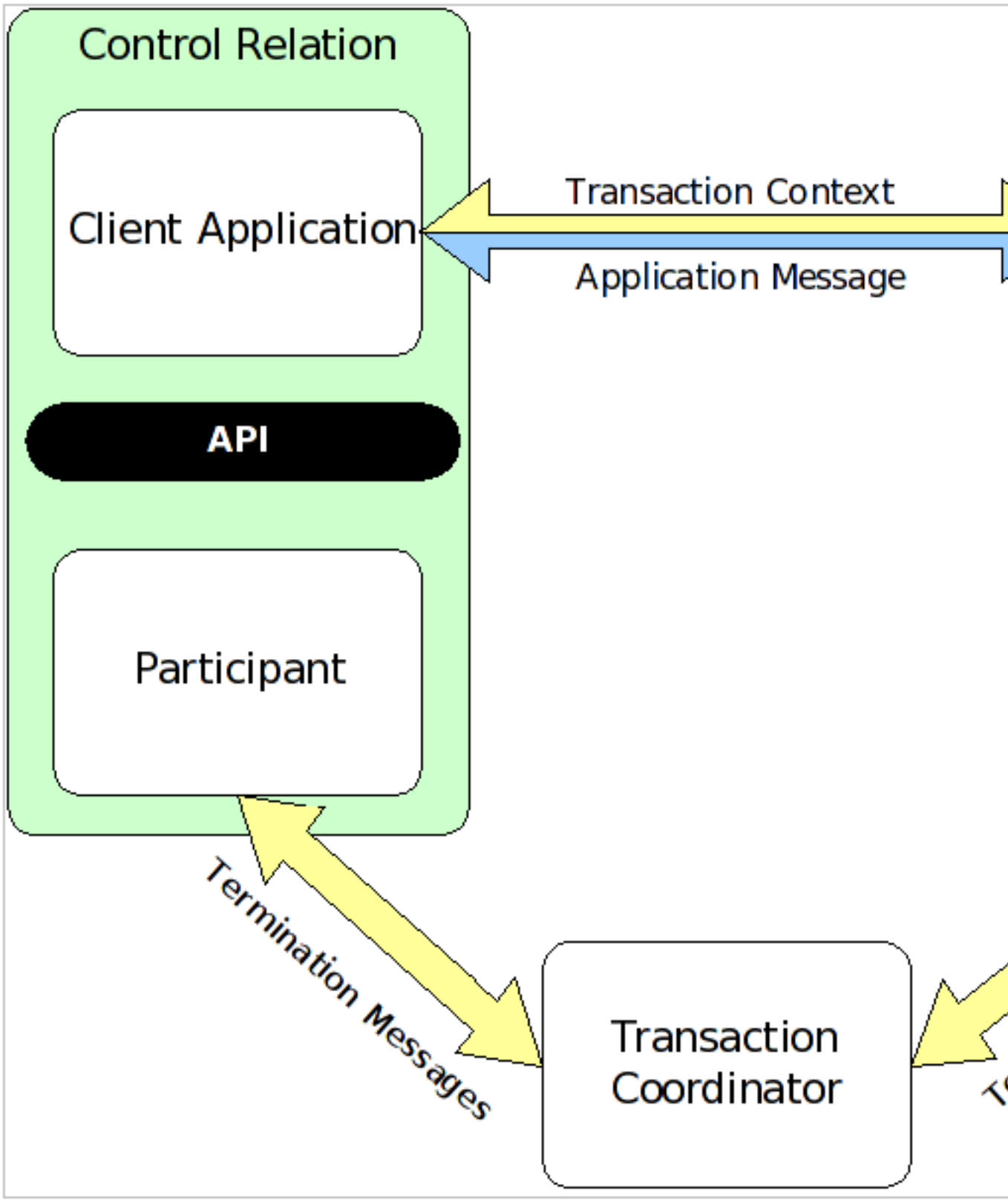

Figure 4.5. WS-Transaction Web Services and Participants

The transaction-aware web service employs a back end database accessed via a JDBC driver, which sends SQL statements to the database for processing. However, those statements should only commit if the enclosing web service transaction does. For this to work, the web service must employ transaction bridging. Transaction bridging registers a participant with the coordinator for the web service transaction and creates a matching XA transaction within which it can invoke the driver to make tentative changes to the database. The web service ensures that service requests associated with a specific web service transaction are executed in the scope of the corresponding XA transaction, grouping changes common to a given transaction while isolating changes belonging to different transactions. The participant responds to prepare, commit, or rollback requests associated from the web service transaction coordinator by forwarding the same operations to the underlying XA transaction coordinator, ensuring that the local outcome in the database corresponds with the global outcome of the web service transaction as a whole.

Things are less complex for the client. Through its API, the client application registers a participant with the transaction, and uses this participant to control termination of the transaction.

4.2.3. WS_Transaction Models

It has been established that traditional transaction models are not appropriate for Web Services. No one specific protocol is likely to be sufficient, given the wide range of situations where Web service transactions are likely to be used. The WS-Transaction specification proposes two distinct models, where each supports the semantics of a particular kind of B2B interaction.

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 *Chapter 7, The XTS API*.

4.2.3.1. Atomic Transactions

An *atomic transaction (AT)* is similar to traditional ACID transactions, and is designed to support short-duration interactions where ACID semantics are appropriate. Within the scope of an AT, web services typically employ bridging to allow them to access XA resources, such as databases and message queues, under the control of the web service transaction. When the transaction terminates, the participant propagates the outcome decision of the AT to the XA resources, and the appropriate commit or rollback actions are taken by each.

All services and associated participants are expected to provide ACID semantics, and it is expected that any use of atomic transactions occurs in environments and situations where ACID is appropriate. Usually, this environment is a trusted domain, over short durations.

Procedure 4.1. Atomic Transaction Process

1. To begin an atomic transaction, the client application first locates a WS-C Activation Coordinator web service that supports WS-Transaction.

2. The client sends a WS-C `CreateCoordinationContext` message to the service, specifying *http://schemas.xmlsoap.org/ws/2004/10/wsat* as its coordination type.

3. The client receives an appropriate WS-Transaction context from the activation service.

4. 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*. It also contains a reference to the atomic transaction coordinator endpoint, the WS-C Registration Service, where participants can be enlisted.

5. The client normally proceeds to invoke Web Services and complete the transaction, either committing all the changes made by the web services, or 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.

6. 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.

7. Once all the necessary application-level work is complete, the client can terminate the transaction, with the intent of making any changes to the service state permanent. The completion participant instructs the coordinator to try to commit or roll back the transaction. When the commit or roll-back operation completes, a status is returned to the participant to indicate the outcome of the transaction.

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

Volatile2pc

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

When the completion participant initiates a `commit` operation, all Volatile2PC participants are informed that the transaction is about to complete, via the `prepare` message. The participants can respond with one of three messages: `prepared`, `aborted`, or `readonly`. A failure at this stage causes the transaction to roll back.

Durable2PC

The next protocol in the WS-Atomic Transaction is *Durable2PC*. The Durable2PC protocol is at the core of WS-Atomic Transaction. It brings about the necessary consensus between participants in a transaction, so the transaction can safely be terminated.

The Durable2PC protocol ensures atomicity between participants, and is based on the classic technique of two-phase commit with presumed abort.

Procedure 4.2. Durable2PC Procedure

1. 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, so these changes can either be rolled back or committed later. None of the original state information can be lost at this point, since the atomic transaction may still roll back. If the participant cannot `prepare`, it must inform the coordinator, by means of the `aborted` message. The transaction will ultimately roll back. If the participant is responsible for a service that did not change any of the transaction's data,, it can return the `readonly` message, causing it to be omitted from the second phase of the commit protocol. Otherwise, the `prepared` message is sent by the participant.

2. If no failures occur during the first phase, Durable2PC proceeds to the second phase, in which the coordinator sends the `commit` message to participants. Participants then make permanent the tentative work done by their associated services, and send a `committed` message to the coordinator. If any failures occur, the coordinator sends the `rollback` message to all participants, causing them to discard tentative work done by their associated services, and delete any state information saved to persistent storage at `prepare`, if they have reached that stage. Participants respond to a rollback by sending an `aborted` message to the coordinator.

Note

The semantics of the WS-Atomic Transaction protocol do not include the one-phase commit optimization. A full two-phase commit is always used, even where only a single participant is enlisted.

Figure 4.6, "WS-Atomic Two-Phase Participant State Transitions" shows the state transitions of a WS-Atomic Transaction and the message exchanges between coordinator and participant. Messages generated by the coordinator are represented by solid lines, while the participants' messages use dashed lines.

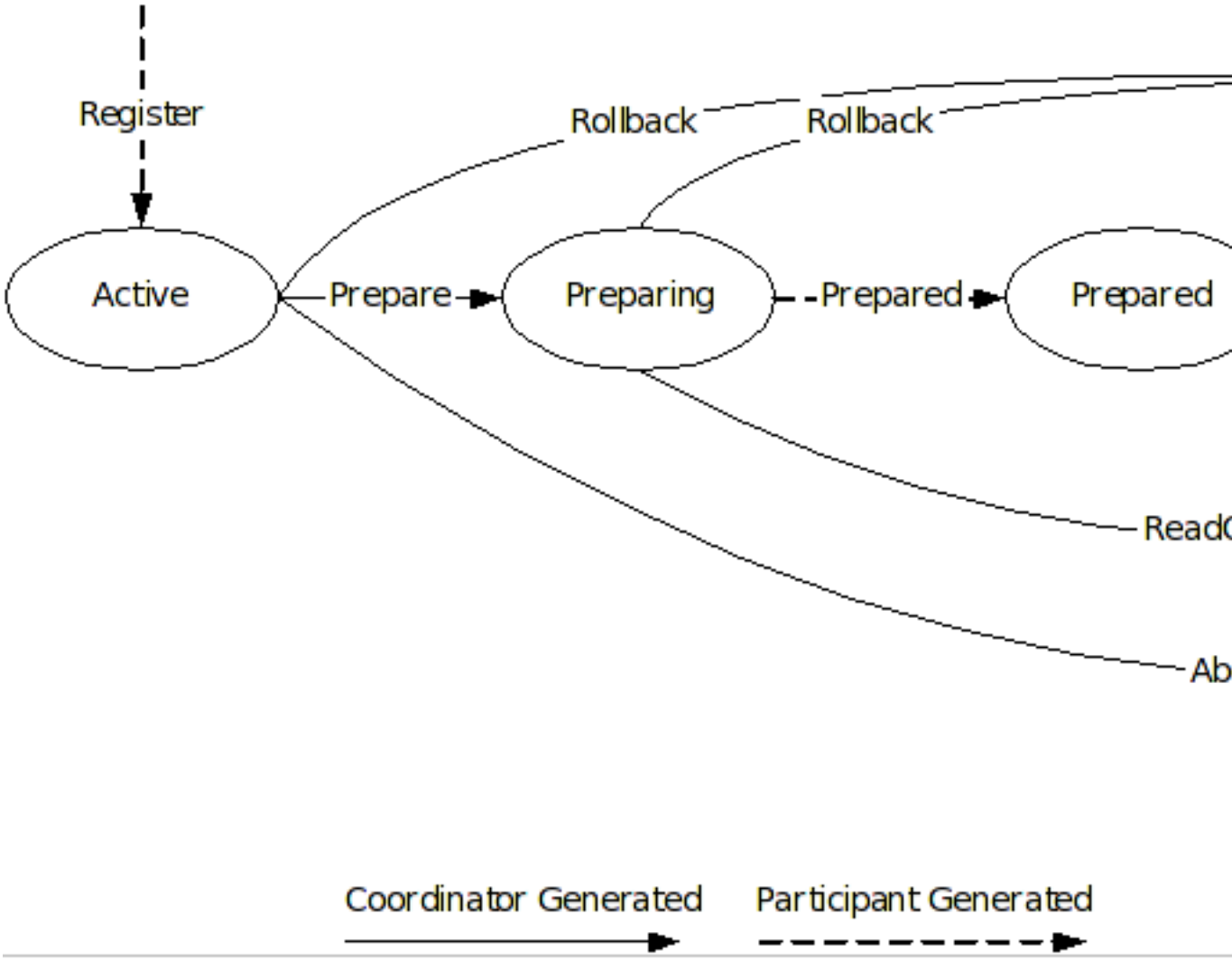

Figure 4.6. WS-Atomic Two-Phase Participant State Transitions

Once the Durable2PC protocol completes, 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 about the transaction's completion, so that they can release resources such as database connections.

Any registered Volatile2PC participants are invoked after the transaction terminates, and are informed about the transaction's completion state by the coordinator. Since the transaction has terminated, any failures of participants at this stage are ignored, since they have no impact on outcomes.

Figure 4.7, “” illustrates the intricate interweaving of individual protocols comprising the AT as a whole.

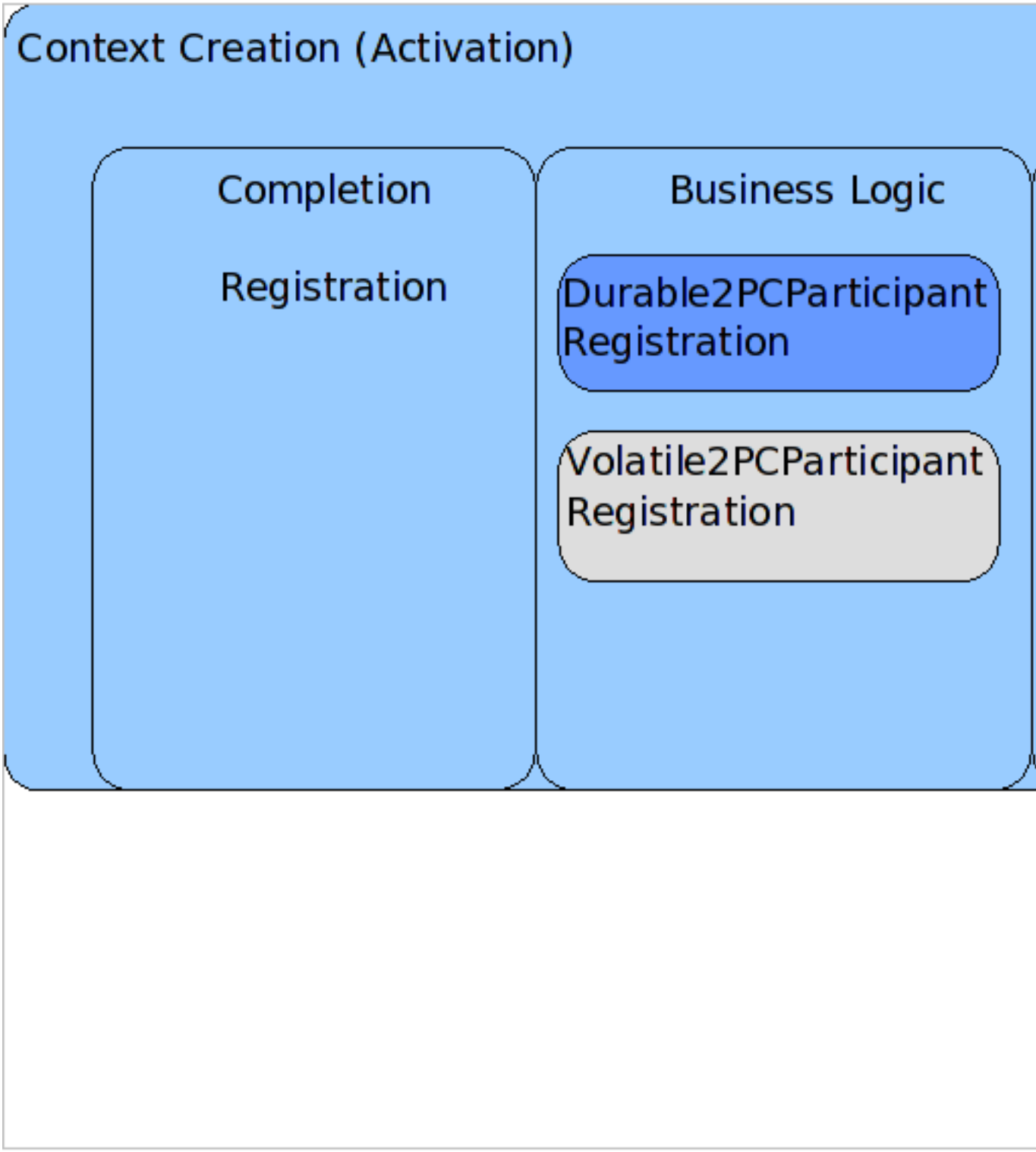
Context Creation (Activation)
Completion
Registration
Business Logic
Durable2PCParticipant Registration
Volatile2PCParticipant Registration

Figure 4.7.

4.2.3.2. Business Activities

Most B2B 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. 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. However, if the individual does not purchase the books within that period, they become available again for purchase by other customers. Because it is not possible to have an infinite supply of stock, some online shops may seem, from the user's perspective, to reserve items for them, while actually allow others to preempt the reservation. A user may discover, to his disappointment, that the item is no longer available.

A *Business Activity (BA)* is designed specifically for these kinds of long-duration interactions, where it is impossible or impractical to exclusively lock resources.

Procedure 4.3. BA Process Overview

1. Services are requested to do work.
2. Where those services have the ability to undo any work, they inform the BA, in case the BA later decides the cancel the work. If the BA suffers a failure. it can instruct the service to execute its `undo` behavior.

The key to BA is that how services do their work and provide compensation mechanisms is not the responsibility of the WS-BA specification. It is delegated to the service provider.

The WS-BA defines a protocol for Web Services-based applications to enable existing business processing and work-flow systems to wrap their proprietary mechanisms and interoperate across implementations and business boundaries.

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

The BA protocols employ a compensation-based transaction model. When a participant in a business activity completes its work, it may choose to exit the activity. This choice does not allow any subsequent rollback. 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 asks each non-exited participant to compensate for the failure, giving them the opportunity to execute

whatever compensating action they consider appropriate. For instance, participant might credit a bank account which it previously debited. If 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, detailed in *Assumptions of WS-BA*.

Assumptions of WS-BA

- 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, not as the output of the request. Message I/O implementations typically have timeout requirements too short for BA responses. If the response is not received after a timeout, it is re-sent, repeatedly, until a response is received. The receiver discards all but one identical request received.

The BA model has two participant protocols: `BusinessAgreementWithParticipantCompletion` and `BusinessAgreementWithCoordinatorCompletion`. Unlike the AT protocols which are driven from the coordinator down to participants, this protocol takes the opposite approach.

BusinessAgreementWithParticipantCompletion

1. A participant is initially created in the Active state.
2. If it finishes its work and it is no longer needed within the scope of the BA (such as when the activity operates on immutable data), the participant can unilaterally decide to exit, sending an `exited` message to the coordinator. However, if the participant finishes and wishes to continue in the BA, 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 it about the final outcome of the BA. This outcome is either a `close` message, meaning the BA has completed successfully, or a `compensate` message indicating that the participant needs to reverse its work.

BusinessAgreementWithCoordinatorCompletion

The `BusinessAgreementWithCoordinatorCompletion` differs from the `BusinessAgreementWithParticipantCompletion` protocol in that the participant cannot autonomously decide to complete its participation in the BA, even if it can be compensated.

1. Instead, the completion stage is driven by the client which created the BA, which sends a `completed` message to the coordinator.

2. The coordinator sends a `complete` message to each participant, indicating that no further requests will be sent to the service associated with the participant.

3. The participant continues on in the same manner as in the BusinessAgreementWithParticipantCompletion protocol.

The advantage of the BA model, compared to the AT model, is that is allows the participation of services that cannot lock resources for extended periods.

While the full ACID semantics are not maintained by a BA, consistency can still be maintained through compensation. The task of writing correct compensating actions to preserve overall system consistency is the responsibility of the developers of the individual services under control of the BA. Such compensations may use backward error recovery, but forward recovery is more common.

Figure 4.8, “” shows the state transitions of a WS-BA `BusinessAgreementWithParticipantCompletion` participant and the message exchanges between coordinator and participant. Messages generated by the coordinator are shown with solid lines, while the participants' messages are illustrated with dashed lines.

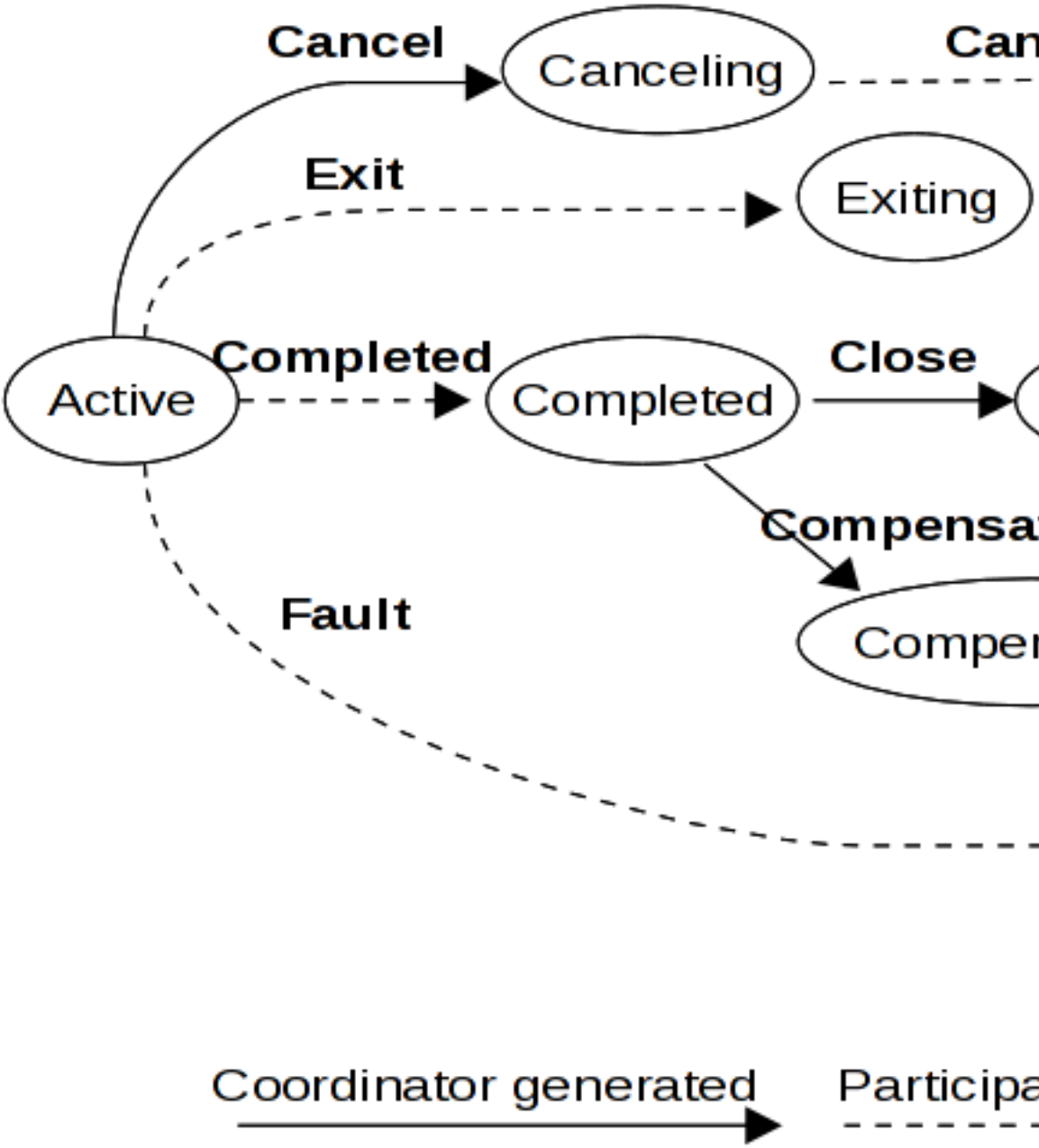

Figure 4.8.

Figure 4.9, “” shows the state transitions of a WS-BA `BusinessAgreementWithCoordinatorCompletion` participant and the message exchanges between coordinator and participant. Messages generated by the coordinator are shown with solid lines, while the participants' messages are illustrated with dashed lines.

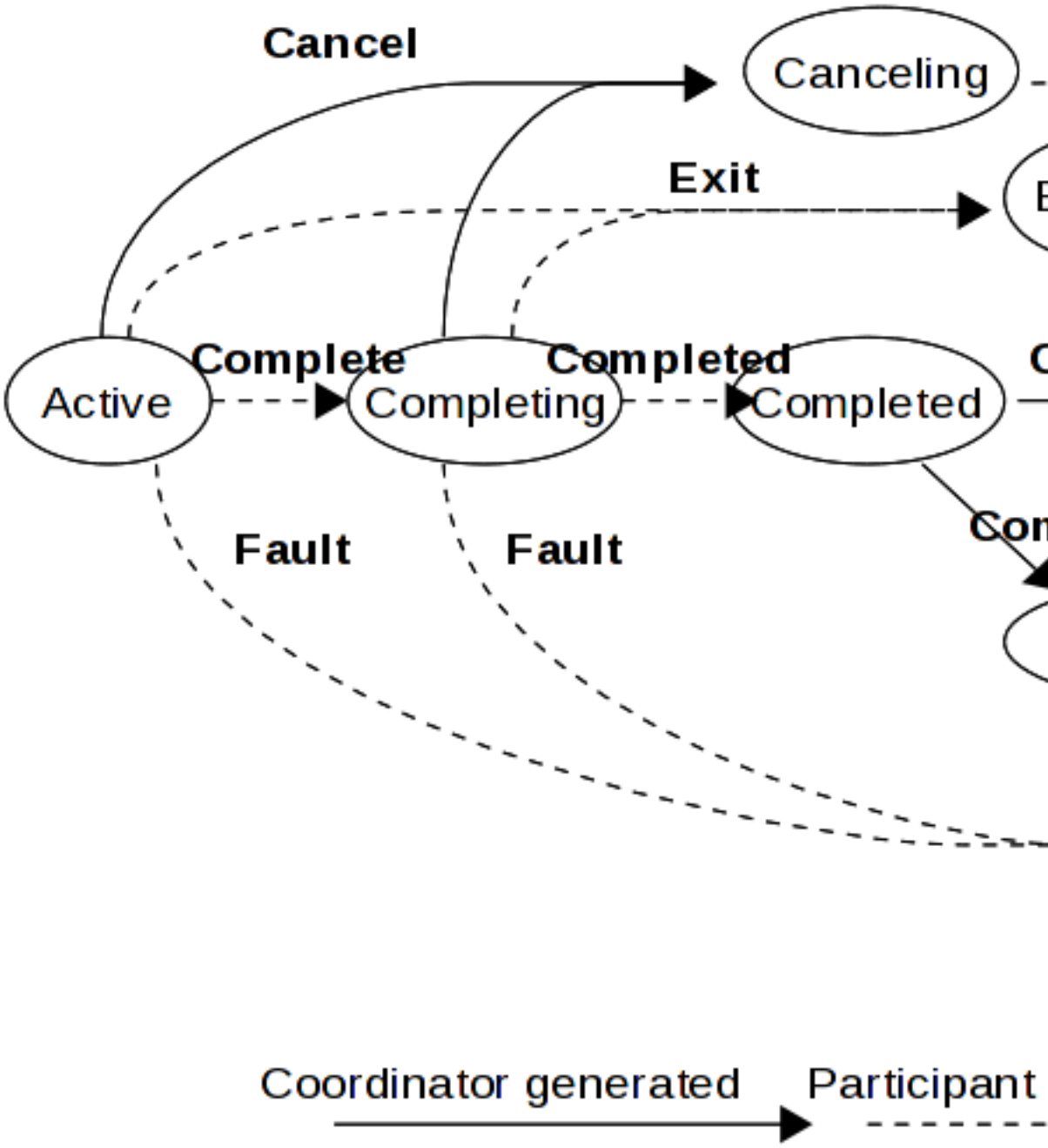

Figure 4.9.

4.2.4. Application Messages

Application messages are the requests and responses 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. This means that you are free to use any appropriate Web Services protocol. In XTS, the transaction context is propagated within the headers of SOAP messages.

XTS ships with support for service developers building WS-Transactions-aware services on the JBoss Application Server. Interceptors are provided for automatic context handling at both client and service, which significantly simplifies development, allowing you to concentrate on writing the business logic without being sidetracked by the transactional infrastructure. The interceptors add and remove context elements to application messages, without altering the semantics of the messages themselves. Any service which understands what to do with a WS-C context can use it. Services which are not aware of WS-C, WS-Atomic Transaction and WS-Business Activity can ignore the context. XTS manages contexts without user intervention.

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

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

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` and `prepare`.

Note

XTS messages do not interfere with messages from the application, an application need not use the same transport as the transaction-specific messages. For example, a client application might deliver its application-specific messages using SOAP RPC over SMTP, even though the XTS messages are delivered using a different mechanism.

4.3. Summary

XTS provides a coordination infrastructure which allows transactions to run between services owned by different business, 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, because its functionality is exposed through a simple transactioning API. XTS provides everything necessary to keep application and transactional aspects of an application separate, and to ensure that a system's use of transactions does not interfere with the functional aspects of the system itself.

Getting Started

5.1. Installing the XTS Service Archive into JBoss Transaction Service

XTS, which is the Web Services component of JBoss Transaction Service, provides WS-AT and WS-BA support for Web Services hosted on the JBoss Application Server. The module is packaged as a *Service Archive (.sar)* located in `$JBOSS_HOME/docs/examples/transactions/`. To install it, follow *Installing the XTS Module*.

Procedure 5.1. Installing the XTS Module

1. Create a sub-directory in the `$JBOSS_HOME/server/[name]/deploy/` directory, called `jbossxts.sar/`.
2. Unpack the SAR, which is a ZIP archive, into this new directory.
3. Restart JBoss Application Server to activate the module.

5.2. Creating Client Applications

There are two aspects to a client application using XTS, the transaction declaration aspects, and the business logic. The business logic includes the invocation of Web Services.

Transaction declaration aspects are handled automatically with the XTS client API. This API provides simple transaction directives such as `begin`, `commit`, and `rollback`, which the client application can use to initialize, manage, and terminate transactions. Internally, this API uses SOAP to invoke operations on the various WS-C, WS-AT and WS-BA services, in order to create a coordinator and drive the transaction to completion.

5.2.1. User Transactions

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

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

Full details of the WS-AT APIs are provided in *Chapter 7, The XTS API*.

5.2.2. 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 the `begin` method of the `UserBusinessActivity` class. A client can terminate a business activity by calling the `close` method, and cancel it by calling the `cancel` method.

If any of the Web Services invoked by the client register for the `BusinessActivityWithCoordinatorCompletion` protocol, the client can call the `completed` method before calling the `close` method, to notify the services that it has finished making service invocations in the current activity.

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

Full details of the WS-AT APIs are provided in *Chapter 7, The XTS API*.

5.2.3. Client-Side Handler Configuration

XTS does not require the client application to use a specific API to perform invocations on transactional Web Services. The client is free to use any appropriate API to send SOAP messages to the server and receive SOAP responses. The only requirements imposed on the client are:

- It must forward details of the current transaction to the server when invoking a web service.
- 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 extract the context details from the headers of incoming messages and associate the context with the current thread. To simplify this process, the XTS module includes handlers which can perform this task automatically. These handlers are designed to work with JAX-WS clients.

Note

If you choose to use a different SOAP client/server infrastructure for business service invocations, you must provide for header processing. XTS only provides interceptors for or JAX-WS. A JAX-RPC handler is provided only for the 1.0 implementation.

5.2.3.1. JAX-WS Client Context Handlers

In order to register the JAX-WS client-side context handler, the client application uses the APIs provided by the `javax.xml.ws.BindingProvider` and `javax.xml.ws.Binding` classes, to install

a handler chain on the service proxy which is used to invoke the remote endpoint. Refer to the example application client implementation located in the `src/com/jboss/jbosstm/xts/demo/BasicClient.java` file for an example.

You can also specify the handlers by using a configuration file deployed with the application. The file is identified by attaching a `javax.jws.HandlerChain` annotation to the interface class, which declares the JAX-WS client API. This interface is normally generated from the web service WSDL port definition.

You need to instantiate the `com.arjuna.mw.wst11.client.JaxWSHeaderContextProcessor` class when registering a JAX-WS client context handler.

5.3. Creating Transactional Web Services

The two parts to implementing a Web service using XTS are the transaction management and 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*, 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 ensuring that its activities are in step with those of the client and other services involved in the transaction. Internally, this API uses SOAP to invokes operations on the various WS-C, WS-AT and WS-BA services, to drive the transaction to completion.

5.3.1. Participants

A *participant* is a software entity which is driven by the transaction manager on behalf of a Web service. When a web service wants to participate in a particular transaction, it must enroll 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, implements the Durable2PCParticipant interface. The use of participants allows the transactional control management aspects of the Web service to be factored into the participant implementation, while staying separate from the the rest of the Web service's business logic and private transactional data management.

The creation of participants is not trivial, since they ultimately reflect the state of a Web service's back-end processing facilities, an aspect normally associated with an enterprise's own IT infrastructure. Implementations must use one of the following interfaces, depending upon the protocol it will participate within: `com.arjuna.wst11.Durable2PCParticipant`, `com.arjuna.wst11.Volatile2PCParticipant`, `com.arjuna.wst11.BusinessAgreementWithParticipantCompletionParticipant`, or `com.arjuna.wst11.BusinessAgreementWithCoordinatorCompletionParticipant`.

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

5.3.2. Service-Side Handler Configuration

A transactional Web service must ensure that a service invocation is included in the appropriate transaction. This usually only affects the operation of the participants and has no impact on the operation of the rest of the Web service. XTS simplifies this task and decouples it from the business logic, in much the same way as for transactional clients*Add an xref*. 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 5.1, "Context Handlers Registered with the SOAP Server"*.

The service side handlers for JAX-WS come in two different versions. The normal handler resumes any transaction identified by an incoming context when the service is invoked, and suspends this transaction when the service call completes. The alternative handler is used to interpose a local coordinator. The first time an incoming parent context is seen, the local coordinator service creates a subordinate transaction, which is resumed before the web service is called. The handler ensures that this subordinate transaction is resumed each time the service is invoked with the same parent context. When the subordinate transaction completes, the association between the parent transaction and its subordinate is cleared.

Note

The subordinate service side handler is only able to interpose a subordinate coordinator for an Atomic Transaction.

Note

JAX-RPC is provided for the 1.0 implementation only.

5.3.2.1. JAX-WS Service Context Handlers

To register the JAX-WS server-side context handler with the deployed Web Services, you must install a handler chain on the Server Endpoint Implementation class. The endpoint implementation class annotation, which is the one annotated with a `javax.jws.WebService`, must be supplemented with a `javax.jws.HandlerChain` annotation which identifies a handler configuration file deployed with the application. Please refer to the example application configuration file located at `dd/jboss/context-handlers.xml` and the endpoint implementation classes located in `src/com/jboss/jbosstm/xts/demo/services` for an example.

When registering a normal JAX-WS service context handler, you must instantiate the `com.arjuna.mw.wst11.service.JaxWSHeaderContextProcessor` class. If you need coordinator interposition, employ the `com.arjuna.mw.wst11.service.JaxWSSubordinateHeaderContextProcessor` instead.

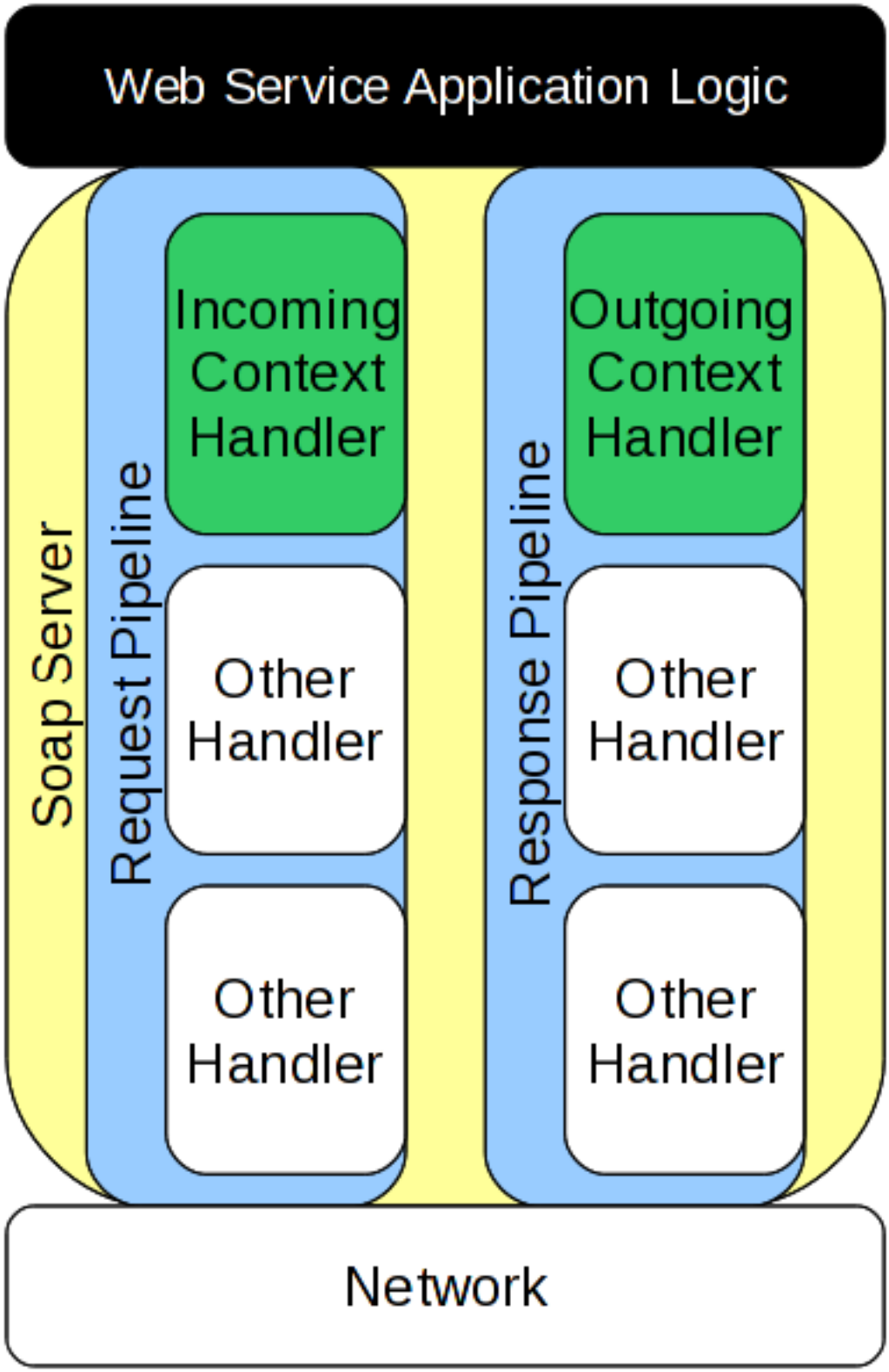

Figure 5.1. Context Handlers Registered with the SOAP Server

5.4. Summary

This chapter gives a high-level overview of each of the major software pieces used by the Web Services transactions component of JBoss Transaction Service. The Web Services transaction manager provided by JBoss Transaction Service is the hub of the architecture and is the only piece of software that user-level software does not bind to directly. XTS provides header-processing infrastructure for use with Web Services transactions contexts for both client applications and Web Services. XTS provides a simple interface for developing transaction participants, along with the necessary document-handling code.

This chapter is only an overview, and does not address the more difficult and subtle aspects of programming Web Services. For fuller explanations of the components, please continue reading.

Participants

6.1. 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 (in the example code, a theater booking system) contains some business logic to reserve a seat and inquire about availability, but it needs to be supported by something that maintains information in a durable manner. Typically this is a database, but it could be a file system, NVRAM, or other storage mechanism.

Although the service may talk to the back-end database directly, it cannot commit or undo any changes, since committing and rolling back are ultimately under the control of a transaction. For the transaction to exercise this control, it must communicate with the database. In XTS, participant does this communication, as shown in *Figure 6.1, “Transactions, Participants, and Back-End Transaction Control”*.

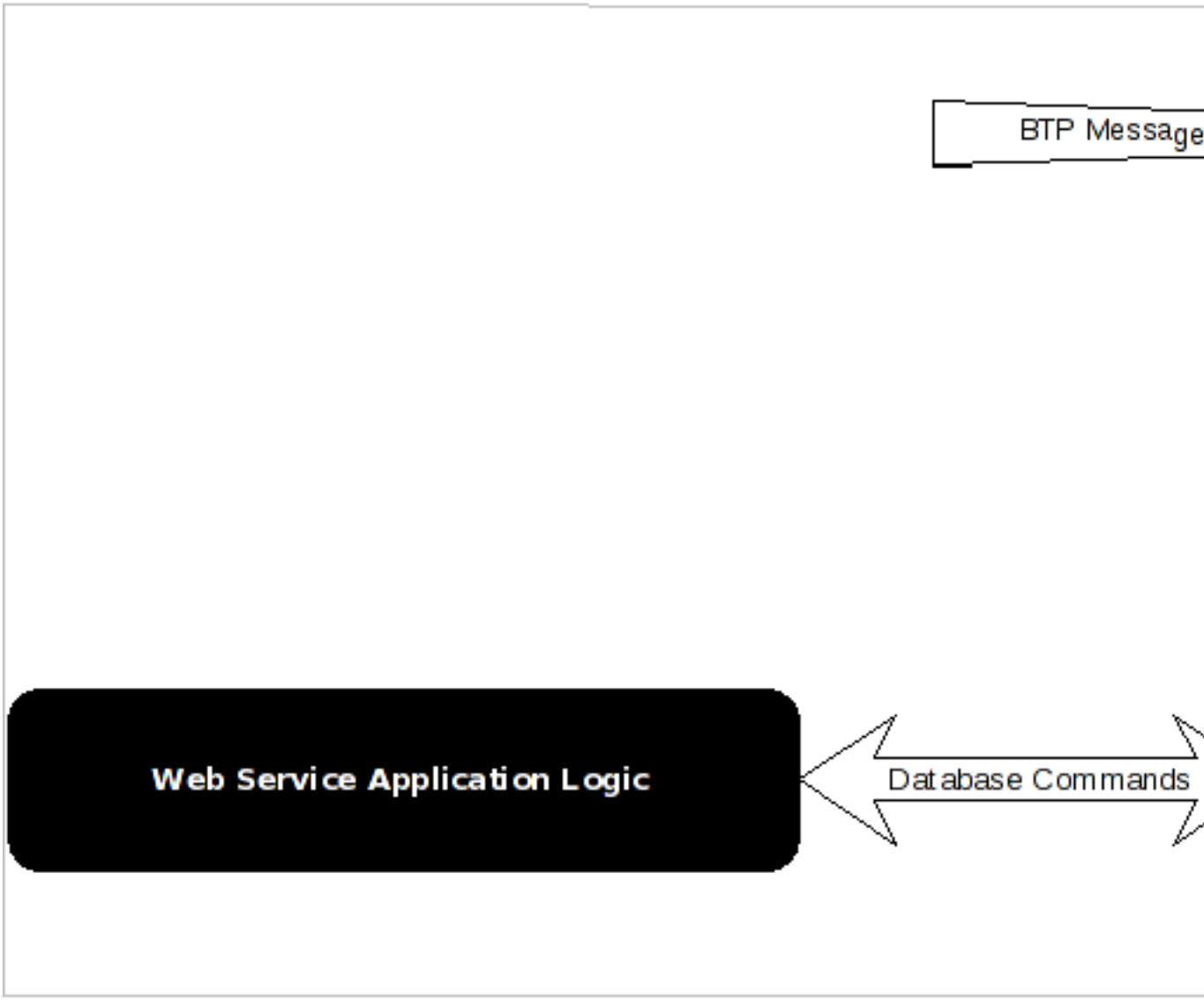

Figure 6.1. Transactions, Participants, and Back-End Transaction Control

6.1.1. Atomic Transaction

All Atomic Transaction participants are instances of the *Section 6.1.1.1, "Durable2PCParticipant"* or *Section 6.1.1.2, "Volatile2PCParticipant"*.

6.1.1.1. Durable2PCParticipant

A Durable2PCParticipant supports the WS-Atomic Transaction Durable2PC protocol with the signatures listed in *Durable2PCParticipant Signatures*, as per the `com.arjuna.wst11.Durable2Participant` interface.

Durable2PCParticipant Signatures

`prepare`
: The participant should perform any work necessary, so that it can either commit or roll back 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 indicates whether it can `prepare` by returning an instance of the `com.arjuna.wst11.Vote`, with one of three values.

  - `ReadOnly` indicates that the participant does not need to be informed of the transaction outcome, because it did not update any state information.
  - `Prepared` indicates that the participant is ready to commit or roll back, depending on the final transaction outcome. Sufficient state updates have been made persistent to accomplish this.
  - `Aborted` indicates that the participant has aborted and the transaction should also attempt to do so.

`commit`
: The participant should make its work permanent. How it accomplishes this depends upon its implementation. For instance, in the theater example, the reservation of the ticket is committed. If commit processing cannot complete, the participant should throw a `SystemException` error, potentially leading to a heuristic outcome for the transaction.

`rollback`
: The participant should undo its work. If rollback processing cannot complete, the participant should throw a `SystemException` error, potentially leading to a heuristic outcome for the transaction.

`unknown`
: This method has been deprecated and is slated to be removed from XTS in the future.

`error`
: In rare cases when recovering from a system crash, it may be impossible to complete or roll back a previously prepared participant, causing the `error` operation to be invoked.

6.1.1.2. Volatile2PCParticipant

This participant supports the WS-Atomic Transaction Volatile2PC protocol with the signatures listed in *Volatile2PCParticipant Signatures*, as per the `com.arjuna.wst11.Volatile2Participant` interface.

Volatile2PCParticipant Signatures

`prepare`
: The participant should perform any work necessary to flush any volatile data created by the Web service under the scope of the transaction, to the system store. The implementation is free to do whatever it needs to in order to fulfill the implicit contract between it and the coordinator.

  The participant indicates whether it can `prepare` by returning an instance of the `com.arjuna.wst11.Vote`, with one of three values.

  - `ReadOnly` indicates that the participant does not need to be informed of the transaction outcome, because it did not change any state information during the life of the transaction.
  - `Prepared` indicates that the participant wants to be notified of the final transaction outcome via a call to `commit` or `rollback`.
  - `Aborted` indicates that 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. These cleanup activities depend upon its implementation. For instance, it may 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` error. This will not affect the outcome of the transaction but will cause an error to be logged. 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` error. This will not affect the outcome of the transaction but will cause an error to be logged. This method may not be called if a crash occurs during commit processing.

unknown
: This method is deprecated and will be removed in a future release of XTS.

error
: This method should never be called, since volatile participants are not involved in recovery processing.

6.1.2. Business Activity

All Business Activity participants are instances one or the other of the interfaces described in *Section 6.1.2.1, "BusinessAgreementWithParticipantCompletion"* or *Section 6.1.2.2, "BusinessAgreementWithCoordinatorCompletion"* interface.

6.1.2.1. BusinessAgreementWithParticipantCompletion

The `BusinessAgreementWithParticipantCompletion` interface supports the WS-Transactions `BusinessAgreementWithParticipantCompletion` protocol with the signatures listed in *BusinessAgreementWithParticipantCompletion Signatures*, as per interface `com.arjuna.wst11.BusinessAgreementWithParticipantCompletionParticipant`.

`BusinessAgreementWithParticipantCompletion` Signatures

`close`
: The transaction has completed successfully. The participant has 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, and it is now requested to do so. If compensation cannot be performed, the participant should throw a `FaultedException` error, potentially leading to a heuristic outcome for the transaction. If compensation processing cannot complete because of a transient condition then the participant should throw a `SystemException` error, 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`
: This method is deprecated and will be removed a future XTS release.

error
: In rare cases when recovering from a system crash, it may be impossible to compensate a previously-completed participant. In such cases the `error` operation is invoked.

6.1.2.2. BusinessAgreementWithCoordinatorCompletion

The BusinessAgreementWithCoordinatorCompletion participant supports the WS-Transactions `BusinessAgreementWithCoordinatorCompletion` protocol with the signatures listed in *BusinessAgreementWithCoordinatorCompletion Signatures*, as per the `com.arjuna.wst11.BusinessAgreementWithCoordinatorCompletionParticipant` interface.

BusinessAgreementWithCoordinatorCompletion Signatures

`close`
: The transaction completed successfully. The participant previously informed the coordinator that it was ready to complete.

`cancel`
: The transaction canceled, and the participant should undo any work.

`compensate`
: The transaction canceled. The participant previously informed the coordinator that it had finished work but could compensate later if required, and it is now requested to do so. In the unlikely event that compensation cannot be performed the participant should throw a `FaultedException` error, potentially leading to a heuristic outcome for the transaction. If compensation processing cannot complete because of a transient condition, the participant should throw a `SystemException` error, 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 all work it needs to do within the scope of this business activity has been completed and that it should make permananent any provisional changes it has made.

`status`
: Returns the status of the participant.

`unknown`
: This method is deprecated and will be removed in a future release of XTS.

`error`
: In rare cases when recovering from a system crash, it may be impossible to compensate a previously completed participant. In such cases, the `error` method is invoked.

6.1.2.3. BAParticipantManager

In order for the Business Activity protocol to work correctly, the participants must be able to autonomously notify the coordinator about changes in their status. Unlike the Atomic Transaction protocol, where all interactions between the coordinator and participants are instigated by the coordinator when the transaction terminates, the BAParticipantManager interaction pattern requires the participant to be able to talk to the coordinator at any time during the lifetime of the business activity.

Whenever a participant is registered with a business activity, it receives a handle on the coordinator. This handle is an instance of interface com.arjuna.wst11.BAParticipantManager with the methods listed in *BAParticipantManager Methods*.

BAParticipantManager Methods

`exit`
: The participant uses the method `exit` to inform the coordinator that is has left the activity. It will not be informed when and how the business activity terminates. This method may only be invoked while the participant is in the `active` state (or the `completing` state, in the case of a participant registered for the `ParticipantCompletion` protocol). If it is called when

the participant is in any other state, a `WrongStateException` error is thrown. An `exit` does not stop the activity as a whole from subsequently being closed or canceled/compensated, but only ensures that the exited participant is no longer involved in completion, close or compensation of the activity.

`completed`
: The participant has completed its work, but wishes to continue in the business activity, so that it will eventually be informed when, and how, the activity terminates. The participant may later be asked to compensate for the work it has done or learn that the activity has been closed.

`fault`
: The participant encountered an error during normal activation and has done whatever it can to compensate the activity. The `fault` method places the business activity into a mandatory `cancel-only` mode. The faulted participant is no longer involved in completion, close or compensation of the activity.

6.2. Participant Creation and Deployment

The participant provides the plumbing that drives the transactional aspects of the service. This section discusses the specifics of Participant programming and usage.

6.2.1. Implementing Participants

Implementing a participant is a relatively straightforward task. However, depending on the complexity of the transactional infrastructure that the participant needs to manage, the task can vary greatly in complexity and scope. Your implementation needs to implement one of the interfaces found under com.arjuna.wst11.

Note

The corresponding participant interfaces used in the 1.0 protocol implementation are located in package `com.arjuna.wst`.

6.2.2. Deploying Participants

Transactional web services and transactional clients are deployed by placing them in the application server deploy directory alongside the XTS service archive (SAR). The SAR exports all the client and web service API classes needed to manage transactions and enroll and manage participant web services. It provides implementations of all the WS-C and WS-T coordination services, not just the coordinator services. In particular, it exposes the client and web service participant endpoints which are needed to receive incoming messages originating from the coordinator.

Normally, a transactional application client and the transaction web service it invokes will be deployed in different application servers. As long as the XTS SAR is deployed to each of these

containers XTS will transparently route coordination messages from clients or web services to their coordinator and vice versa. When the the client begins a transaction by default it creates a context using the coordination services in its local container. The context holds a reference to the local Registration Service which means that any web services enlisted in the transaction enrol with the cooridnation services in the same container."

The coordinator does not need to reside in the same container as the client application. By configuring the client deployment appropriately it is possible to use the coordinator services co-located with one of the web services or even to use services deployed in a separate, dedicated container. See Chapter 8 Stand-Alone Coordination for details of how to configure a coordinator located in a different container to the client.

Warning

In previous releases, XTS applications were deployed using the appropriate XTS and Transaction Manager `.jar`, `.war`, and configuration files bundled with the application. This deployment method is no longer supported in the JBoss Application Server.

During JBoss Application Server startup, you should only deploy a transactional web service or transactional client after the XTS services are available. Declare this dependency in a `jboss-beans.xml` file located in the `META-INF` directory of the web service or client deployment. *Example 6.1, "Example jboss-beans.xml"* shows one way of declaring this dependency.

After the XTS service starts, it creates an instance of the application class `org.my.ServiceBean` by calling its `start` method. During JBoss Application Server shutdown, XTS stops the same instance by calling its `stop` method, prior to shutting down the XTS Service.

Example 6.1. Example `jboss-beans.xml`

```
+<!-- example jboss-beans.xml file declaring dependency on XTS service -->
+<?xml version="1.0" encoding="UTF-8"?>
+<deployment xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
+  xsi:schemaLocation="urn:jboss:bean-deployer bean-deployer_1_0.xsd"
+  xmlns="urn:jboss:bean-deployer">
+   <!-- bean class should implement start() and stop() methods -->
+   <bean name="MyService class=org.my.ServiceBean>
+       <depends>jboss:service=XTSService</depends>
+   </bean>
+</deployment>
```

The XTS API

This chapter discusses the XTS API. You can use this information to write client and server applications which consume transactional Web Services and coordinate back-end systems.

7.1. API for the Atomic Transaction Protocol

7.1.1. Vote

During the two-phase commit protocol, a participant is asked to vote on whether it can prepare to confirm the work that it controls. It must return an instance of one of the subtypes of `com.arjuna.wst11.Vote` listed in *Subclasses of com.arjuna.wst11.Vote*.

Subclasses of `com.arjuna.wst11.Vote`

Prepared
: Indicates that the participant can can prepare if the coordinator requests it. Nothing has been committed, because the participant does not know the final outcome of the transaction.

Aborted
: The participant cannot prepare, and has rolled back. The participant should not expect to get a second phase message.

ReadOnly
: The participant has not made any changes to state, and it does not need to know the final outcome of the transaction. Essentially the participant is resigning from the transaction.

Example 7.1. Example Implementation of 2PC Participant's `prepare` Method

```
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();
  }
}
```

7.1.2. TXContext

com.arjuna.mw.wst11.TxContext is an opaque representation of a transaction context. It returns one of two possible values, as listed in *TxContext Return Values*.

TxContext Return Values

valid
: Indicates whether the contents are valid.

equals
: Can be used to compare two instances for equality.

Note

The corresponding participant interfaces used in the 1.0 protocol implementation are located in package com.arjuna.wst.

7.1.3. UserTransaction

`com.arjuna.mw.wst11.UserTransaction` is the class that clients typically employ. Before a client can begin a new atomic transaction, it must first obtain a `UserTransaction` from the `UserTransactionFactory`. This class isolates the user from the underlying protocol-specific aspects of the XTS implementation. A `UserTransaction` does not represent a specific transaction. Instead, it provides access to an implicit per-thread transaction context, similar to the `UserTransaction` in the JTA specification. All of the `UserTransaction` methods implicitly act on the current thread of control.

`userTransaction` Methods

begin
: Used to begin a new transaction and associate it with the invoking thread.

Parameters

timeout
: This optional parameter, measured in milliseconds, specifies a time interval after which the newly created transaction may be automatically rolled back by the coordinator

Exceptions

`WrongStateException`
: A transaction is already associated with the thread.

commit
: Volatile2PC and Durable2PC participants enrolled in the transaction are requested first to prepare and then to commit their changes. If any of the participants fails to prepare in the first phase then all other participants are requested to abort.

Exceptions

`UnknownTransactionException`
: No transaction is associated with the invoking thread.

`TransactionRolledBackException`
: The transaction was rolled back either because of a timeout or because a participant was unable to commit.

rollback
: Terminates the transaction. Upon completion, the `rollback` method disassociates the transaction from the current leaving it unassociated with any transactions.

Exceptions

`UnknownTransactionException`
: No transaction is associated with the invoking thread.

7.1.4. UserTransactionFactory

Call the `getUserTransaction` method to obtain a *Section 7.1.3, "UserTransaction"* instance from a `UserTransactionFactory`.

7.1.5. TransactionManager

Defines the interaction between a transactional web service and the underlying transaction service implementation. A `TransactionManager` does not represent a specific transaction. Instead, it provides access to an implicit per-thread transaction context.

Methods

`currentTransaction`
: Returns a `TxContext` for the current transaction, or null if there is no context. Use the `currentTransaction` method to determine whether a web service has been invoked from within an existing transaction. You can also use the returned value to enable multiple threads

to execute within the scope of the same transaction. Calling the `currentTransaction` method does not disassociate the current thread from the transaction.

`suspend`
Dissociates a thread from any transaction. This enables a thread to do work that is not associated with a specific transaction.

The `suspend` method returns a `TxContext` instance, which is a handle on the transaction.

`resume`
Associates or re-associates a thread with a transaction, using its `TxContext`. Prior to association or 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. In this way, the result is the same as if the `suspend` method were used instead.

Parameters

txContext
A TxContext instance as return by `suspend`, identifying the transaction to be resumed.

Exceptions

`UnknownTransactionException`
The transaction referred to by the `TxContext` is invalid in the scope of the invoking thread.

`enlistForVolitaleTwoPhase`
Enroll the specified participant with the current transaction, causing it to participate in the Volatile2PC protocol. You must pass a unique identifier for the participant.

Parameters

participant
An implementation of interface Volatile2PCParticipant whose prepare, commit and abort methods are called when the corresponding coordinator message is received.

id
A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

UnknownTransactionException
No transaction is associated with the invoking thread.

`WrongStateException`
The transaction is not in a state that allows participants to be enrolled. For instance, it may be in the process of terminating.

`enlistForDurableTwoPhase`
: Enroll the specified participant with the current transaction, causing it to participate in the Durable2PC protocol. You must pass a unique identifier for the participant.

Exceptions

UnknownTransactionException
: No transaction is associated with the invoking thread.

`WrongStateException`
: The transaction is not in a state that allows participants to be enrolled. For instance, it may be in the process of terminating.

7.1.6. TransactionManagerFactory

Use the `getTransactionManager` method to obtain a *Section 7.1.5, "TransactionManager"* from a `TransactionManagerFactory`.

7.2. API for the Business Activity Protocol

7.2.1. Compatibility

Previous implementations of XTS locate the Business Activity Protocol classes in the com.arjuna.mw.wst package. In the current implementation, these classes are located in the com.arjuna.mw.wst11 package.

7.2.2. UserBusinessActivity

`com.arjuna.wst11.UserBusinessActivity` is the class that most clients employ. A client begins a new business activity by first obtaining a `UserBusinessActivity` from the `UserBusinessActivityFactory`. This class isolates them from the underlying protocol-specific aspects of the XTS implementation. A UserBusinessActivity does not represent a specific business activity. Instead, it provides access to an implicit per-thread activity. Therefore, all of the `UserBusinessActivity` methods implicitly act on the current thread of control.

Methods

`begin`
: Begins a new activity, associating it with the invoking thread.

Parameters

timeout
: The interval, in milliseconds, after which an activity times out. Optional.

Exceptions

`WrongStateException`
: The thread is already associated with a business activity.

`close`
: First, all Coordinator Completion participants enlisted in the activity are requested to complete the activity. Next all participants, whether they enlisted for Coordinator or Participant Completion, are requested to close the activity. If any of the Coordinator Completion participants fails to complete at the first stage then all completed participants are asked to compensate the activity while any remaining uncompleted participants are requested to cancel the activity.

Exceptions

UnknownTransactionException
: No activity is associated with the invoking thread.

TransactionRolledBackException
: The activity has been cancelled because one of the Coordinator Completion participants failed to complete. This exception may also thrown if one of the Participant Completion participants has not completed before the client calls close.

cancel
: Terminates the business activity. All Participant Completion participants enlisted in the activity which have already completed are requested to compensate the activity. All uncompleted Participant Completion participants and all Coordinator Completion participants are requested to cancel the activity.

Exceptions

`UnknownTransactionException`
: No activity is associated with the invoking thread. Any participants that previous completed are directed to compensate their work.

7.2.3. UserBusinessActivityFactory

Use the `getuserbusinessActivity` method to obtain a *Section 7.2.2, "UserBusinessActivity"* instance from a `userBusinessActivityFactory`.

7.2.4. BusinessActivityManager

com.arjuna.mw.wst11.BusinessActivityManager is the class that web services typically employ. Defines how a web service interacts with the underlying business activity service implementation. A `BusinessActivityManager` does not represent a specific activity. Instead, it provides access to an implicit per-thread activity.

Methods

`currentTransaction`

Returns the `TxContext` for the current business activity, or `NULL` if there is no `TxContext`. The returned value can be used to enable multiple threads to execute within the scope of the same business activity. Calling the `currenTransaction` method does not dissociate the current thread from its activity.

`suspend`

Dissociates a thread from any current business activity, so that it can perform work not associated with a specific activity. The `suspend` method returns a `TxContext` instance, which is a handle on the activity. The thread is then no longer associated with any activity.

`resume`

Associates or re-associates a thread with a business activity, using its `TxContext`. Before associating or re-associating the thread, it is disassociated from any business activity with which it is currently associated. If the `TxContext` is `NULL`, the thread is disassociated with all business activities, as though the `suspend` method were called.

Parameters

txContext

A TxContext instance as returned by `suspend`, identifying the transaction to be resumed.

Exceptions

`UnknownTransactionException`

The business activity to which the `TxContext` refers is invalid in the scope of the invoking thread.

`enlistForBusinessAgreementWithParticipantCompletion`

Enroll the specified participant with current business activity, causing it to participate in the `BusinessAgreementWithParticipantCompletion` protocol. A unique identifier for the participant is also required.

The return value is an instance of BAParticipantManager which can be used to notify the coordinator of changes in the participant state. In particular, since the participant is enlisted for the Participant Completion protcol it is expected to call the completed method of this returned instance when it has completed all the work it expects to do in this activity and has made all its changes permanent. Alternatively, if the participant does not need to perform any compensation actions should some other participant fail it can leave the activity by calling the exit method of the returned BAParticipantManager instance.

Parameters

participant

An implementation of interface `BusinessAgreementWithParticipantCompletionParticipant` whose `close`, `cancel`,

and `compensate` methods are called when the corresponding coordinator message is received.

id
: A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

UnknownTransactionException
: No transaction is associated with the invoking thread.

WrongStateException
: The transaction is not in a state where new participants may be enrolled, as when it is terminating.

`enlistForBusinessAgreementWithCoordinatorCompletion`
: Enroll the specified participant with current activity, causing it to participate in the `BusinessAgreementWithCoordinatorCompletion` protocol. A unique identifier for the participant is also required.

  The return value is an instance of `BAParticipantManager` which can be used to notify the coordinator of changes in the participant state. Note that in this case it is an error to call the `completed` method of this returned instance. With the Coordinator Completion protocol the participant is expected to wait until its `completed` method is called before it makes all its changes permanent. Alternatively, if the participant determiens that it has no changes to make, it can leave the activity by calling the `exit` method of the returned `BAParticipantManager` instance.

Parameters

participant
: An implementation of interface BusinessAgreementWithCoordinatorCompletionParticipant whose completed, close, cancel and compensate methods are called when the corresponding coordinator message is received.

id
: A unique identifier for the participant. The value of this String should differ for each enlisted participant. It should also be possible for a given identifier to determine that the participant belongs to the enlisting web service rather than some other web service deployed to the same container.

Exceptions

UnknownTransactionException
: No transaction is associated with the invoking thread.

WrongStateException

The transaction is not in a state where new participants may be enrolled, as when it is terminating.

7.2.5. BusinessActivityManagerFactory

Use the `getBusinessActivityManager` method to obtain a *Section 7.2.4, "BusinessActivityManager"* instance from a `BusinessActivityManagerFactory`.

Stand-Alone Coordination

8.1. Introduction

The XTS service is deployed as a JBoss service archive (SAR). The version of the service archive provided with the Transaction Service implements version 1.1 of the WS-C, WS-AT and WS-BA services. You can rebuild the XTS service archive to include both the 1.0 and 1.1 implementations and deploy them side by side. See the service archive build script for for further details.

The release service archive obtains coordination contexts from the Activation Coordinator service running on the deployed host. Therefore, WS-AT transactions or WS-BA activities created by a locally-deployed client application are supplied with a context which identifies the Registration Service running on the client's machine. Any Web Services invoked by the client are coordinated by the Transaction Protocol services running on the client's host. This is the case whether the Web Services are running locally or remotely. Such a configuration is called *local coordination*.

You can reconfigure this setting globally for all clients, causing context creation requests to be redirected to an Activation Coordinator Service running on a remote host. Normally, the rest of the coordination process is executed from the remote host. This configuration is called *stand-alone coordination*.

Reasons for Choosing a Stand-Alone Coordinator

- Efficiency: if a client application invokes Web Services on a remote JBoss Application Server, coordinating the transaction from the remote server might be more efficient, since the protocol-specific messages between the coordinator and the participants do not need to travel over the network.
- Reliability: if the coordinator service runs on a dedicated host, there is no danger of failing applications or services affecting the coordinator and causing failures for unrelated transactions.
- A third reason might be to use a coordination service provided by a third party vendor.

8.2. Configuring the Activation Coordinator

The XTS service archive used to deploy XTS to the JBoss Application Server includes a bean configuration file, `xts-jboss-beans.xml`, in its `META-INF` directory. This file specifies configuration values which the JBoss Application Server injects into the beans, and which define the XTS runtime configuration when it starts the XTS service. The location of the XTS coordinator is defined by values injected into the `WSCEnvironmentBean`. *Example 8.1, "Example xts-jboss-beans.xml configuration settings"* shows a fragment of this file which details serveral possible configuration options.

Example 8.1. Example `xts-jboss-beans.xml` configuration settings

```
<bean name="XTS:WSCEnvironmentBean" class="org.jboss.jbossts.xts.environment.WSCEnvironmentBean
```

```
  <constructor factoryClass="org.jboss.jbossts.xts.environment.XTSPropertyManager"
               factoryMethod="getWSCEnvironmentBean"/>
  <!-- we need the bind address and port from jboss web -->
  <depends>jboss.web:service=WebServer</depends>
  <depends>jboss:service=TransactionManager</depends>
  . . .
  <!--
      if you want to use a coordinator running in a remote JVM then you can
      simply configure the URL. This will also be necessary if you are using
      a non-JBoss coordination service.

      <property name="coordinatorURL11">
         <value>http://10.0.1.99:8080/ws-c11/ActivationService</value>
      </property>
  -->
  <!--
      if you are using a remote JBoss XTS coordinator you can just redefine
      the scheme, address, port or path to the desired value and the ones
      left undefined will be defaulted to use the standard XTS coordinator
      URL elements. So, for example if your XTS coordinator services is
      deployed in another AS on host myhost.myorg.com you only need to define
      property coordinatorAddress11 to have value myhost.myorg.com and the
      coordinator address used by clients will be
      http://myhost.myorg.com:8080/ws-c11/ActivationService n.b. if the remote
      machine is using JBoss XTS then you won't want to redefine the port
     unless you have monkeyed around with the port config in the remote AS. also
      you won't need to change the path unless you have tweaked the deployment
      descriptor to relocate the XTS services.

      <property name="coordinatorScheme11">
         <value>http</value>
      </property>
      <property name="coordinatorAddress11">
         <value>10.0.1.99</value>
      </property>
      <property name="coordinatorPort11">
         <value>9191</value>
      </property>
      <property name="coordinatorPath11">
         <value>ws-c11/ActivationService</value>
      </property>
  -->
</bean>
```

The simplest way to configure a stand-alone coordinator is to provide a complete URL for the remote coordinator. The example shows how the bean property with name `coordinatorURL11` would be configured with value `http://10.0.1.99:8080/ws-c11/ActivationService`.

You can also specify the individual elements of the URL using the properties `coordinatorScheme11`, `coordinatorAddress11`, and so forth. These values only apply when the `coordinatorURL11` is not set. The URL is constructed by combining the specified values with default values for any missing elements. This is particularly useful for two specific use cases.

1. The first case is where the client is expected to use an XTS coordinator deployed in another JBoss Application Server. If, for example, this JBoss Application Server is bound to address `10.0.1.99`, setting property `coordinatorAddress11` to `10.0.1.99` is normally all that is required to configure the coordinator URL to idenitfy the remote JBoss Application Server's coordination service. If the Web service on the remote JBoss Application Server were reset to`9090` then it would also be necessary to set property `coordinatorPort11` to this value.

2. The second common use case is where communications between client and coordinator, and between participant and coordinator, must use secure connections. If property `coordinatorScheme11` is set to value `https`, the client's request to begin a transaction is sent to the coordinator service over a secure https connection. The XTS coordinator and participant services will ensure that all subsequent communications between coordinator and client or coordinator and web services also employ secure https connections. Note that this requires configuring the trust stores in the JBoss Application Server running the client, coordinator and participant web services with appropriate trust certificates.

If none of the above properties is specified, the coordinator URL uses the `http` scheme. The coordinator address and port are obtained from the host address and port configured for the JBoss Web service. These default to `localhost` and `8080`, respectively. The URL path takes the value shown in *Example 8.1, "Example xts-jboss-beans.xml configuration settings"*.

You can configure the bean properties defined above by setting System properties on the Java command line. This is useful in pre-deployment testing. So, for example, command line option `-Dorg.jboss.jbossts.xts11.coordinator.address=10.0.1.99` resets property `coordinatorAddress11`. System properties do not override property values configured in the configuration file. The following table identifies the System properties which can be configured.

Note

The property names have been abbreviated in order to fit into the table. They should each start with prefix `org.jboss.jbossts.xts11.coordinator`.

Table 8.1. Command-Line Options Passed with the `-D` Parameter, Ordered by Priority

Category	Property	Format
Absolute URL	`...coordinatorURL`	*http://coord.host:coord.port/ws-c11/ActivationService*

Category	Property	Format

Coordinator Scheme, Host, Port, and Path	`...coordinator.scheme`
	
	`...coordinator.address`
	
	`...coordinator.port`
	
	`...coordinator.path`
		`http`
		
		`server.bind.address`
		
		`jboss.web.bind.port`

Participant Crash Recovery

A key requirement of a transaction service is to be resilient to a system crash by a host running a participant, as well as the host running the transaction coordination services. Crashes which happen before a transaction terminates or before a business activity completes are relatively easy to accommodate. The transaction service and participants can adopt a *presumed abort* policy.

Procedure 9.1. Presumed Abort Policy

1. If the coordinator crashes, it can assume that any transaction it does not know about is invalid, and reject a participant request which refers to such a transaction.

2. If the participant crashes, 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.

Crash recovery is more complex if the crash happens during a transaction commit operation, or between completing and closing a business activity. The transaction service must ensure as far as possible that participants arrive at a consistent outcome for the transaction.

WS-AT Transaction
: The transaction needs to commit all provisional changes or roll them all back to the state before the transaction started.

WS-Business Activity Transaction
: All participants need to close the activity or cancel the activity, and run any required compensating actions.

On the rare occasions where such a consensus cannot be reached, the transaction service must log and report transaction failures.

XTS includes support for automatic recovery of WS-AT and WS-BA transactions, if either or both of the coordinator and participant hosts crashes. The XTS recovery manager begins execution on coordinator and participant hosts when the XTS service restarts. On a coordinator host, the recovery manager detects any WS-AT transactions which have prepared but not committed, as well as any WS-BA transactions which have completed but not yet closed. It ensures that all their participants are rolled forward in the first case, or closed in the second.

On a participant host, the recovery manager detects any prepared WS-AT participants which have not responded to a transaction rollback, and any completed WS-BA participants which have not yet responded to an activity cancel request, and ensures that the former are rolled back and the latter are compensated. The recovery service also allows for recovery of subordinate WS-AT transactions and their participants if a crash occurs on a host where an interposed WS-AT coordinator has been employed.

9.1. WS-AT Recovery

9.1.1. WS-AT Coordinator Crash Recovery

The WS-AT coordination service tracks the status of each participant in a transaction as the transaction progresses through its two-phase commit. When all participants have been sent a `prepare` message and have responded with a `prepared` message, the coordinator writes a log record storing each participant's details, indicating that the transaction is ready to complete. If the coordinator service crashes after this point has been reached, completion of the two-phase commit protocol is still guaranteed, by reading the log file after reboot and sending a `commit` message to each participant. Once all participants have responded to the `commit` with a `committed` message, the coordinator can safely delete the log entry.

Since the `prepared` messages returned by the participants imply that they are ready to commit their provisional changes and make them permanent, this type of recovery is safe. Additionally, the coordinator does not need to account for any commit messages which may have been sent before the crash, or resend messages if it crashes several times. The XTS participant implementation is resilient to redelivery of the `commit` messages. If the participant has implemented the recovery functions described in *Section 9.1.2.1, "WS-AT Participant Crash Recovery APIs"*, the coordinator can guarantee delivery of `commit` messages if both it crashes, and one or more of the participant service hosts also crash, at the same time.

If the coordination service crashes before the `prepare` phase completes, the presumed abort protocol ensures that participants are rolled back. After system restart, the coordination service has the information about about all the transactions which could have entered the `commit` phase before the reboot, since they have entries in the log. It also knows about any active transactions started after the reboot. If a participant is waiting for a response, after sending its `prepared` message, it automatically re sends the `prepared` message at regular intervals. When the coordinator detects a transaction which is not active and has no entry in the log file after the reboot, it instructs the participant to abort, ensuring that the web service gets a chance to roll back any provisional state changes it made on behalf of the transaction.

A web service may decide to unilaterally commit or roll back provisional changes associated with a given participant, if configured to time out after a specified length of time without a response. In this situation, the the web service should record this action and log a message to persistent storage. When the participant receives a request to commit or roll back, it should throw an exception if its unilateral decision action does not match the requested action. The coordinator detects the exception and logs a message marking the outcome as heuristic. It also saves the state of the transaction permanently in the transaction log, to be inspected and reconciled by an administrator.

9.1.2. 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 assume that the transaction has aborted, and 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 to commit or roll back the transaction. The specific information it needs to save is dependent on the implementation and business logic of the Web Service. However, the participant must save this state before returning a `Prepared` vote from the `prepare` call. If the participant 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 cooperate with the Web service implementation to facilitate 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 information 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, a participant instance is still necessary for the commit or rollback process to continue.

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 instance, which can continue the `commit` process to completion, or roll it back if some other Web Service fails to `prepare`. This information might be as simple as a String key which the participant can use to locate the data it made persistent before returning its Prepared vote. It may be as complex as a serialized object tree containing the original participant instance and 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. In the event of a crash, the participant recovery state is retrieved from the log and passed to the Web Service which created it. The Web Service uses this state to create a new participant, which the XTS implementation uses to drive the transaction to completion. Log records are only deleted after the participant's `commit` or `rollback` method is called.

Warning

If a crash happens just before or just after a `commit` method is called, a `commit` or `rollback` method may be called twice.

9.1.2.1. WS-AT Participant Crash Recovery APIs

9.1.2.1.1. Saving Participant Recovery State

When a Business Activity participant web service completes its work, it may want to save the information which will be required later to close or compensate actions performed during the

activity. The XTS implementation automatically acquires this information from the participant as part of the completion process and writes it to a participant log record. This ensures that the information can be restored and used to recreate a copy of the participant even if the web service container crashes between the complete and close or compensate operations.

For a Participant Completion participant, this information is acquired when the web service invokes the `completed` method of the `BAParticipantManager` instance returned from the call which enlisted the participant. For a Coordinator Completion participant this occurs immediately after the call to it's `completed` method returns. This assumes that the `completed` method does not throw an exception or call the participant manager's `cannotComplete` or `fail` method.

A participant may signal that it is capable of performing recovery processing, by implementing the `java.lang.Serializable` interface. An alternative is to implement the *Example 9.1, "PersistableATParticipant Interface"*.

Example 9.1. `PersistableATParticipant` Interface

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

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

If neither of these APIs is implemented, the XTS implementation logs a warning message and proceeds without saving any recovery state. In the event of a crash on the host machine for the Web service during commit, the transaction cannot be recovered and a heuristic outcome may occur. This outcome is logged on the host running the coordinator services.

9.1.2.1.2. 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 to participate in recovery processing. Registration is performed using class `XTSATRecoveryManager` defined in package org.jboss.jbossts.xts.recovery.participant.at.

Example 9.2. Registering for Recovery

```
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 `XTSBARecoveryModule` in package org.jboss.jbossts.xts.recovery.participant.ba, as an argument to the `register` and `unregister` calls. This instance identifies saved participant recovery records and recreates new, recovered participant instances:

Example 9.3. `XTSBARecoveryModule` Interface

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

If a participant's recovery state was saved using serialization, the recovery module's `deserialize` method is called to recreate the participant. Normally, the recovery module is required to read, cast, and return an object from the supplied input stream. If a participant's recovery state was saved using the `PersistableATParticipant` interface, the recovery module's `recreate` method is called to recreate the participant from the byte array it provided when the state was saved.

The XTS implementation cannot identify which participants belong to which recovery modules. A module only needs to return a participant instance if the recovery state belongs to the module's Web service. If the participant was created by another Web service, the module should return `null`. The participant identifier, which is supplied as argument to the `deserialize` or `recreate` method, is the identifier used by the Web service when the original participant was enlisted in the transaction. Web Services participating in recovery processing should ensure that participant identifiers are unique per service. If a module recognizes that a participant identifier belongs to its Web service, but cannot recreate the participant, it should throw an exception. This situation might arise if the service cannot associate the participant with any transactional information which is specific to the business logic.

Even if a module relies on serialization to create the participant recovery state saved by the XTS implementation, it still must be registered by the application. The `deserialization` operation must employ a class loader capable of loading classes specific to the Web service. XTS fulfills this requirement by devolving responsibility for the `deserialize` operation to the recovery module.

9.2. WS-BA Recovery

9.2.1. WS-BA Coordinator Crash Recovery

The WS-BA coordination service implementation tracks the status of each participant in an activity as the activity progresses through completion and closure. A transition point occurs during closure, once all `CoordinatorCompletion` participants receive a `complete` message and respond with a `completed` message. At this point, all `ParticipantCompletion` participants should have sent a `completed` message. The coordinator writes a log record storing the details of each participant, and indicating that the transaction is ready to close. If the coordinator service crashes after the log record is written, the `close` operation is still guaranteed to be successful. The coordinator checks the log after the system reboots and re sends a `close` message to all participants. After all participants respond to the `close` with a `closed` message, the coordinator can safely delete the log entry.

The coordinator does not need to account for any `close` messages sent before the crash, nor resend messages if it crashes several times. The XTS participant implementation is resilient to redelivery of `close` messages. Assuming that the participant has implemented the recovery functions described below, the coordinator can even guarantee delivery of `close` messages if both it, and one or more of the participant service hosts, crash simultaneously.

If the coordination service crashes before it has written the log record, it does not need to explicitly compensate any completed participants. The presumed abort protocol ensures that all completed participants are eventually sent a `compensate` message. Recovery must be initiated from the participant side.

A log record does not need to be written when an activity is being canceled. If a participant does not respond to a `cancel` or `compensate` request, the coordinator logs a warning and continues. The combination of the presumed abort protocol and participant-led recovery ensures that all participants eventually get canceled or compensated, as appropriate, even if the participant host crashes.

If a completed participant does not detect a response from its coordinator after resending its `completed` response a suitable number of times, it switches to sending `getstatus` messages, to determine whether the coordinator still knows about it. If a crash occurs before writing the log record, the coordinator has no record of the participant when the coordinator restarts, and the `getstatus` request returns a fault. The participant recovery manager automatically compensates the participant in this situation, just as if the activity had been canceled by the client.

After a participant crash, the participant recovery manager detects the log entries for each completed participant. It sends `getstatus` messages to each participant's coordinator host, to determine whether the activity still exists. If the coordinator has not crashed and the activity is still running, the participant switches back to resending `completed` messages, and waits for a `close` or `compensate` response. If the coordinator has also crashed or the activity has been canceled, the participant is automatically canceled.

9.2.2. WS-BA Participant Crash Recovery APIs

9.2.2.1. Saving Participant Recovery State

A participant may signal that it is capable of performing recovery processing, by implementing the `java.lang.Serializable` interface. An alternative is to implement the *Example 9.4, "PersistableBAParticipant Interface"*.

Example 9.4. `PersistableBAParticipant` Interface

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

If a participant implements the `Serializable` interface, the XTS participant services implementation uses the serialization API to create a version of the participant which can be appended to the participant log entry. If the participant implements the `PersistableBAParticipant`, the XTS participant services implementation call the `getRecoveryState` method to obtain the state, which is appended to the participant log entry.

If neither of these APIs is implemented, the XTS implementation logs a warning message and proceeds without saving any recovery state. If the Web service's host machine crashes while the activity is being closed, the activity cannot be recovered and a heuristic outcome will probably be logged on the coordinator's host machine. If the activity is canceled, the participant is not compensated and the coordinator host machine may log a heuristic outcome for the activity.

9.2.2.2. Recovering Participants at Reboot

A Web service must register with the XTS implementation when it is deployed, and unregister when it is undeployed, so it can take part in recovery processing.

Registration is performed using the `XTSBARecoveryManager`, defined in the org.jboss.jbossts.xts.recovery.participant.ba package.

Example 9.5. `XTSBARecoveryManager` Class

```
public abstract class XTSBARecoveryManager {
    . . .
        public static XTSBARecoveryManager getRecoveryManager() ;
    public void registerRecoveryModule(XTSBARecoveryModule module);
    public abstract void unregisterRecoveryModule(XTSBARecoveryModule module)
        throws NoSuchElementException;
    . . .
```

```
}
```

The Web service must provide an implementation of the `XTSBARecoveryModule` in the org.jboss.jbossts.xts.recovery.participant.ba, as an argument to the `register` and `unregister` calls. This instance identifies saved participant recovery records and recreates new, recovered participant instances:

Example 9.6. `XTSBARecoveryModule` Interface

```
public interface XTSBARecoveryModule
{
    public BusinessAgreementWithParticipantCompletionParticipant
        deserializeParticipantCompletionParticipant(String id,
                                                    ObjectInputStream stream)
        throws Exception;
    public BusinessAgreementWithParticipantCompletionParticipant
        recreateParticipantCompletionParticipant(String id,
                                                 byte[] recoveryState)
        throws Exception;
    public BusinessAgreementWithCoordinatorCompletionParticipant
        deserializeCoordinatorCompletionParticipant(String id,
                                                    ObjectInputStream stream)
        throws Exception;
    public BusinessAgreementWithCoordinatorCompletionParticipant
        recreateCoordinatorCompletionParticipant(String id,
                                                 byte[] recoveryState)
        throws Exception;
    public void endScan();
}
```

If a participant's recovery state was saved using serialization, one of the recovery module's `deserialize` methods is called, so that it can recreate the participant. Which method to use depends on whether the saved participant implemented the `ParticipantCompletion` protocol or the `CoordinatorCompletion` protocol. Normally, the recovery module reads, casts and returns an object from the supplied input stream. If a participant's recovery state was saved using the `PersistableBAParticipant` interface, one of the recovery module's `recreate` methods is called, so that it can recreate the participant from the byte array provided when the state was saved. The method to use depends on which protocol the saved participant implemented.

The XTS implementation does not track 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, the module should return `null`. The participant identifier supplied as an argument to the `deserialize` or `recreate` calls is the identifier used by the Web service when the original participant was enlisted in the transaction. Web Services which participate in recovery processing should ensure

that the participant identifiers they employ are unique per service. If a module recognizes a participant identifier as belonging to its Web service, but cannot recreate the participant, it throws an exception. This situation might arise if the service cannot associate the participant with any transactional information specific to business logic.

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 capable of loading Web service-specific classes. The XTS implementation achieves this by delegating responsibility for the `deserialize` operation to the recovery module.

9.2.2.3. Securing Web Service State Changes

When a BA participant completes, it is expected to commit changes to the web service state made during the activity. The web service usually also needs to persist these changes to a local storage device. This leaves open a window where the persisted changes may not be guarded with the necessary compensation information. The web service container may crash after the changes to the service state have been written but before the XTS implementation is able to acquire the recovery state and write a recovery log record for the participant. Participants may close this window by employing a two phase update to the local store used to persist the web service state.

A participant which needs to persist changes to local web service state should implement interface `ConfirmCompletedParticipant` in package com.arjuna.wst11. This signals to the XTS implementation that it expects confirmation after a successful write of the participant recovery record, allowing it to roll forward provisionally persisted changes to the web service state. Delivery of this confirmation can be guaranteed even if the web service container crashes after writing the participant log record. Conversely, if a recovery record cannot be written because of a fault or a crash prior to writing, the provisional changes can be guaranteed to be rolled back.

Example 9.7. `ConfirmCompletedParticipant` Interface

```
public interface ConfirmCompletedParticipant
{
    public void confirmCompleted(boolean confirmed);
}
```

When the participant is ready to complete, it should prepare its persistent changes by temporarily locking access to the relevant state in the local store and writing the changed data to disk, retaining both the old and new versions of the service state. For a Participant Completion participant, this prepare operation should be done just before calling the participant manager's `completed` method. For a Coordinator Completion participant, it should be done just before returning from the call to the participant's `completed` method. After writing the participant log record, the XTS implementation calls the participant's `confirmCompleted` method, providing value `true` as the argument. The participant should respond by installing the provisional state changes and releasing any locks. If the log record cannot be written, the XTS implementation calls the participant's

`confirmCompleted` method, providing value `false` as the argument. The participant should respond by restoring the original state values and releasing any locks.

If a crash occurs before the call to `confirmCompleted`, the application's recovery module can make sure that the provisional changes to the web service state are rolled forward or rolled back as appropriate. The web service must identify all provisional writes to persistent state before it starts serving new requests or processing recovered participants. It must reobtain any locks required to ensure that the state is not changed by new transactions. When the recovery module recovers a participant from the log, its compensation information is available. If the participant still has prepared changes, the recovery code must call `confirmCompleted`, passing value true. This allows the participant to finish the `complete` operation. The XTS implementation then forwards a `completed` message to the coordinator, ensuring that the participant is subsequently notified either to close or to compensate. At the end of the first recovery scan, the recovery module may find some prepared changes on disk which are still unaccounted for. This means that the participant recovery record is not available. The recovery module should restore the original state values and release any locks. The XTS implementation responds to coordinator requests regarding the participant with an `unknown participant` fault, forcing the activity as a whole to be rolled back.

Web Service Component

To configure the demo application of the Web Services transactions component and standalone coordinator, edit the appropriate `build.xml` file before running `ant`. Consult the trail map accompanying these components for details.

When running within JBoss Application Server, it is recommended to use the `all` server profile. Specify the profile with the *`-c`* switch: `run.sh -c all` in Linux, or `run.bat -c all` in Microsoft Windows.

Web Service Transaction Service (XTS) Management

The basic building blocks of a transactional Web Services application include the application itself, the Web services that the application consumes, the Transaction Manager, and the transaction participants which support those Web services. Although it is likely that different developers will be responsible for each piece, the concepts are presented here so that you can see the whole picture. Often, developers produce services, or applications that consume services, and system administrators run the transaction-management infrastructure.

11.1. Transaction manager overview

The transaction manager is a Web service which coordinates JBossTS transactions. It is the only software component in JBossTS that is designed to be run directly as a network service, rather than to support end-user code. The transaction manager runs as a JAXM request/response Web service.

Note

When starting up an application server instance that has JBossTS transaction manager deployed within it, you may see various “error” messages in the console or log. For example 16:53:38,850 ERROR [STDERR] Message Listener Service: started, message listener jndi name activationcoordinator". These are for information purposes only and are not actual errors.

11.2. Configuring the transaction manager

You can configure the Transaction Manager and related infrastructure by using three properties files: `wscf.xml`, `wst.xml`, and `wstx.xml`. Each file is located in the `conf/` directory. Both the demo application and the stand-alone module rely on them for configuration.

For the most part the default values in these files are suitable. However, the `ObjectStoreEnvironmentBean.objectStoreDir` property, which determines the location of the persistent store used to record transaction state, should be modified to suit your environment. The default value is `C:/temp/ObjectStore`. For production environments this directory should reside on fault-tolerant media such as a RAID array.

When an application uses a standalone coordinator, you must enable and modify two additional properties in `wstx.xml`. These properties are `com.arjuna.mw.wst.coordinatorURL` and `com.arjuna.mw.wst.terminatorURL`. They specify the URLs needed by client application to contact the standalone coordinator, and need to specify the correct hostname and port for the stand-alone server.

JBossTS is extremely modular. To allow flexible deployment of individual components, the same property values sometimes need to appear in more than one configuration file. Except in special circumstances, maintain consistent values for properties that are defined in more than one file.

11.3. Deploying the transaction manager

The JBossTS XTS component consists of a number of `.jar` files containing the application's class files, plus Web service (`.war`) files which expose the necessary services. These components are typically included in an application's Enterprise Archive (`.ear`) file during application development, as this simplifies deployment of the transaction infrastructure. For production, you can install the Transaction Manager as an application in its own right, allowing for centralized configuration and management at the server level, independent of specific applications. The demonstration application shipped with JBossTS provides a sample deployment descriptor illustrating how the Transaction Manager components can be included in an application.

JBossTS 4.x uses fixed endpoints for its underlying protocol communication. Therefore, problems may arise if you deploy multiple applications using JBossTS to the same server concurrently. If you need to deploy several transactional applications in the same server, you must deploy the Transaction Manger as a separate application, rather than embedding it within the development of individual applications.

The `coordinator/` directory in the JBossTS installation can assist in the configuration and deployment of a stand-alone transaction manager.

Procedure 11.1. Using the `coordinator/` diretory to configure and deploy a stand-alone transaction manager

1. Install JBossTS 4.15.
2. Use a separate application server installation for the coordinator. This installation can be on a separate machine. To set this up on JBoss Application Server, see *http://www.yorku.ca/dkha/jboss/docs/MultipleInstances.htm* for more information.
3. Install Ant 1.4 or later.

Warning

A separate application server installation must be used, separate from the one that clients and services are deployed into, to prevent conflicts between the various JBossTS components.

4. Edit the `build.xml` included in the `coordinator/` directory, to point to the application server installation where the transaction coordinator will be deployed and the location of the JBossTS installation. The files `ws-c_jaxm_web-app.xml` and `ws-t_jaxm_web-app.xml` in the `dd/`

subdirectory of `coordinator/` are the deployment descriptors for the WS-C and WS-T `war` files. These files contain templated URLs. During the build phase, ant will substitute the hostname and port values you specify in the `build.xml` into these files.

5. Run ant, with one of targets `deploy-weblogic`, `deploy-jboss`, or `deploy-webmethods`, to create and deploy a new coordinator into the correct application server installation.

6. Finally, point your client at the required coordinator. To do this, generate the demo application. specifying the port and hostname of the coordinator.

11.4. Deployment descriptors

In general, changing the contents of the various deployment descriptors used by JBossTS is not necessary. However, if you do need to modify them they are all included in the coordinator module.

Not all JBossTS components have ready access to the information in the deployment descriptors. Therefore, if you modify the JNDI names used by any of the WS-C or WS-T deployment descriptors, you may need to inform other JBossTS components at runtime, by setting an appropriate property in the `wstx.xml` configuration file.

Important

You need to prefix each property in this table with the string `com.arjuna.mw.wst..` The prefix has been removed for formatting reasons, and has been replaced by `....`

Table 11.1. Deployment descriptor values and properties

JNDI Name	Property
Activationrequester	...at.activationrequester
Activationcoordinator	...at.activationcoordinator
Completionparticipant	...at.completionparticipant
Registrationrequester	...at.registrationrequester
durable2pcdispatcher	...at.durable2pcdispatcher
durable2pcparticipant	...at.durable2pcparticipant
volatile2pcdispatcher	...at.volatile2pcdispatcher
volatile2pcparticipant	...at.volatile2pcparticipant
businessagreementwithparticipantcompletiondispatcher	...at.businessagreementwpcdispatcher
businessagreementwithparticipantcompletionparticipant	...at.businessagreementwpcparticipant
businessagreementwithcoordinatorcompletiondispatcher	...at.businessagreementwccdispatcher
Businessagreementwithcoordinatorcompletionparticipant	...at.businessagreementwccparticipant

Appendix A. Revision History

Revision History

Revision 0	Mon Jul 12 2010	MistyStanley-Jones<mstanley@redhat.com>

Initial creation of book by publican