JBoss Transactions 4.16.5.Final

Transactions Overview Guide

Mark Little

JBoss Transactions 4.16.5.Final Transactions Overview Guide

Author Mark Little mlittle@redhat.com

The Transactions Overview Guide contains information on how to use JBoss Transactions to develop applications that use transaction technology to manage business processes.

Preface

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*[1] 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

[1] https://fedorahosted.org/liberation-fonts/

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;
```

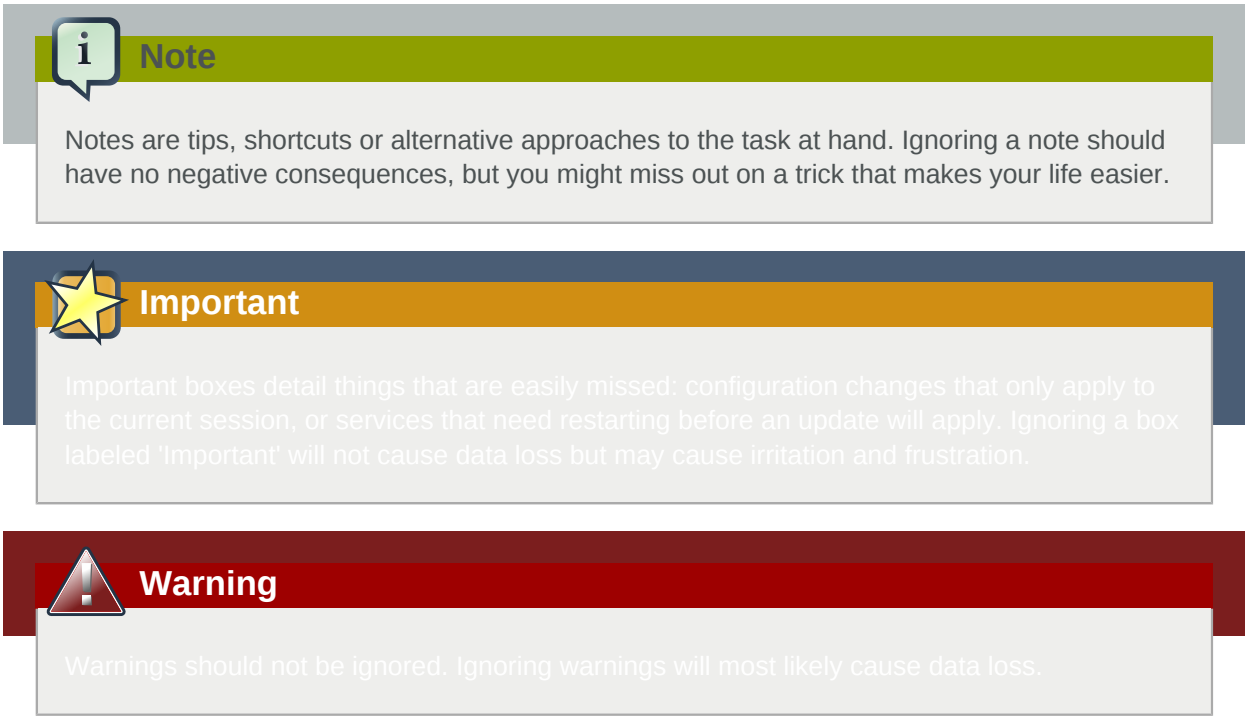
```
public class ExClient
{
   public static void main(String args[])
       throws Exception
   {
      InitialContext iniCtx = new InitialContext();
      Object         ref    = iniCtx.lookup("EchoBean");
      EchoHome       home   = (EchoHome) ref;
      Echo           echo   = home.create();

      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.



2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the JBoss Issue Tracker: *https://jira.jboss.org/* against the product **JBoss Transactions.**

When submitting a bug report, be sure to mention the manual's identifier: *Transactions_Overview_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

About This Guide

The Transaction Fundamentals describes what transactions are, why ACID transactions are good in most cases but extended transactions are necessary in other areas, and other useful information to best use JBossTS.

1.1. Audience

This guide is most relevant for developers who want to understand the details behind transaction systems.

1.2. Prerequisites

None.

Transactions Overview

2.1. What is a transaction?

Note

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

Consider the following situation: a user wishes to purchase access to an on-line newspaper and requires to pay for this access from an account maintained by an on-line bank. Once the newspaper site has received the user's credit from the bank, they will deliver an electronic token to the user granting access to their site. Ideally the user would like the debiting of the account, and delivery of the token to be "all or nothing" (atomic). However, hardware and software failures could prevent either event from occurring, and leave the system in an indeterminate state.

- Atomic transactions (transactions) possess an "all-or-nothing" property, and are a well-known technique for guaranteeing application consistency in the presence of failures. Transactions possess the following ACID properties:
- Atomicity: The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- Consistency: Transactions produce consistent results and preserve application specific invariants.
- Isolation: Intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
- Durability: The effects of a committed transaction are never lost (except by a catastrophic failure).

A transaction can be terminated in two ways: committed or aborted (rolled back). When a transaction is committed, all changes made within it are made durable (forced on to stable storage, e.g., disk). When a transaction is aborted, all of the changes are undone. Atomic actions can also be nested; the effects of a nested action are provisional upon the commit/abort of the outermost (top-level) atomic action.

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

The main components involved in using and defining transactional applications are:

- A Transaction Service: The Transaction Service captures the model of the underlying transaction protocol and coordinates parties affiliated with the transaction according to that model.

- A Transaction API: Provides an interface for transaction demarcation and the registration of participants.
- A Participant: The entity that cooperates with the transaction service on behalf of its associated business logic.
- The Context: Captures the necessary details of the transaction such that participants can enlist within its scope.

2.2. The Coordinator

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

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

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

2.3. The Transaction Context

In order for a transaction to span a number of services, certain information has to be shared between those services in order to propagate information about the transaction. This information is known as the Context. The context is often automatically propagated and processed by transaction-aware components of an application:

Contents of a Context

Transaction Identifier
: Guarantees global uniqueness for an individual transaction.

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

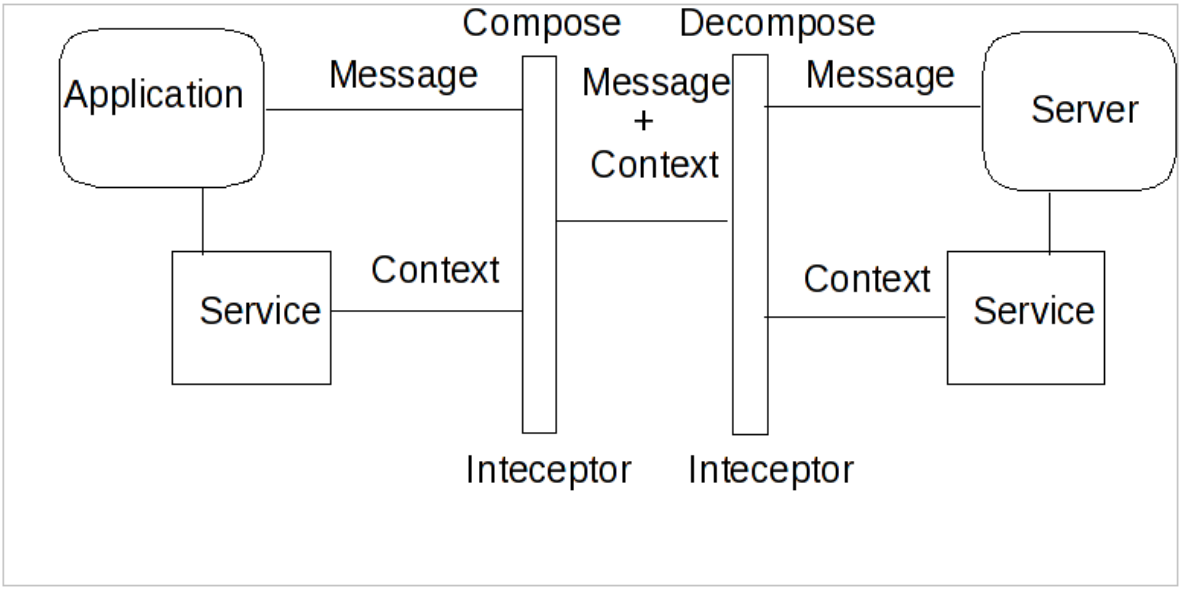


Figure 2.1. Context Flow

2.4. Participants

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

2.5. Commit protocol

A two-phase commit protocol is required to guarantee that all of the action participants either commit or abort any changes made. See *Figure 2.2, "Two-Phase Commit Overview"* which illustrates the main aspects of the commit protocol: during phase 1, the action coordinator, C, attempts to communicate with all of the action participants, A and B, to determine whether they will commit or abort. An abort reply from any participant acts as a veto, causing the entire action to abort. Based upon these (lack of) responses, the coordinator arrives at the decision of whether to commit or abort the action. If the action will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the action aborts.

When each participant receives the coordinator's phase 1 message, they record sufficient information on stable storage to either commit or abort changes made during the action. After returning the phase 1 response, each participant who returned a commit response must remain blocked until it has received the coordinator's phase 2 message. Until they receive this message, these resources are unavailable for use by other actions. If the coordinator fails before delivery of this message, these resources remain blocked. However, if crashed machines eventually recover, crash recovery mechanisms can be employed to unblock the protocol and terminate the action.

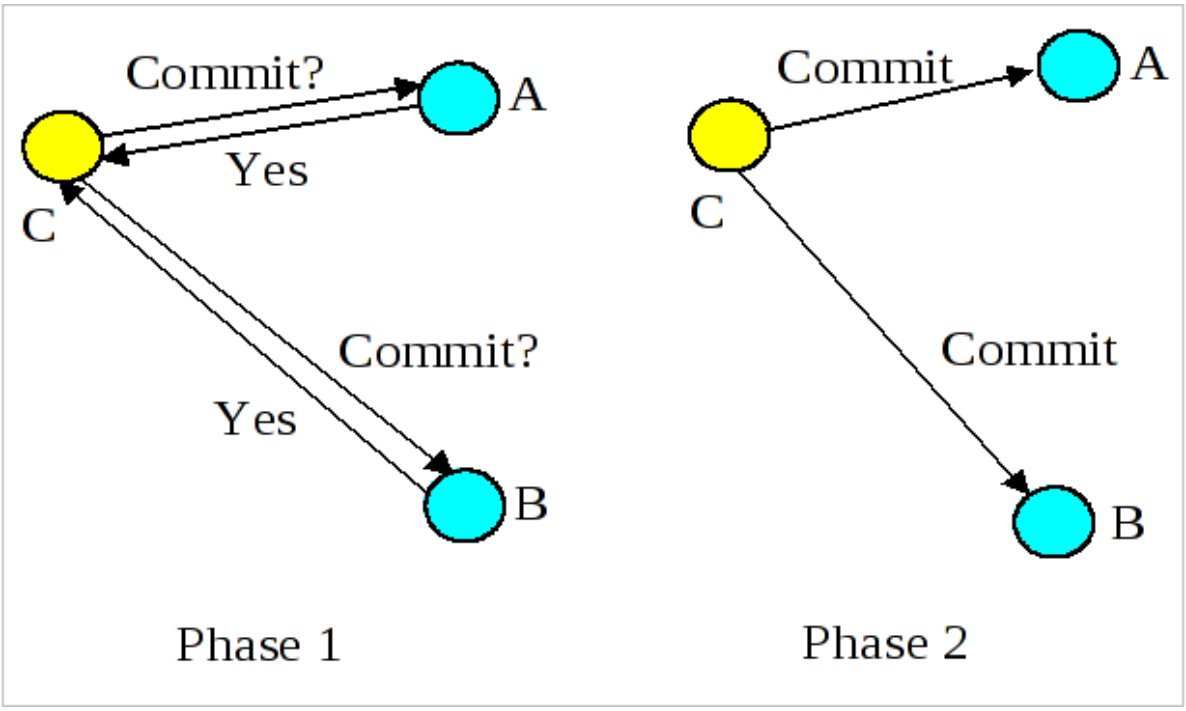


Figure 2.2. Two-Phase Commit Overview

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.

2.6. 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 2.1. 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.

2.7. 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 2.1, "Variants to the Two-Phase Commit Protocol"* gives more information about each one.

Table 2.1. 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 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

Variant	Description
	no affect on the transaction. Therefore, a read-only participant can be omitted from the second phase of the commit protocol.

2.8. 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 2.2, "Heuristic Outcomes"*.

Table 2.2. 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.

2.9. Interposition

Interposition is a scoping mechanism which allows coordination of a transaction to be delegated across a hierarchy of coordinators. See *Figure 2.3, "Interpositions"* for a graphical representation of this concept.

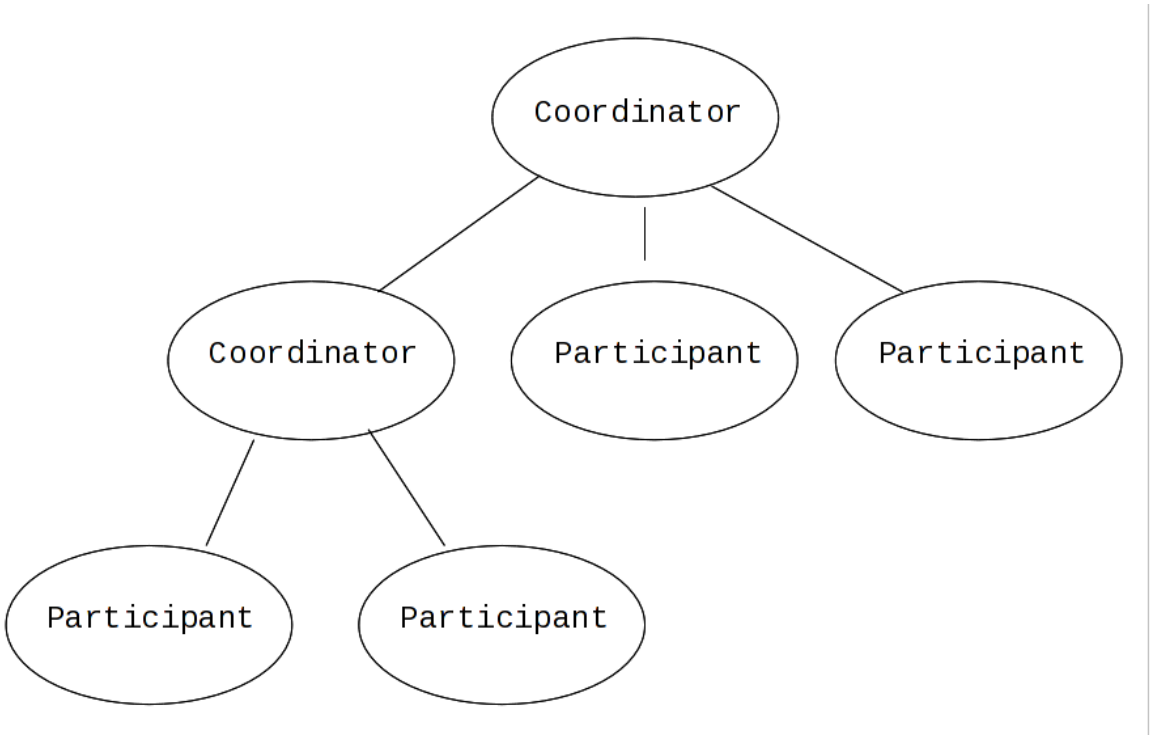


Figure 2.3. Interpositions

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 2.3, "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.

2.10. 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.

2.10.1. Addressing the Problems of Transactioning in Loosely Coupled Systems

Though extended transaction models which relax the ACID properties have been proposed over the years, standards such as OASIS WS-TX provide a new transaction protocol to implement these concepts for the Web services architecture. The are designed to accommodate four underlying requirements inherent in any loosely coupled architecture like 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.

Appendix A. Revision History

Revision 1 **Tue Apr 12 2010** **Tom Jenkinson**
tom.jenkinson@redhat.com

Initial creation of book by publican