JBossJTS Development Guide

Developing distributed transactional applications with JBossJTS

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

edited by Misty Red Hat Stanley-Jones

1. Audience

This guide is specifically intended for service developers using JBoss Transactions. It is also contains useful information about how transactional applications work in general.

2. Prerequisites

To understand this guide, you need a basic familiarity with Java service development and object-oriented programming.

Other helpful knowledge

- A general understanding of the APIs, components, and objects that are present in Java applications.
- A general understanding of Linux, UNIX, or Microsoft Windows server.

3. 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.

3.1. 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.

3.2. 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"));
   }
}
```

3.3. 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.

4. We Need Feedback!

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

Transaction Processing Overview

1.1. Defining a transaction

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.

Practical Example. If you subscribe to a newspaper using a credit card, you are using a transactional system. Multiple systems are involved, and each of the systems needs the ability to roll back its work, and cause the entire transaction to roll back if necessary. For instance, if the newspaper's subscription system goes offline halfway through your transaction, you don't want your credit card to be charged. If the credit card is over its limit, the newspaper doesn't want your subscription to go through. In either of these cases, the entire transaction should fail of any part of it fails. Neither you as the customer, nor the newspaper, nor the credit card processor, wants an unpredictable (indeterminate) outcome to the transaction.

This ability to roll back an operation if any part of it fails is what JBoss Transactions is all about. This guide assists you in writing transactional applications to protect your data.

"Transactions" in this guide refers to atomic transactions, and embody the "all-or-nothing" concept outlined above. Transactions are used to guarantee the consistency of data in the presence of failures. Transactions fulfill the requirements of ACID: Atomicity, Consistency, Isolation, Durability.

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.

1.2. Commit protocol

A two-phase commit protocol guarantees that all of the transaction participants either commit or abort any changes made. *Figure 1.1, "Two-Phase Commit"* illustrates the main aspects of the commit protocol.

Procedure 1.1. Two-phase commit protocol

1. 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.
2. An abort reply from any participant acts as a veto, causing the entire action to abort.
3. Based upon these (lack of) responses, the coordinator chooses to commit or abort the action.
4. 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.
5. When each participant receives the coordinator's phase-one message, it records sufficient information on stable storage to either commit or abort changes made during the action.
6. After returning the phase-one response, each participant who returned a commit response must remain blocked until it has received the coordinator's phase-two message.
7. 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.

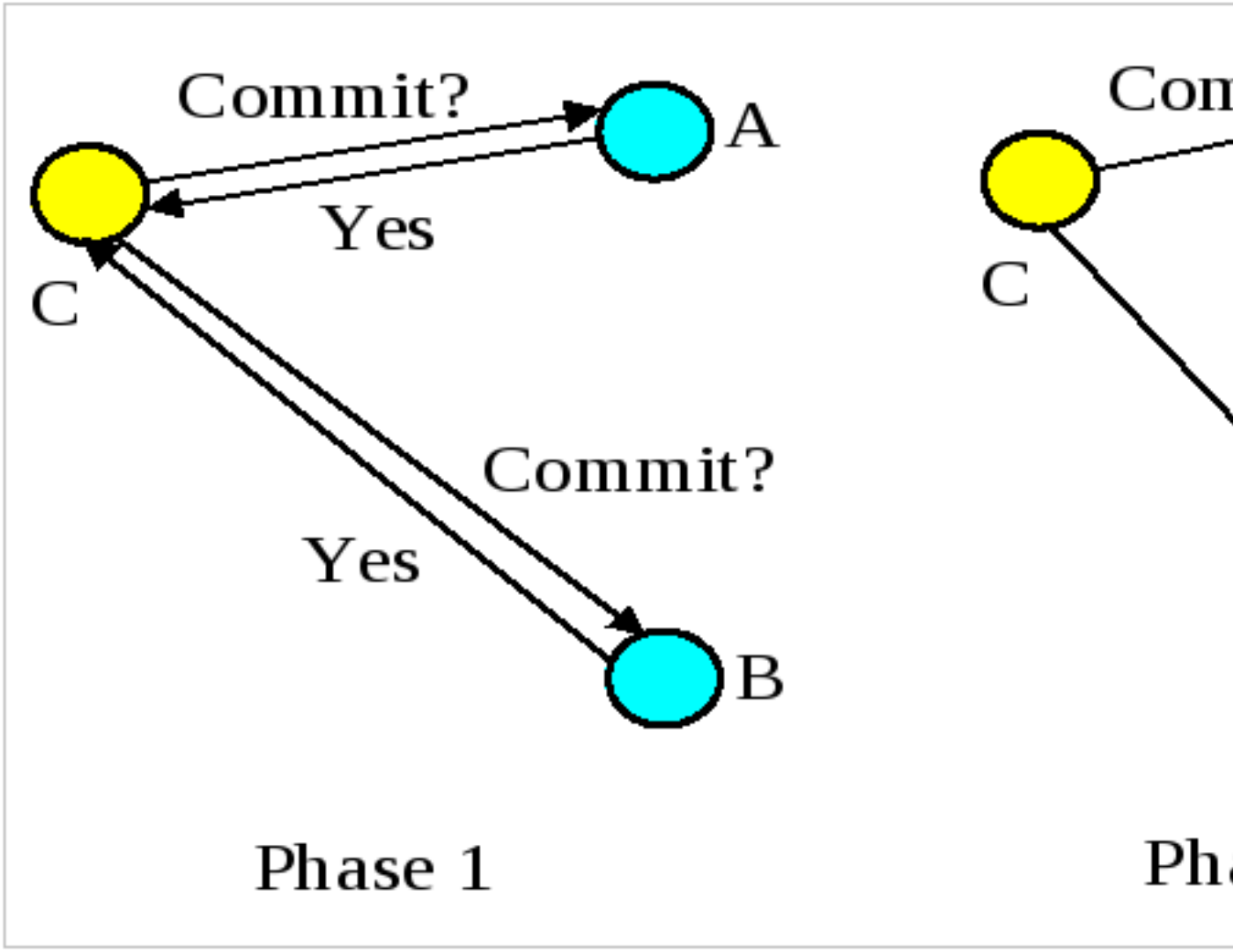

Figure 1.1. Two-Phase Commit

1.3. Transactional proxies

The action coordinator maintains a transaction context where resources taking part in the action need to be registered. Resources must obey the transaction commit protocol to guarantee ACID properties. Typically, the resource provides specific operations which the action can invoke during the commit/abort protocol. However, some resources may not be able to be transactional in this way. This may happen if you have legacy code which cannot be modified. Transactional proxies allow you to use these anomalous resources within an action.

The proxy is registered with, and manipulated by, the action as though it were a transactional resource, and the proxy performs implementation specific work to make the resource it represents transactional. The proxy must participate within the commit and abort protocols. Because the work of the proxy is performed as part of the action, it is guaranteed to be completed or undone despite failures of the action coordinator or action participants.

1.4. Nested transactions

Given a system that provides transactions for certain operations, you can combine them to form another operation, which is also required to be a transaction. The resulting transaction's effects are a combination of the effects of its constituent transactions. This paradigm creates the concept of nested subtransactions, and the resulting combined transaction is called the enclosing transaction. The enclosing transaction is sometimes referred to as the parent of a nested (or child) transaction. It can also be viewed as a hierarchical relationship, with a top-level transaction consisting of several subordinate transactions.

An important difference exists between nested and top-level transactions.

The effect of a nested transaction is provisional upon the commit/roll back of its enclosing transactions. The effects are recovered if the enclosing transaction aborts, even if the nested transaction has committed.

Subtransactions are a useful mechanism for two reasons:

fault-isolation
: If a subtransaction rolls back, perhaps because an object it is using fails, the enclosing transaction does not need to roll back.

modularity
: If a transaction is already associated with a call when a new transaction begins, the new transaction is nested within it. Therefore, if you know that an object requires transactions, you can them within the object. If the object's methods are invoked without a client transaction, then the object's transactions are top-level. Otherwise, they are nested within the scope of the client's transactions. Likewise, a client does not need to know whether an object is transactional. It can begin its own transaction.

1.5. The Object Transaction Service (OTS)

The CORBA architecture, as defined by the OMG, is a standard which promotes the construction of interoperable applications that are based upon the concepts of distributed objects. The architecture principally contains the following components:

Object Request Broker (ORB)
: Enables objects to transparently send and receive requests in a distributed, heterogeneous environment. This component is the core of the OMG reference model.

Object Services
: A collection of services that support functions for using and implementing objects. Such services are necessary for the construction of any distributed application. The Object Transaction Service (OTS) is the most relevant to JBossJTS.

Common Facilities

Other useful services that applications may need, but which are not considered to be fundamental. Desktop management and help facilities fit this category.

The CORBA architecture allows both implementation and integration of a wide variety of object systems. In particular, applications are independent of the location of an object and the language in which an object is implemented, unless the interface the object explicitly supports reveals such details. As defined in the OMG CORBA Services documentation, *object services* are defined as a collection of services (interfaces and objects) that support the basic functions for using and implementing objects. These services are necessary to construct distributed application, and are always independent of an application domain. The standards specify several core services including naming, event management, persistence, concurrency control and transactions.

Note

The OTS specification allows, but does not require, nested transactions. JBossTS is a fully compliant version of the OTS version 1.1 draft 5, and support nested transactions.

The transaction service provides interfaces that allow multiple distributed objects to cooperate in a transaction, committing or rolling back their changes as a group. However, the OTS does not require all objects to have transactional behavior. An object's support of transactions can be none at all, for some operations, or fully. Transaction information may be propagated between client and server explicitly, or implicitly. You have fine-grained control over an object's support of transactions. If your objects supports partial or complete transactional behavior, it needs interfaces derived from interface `TransactionalObject`.

The Transaction Service specification also distinguishes between recoverable objects and transactional objects. Recoverable objects are those that contain the actual state that may be changed by a transaction and must therefore be informed when the transaction commits or aborts to ensure the consistency of the state changes. This is achieved be registering appropriate objects that support the Resource interface (or the derived SubtransactionAwareResource interface) with the current transaction. Recoverable objects are also by definition transactional objects.

In contrast, a simple transactional object does not necessarily need to be recoverable if its state is actually implemented using other recoverable objects. A simple transactional object does not need to participate the commit protocol used to determine the outcome of the transaction since it maintains no state information of its own.

The OTS is a protocol engine that guarantees obedience to transactional behavior. It does not directly support all of the transaction properties, but relies on some cooperating services:

Persistence/Recovery Service	Supports properties of atomicity and durability.
Concurrency Control Service	Supports the isolation properties.

You are responsible for using the appropriate services to ensure that transactional objects have the necessary ACID properties.

JBossTS Basics

2.1. Introduction

JBossTS is based upon the original Arjuna system developed at the University of Newcastle between 1986 and 1995. Arjuna predates the OTS specification and includes many features not found in the OTS. JBossTS is a superset of the OTS. Applications written using the standard OTS interfaces are portable across OTS implementations.

JBossTS features in terms of OTS specifications

- full draft 5 compliance, with support for Synchronization objects and PropagationContexts.
- support for subtransactions.
- implicit context propagation where support from the ORB is available.
- support for multi-threaded applications.
- fully distributed transaction managers, i.e., there is no central transaction manager, and the creator of a top-level transaction is responsible for its termination. Separate transaction manager support is also available, however.
- transaction interposition.
- X/Open compliance, including checked transactions. This checking can optionally be disabled. Note: checked transactions are disabled by default, i.e., any thread can terminate a transaction.
- JDBC support.
- Full JTA 1.1 support.

You can use JBossTS in three different levels, which correspond to the sections in this chapter, and are each explored in their own chapters as well.

Because of differences in ORB implementations, JBossTS uses a separate ORB Portability library which acts as an abstraction later. Many of the examples used throughout this manual use this library. Refer to the ORB Portability Manual for more details.

2.1.1. Raw OTS

The OTS is only a protocol engine for driving registered resources through a two-phase commit protocol. You are responsible for building and registering the `Resource` objects which handle persistence and concurrency control, ensuring ACID properties for transactional application objects. You need to register `Resources` at appropriate times, and ensure that a given `Resource` is

only registered within a single transaction. Programming at the raw OTS level is extremely basic. You as the programmer are responsible for almost everything to do with transactions, including managing persistence and concurrency control on behalf of every transactional object.

2.1.2. Enhanced OTS functionality

The OTS implementation of nested transactions is extremely limited, and can lead to the generation of heuristic results. An example of such a result is when a subtransaction coordinator discovers part of the way through committing that some resources cannot commit, but being unable to tell the committed resources to abort. JBossTS allows nested transactions to execute a full two-phase commit protocol, which removes the possibility that some resources will comment while others roll back.

When resources are registered with a transaction, you have no control over the order in which these resources are invoked during the commit/abort protocol. For example, if previously registered resources are replaced with newly registered resources, resources registered with a subtransaction are merged with the subtraction's parent. JBossTS provides an additional Resource subtype which you this level of control.

2.1.3. Advanced API

The OTS does not provide any `Resource` implementations. You are responsible for implementing these interfaces. The interfaces defined within the OTS specification are too low-level for most application programmers. Therefore, JBossTS includes **Transactional Objects for Java (TXOJ)**, which makes use of the raw Common Object Services interfaces but provides a higher-level API for building transactional applications and frameworks. This API automates much of the activities concerned with participating in an OTS transaction, freeing you to concentrate on application development, rather than transactions.

The architecture of the system is shown in Figure 2. The API interacts with the concurrency control and persistence services, and automatically registers appropriate resources for transactional objects. These resources may also use the persistence and concurrency services.

JBossTS exploits object-oriented techniques to provide you with a toolkit of Java classes which are inheritable by application classes, to obtain transactional properties. These classes form a hierarchy, illustrated in *Figure 2.1, “JBossTS class hierarchy”*.

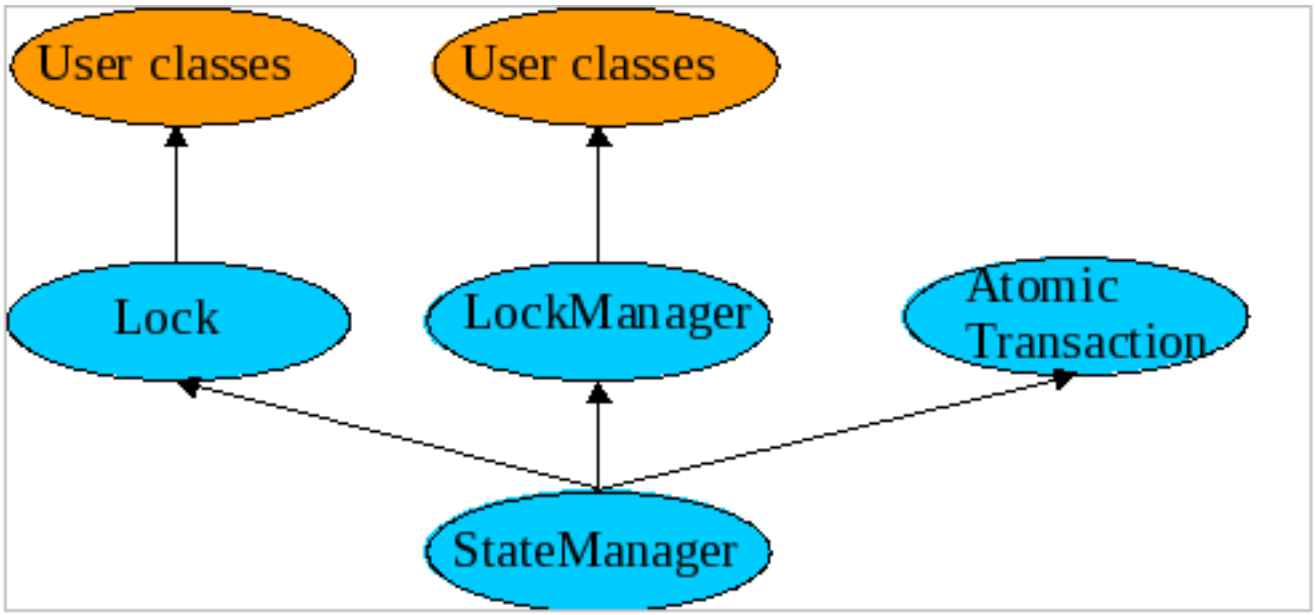

Figure 2.1. JBossTS class hierarchy

Your main responsibilities are specifying the scope of transactions and setting appropriate locks within objects. JBossTS guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions. Crash recovery mechanisms are invoked automatically in the event of failures. When using the provided interfaces, you do not need to create or register `Resource` objects or call services controlling persistence or recovery. If a transaction is nested, resources are automatically propagated to the transaction's parent upon commit.

The design and implementation goal of JBossTS was to provide a programming system for constructing fault-tolerant distributed applications. Three system properties were considered highly important:

Integration of Mechanisms	Fault-tolerant distributed systems require a variety of system functions for naming, locating and invoking operations upon objects, as well as for concurrency control, error detection and recovery from failures. These mechanisms are integrated in a way that is easy for you to use.
Flexibility	Mechanisms must be flexible, permitting implementation of application-specific enhancements, such as type-specific concurrency and recovery control, using system defaults.
Portability	You need to be able to run JBossTS on any ORB.

JBossTS is implemented in Java and extensively uses the type-inheritance facilities provided by the language to provide user-defined objects with characteristics such as persistence and recoverability.

2.2. JBossTS and the OTS implementation

The OTS specification is written with flexibility in mind, to cope with different application requirements for transactions. JBossTS supports all optional parts of the OTS specification. In addition, if the specification allows functionality to be implemented in a variety of different ways, JBossTS supports all possible implementations.

Table 2.1. JBossTS implementation of OTS specifications

OTS specification	JBossTS default implementation
If the transaction service chooses to restrict the availability of the transaction context, then it should raise the `Unavailable` exception.	JBossTS does not restrict the availability of the transaction context.
An implementation of the transaction service need not initialize the transaction context for every request.	JBossTS only initializes the transaction context if the interface supported by the target object extends the `TransactionalObject` interface.
An implementation of the transaction service may restrict the ability for the `Coordinator`, `Terminator`, and `Control` objects to be transmitted or used in other execution environments to enable it to guarantee transaction integrity.	JBossTS does not impose restrictions on the propagation of these objects.
The transaction service may restrict the termination of a transaction to the client that started it.	JBossTS allows the termination of a transaction by any client that uses the `Terminator` interface. In addition, JBossTS does not impose restrictions when clients use the `Current` interface.
A `TransactionFactory` is located using the `FactoryFinder` interface of the life-cycle service.	JBossTS provides multiple ways in which the `TransactionFactory` can be located.
A transaction service implementation may use the Event Service to report heuristic decisions.	JBossTS does not use the Event Service to report heuristic decisions.
An implementation of the transaction service does not need to support nested transactions.	JBossTS supports nested transactions.
`Synchronization` objects must be called whenever the transaction commits.	JBossTS allows `Synchronizations` to be called no matter what state the transaction terminates with.
A transaction service implementation is not required to support interposition.	JBossTS supports various types of interposition.

2.3. Thread class

JBossTS is fully multi-threaded and supports the OTS notion of allowing multiple threads to be active within a transaction, and for a thread to execute multiple transactions. A thread can only be active within a single transaction at a time, however. By default, if a thread is created within the scope of a transaction, the new thread is not associated with the transaction. If the thread needs to be associated with the transaction, use the `resume` method of either the `AtomicTransaction` class or the `Current` class.

However, if newly created threads need to automatically inherit the transaction context of their parent, then they should extend the `OTS_Thread` class.

Example 2.1. Extending the `OTS_Thread` class

```
public class OTS_Thread extends Thread
{
    public void terminate ();
    public void run ();

    protected OTS_Thread ();
};
```

Call the `run` method of `OTS_Thread` at the start of the application thread class's `run` method. Call `terminate` before you exit the body of the application thread's `run` method.

2.4. ORB portability issues

Although the CORBA specification is a standard, it is written so that an ORB can be implemented in multiple ways. As such, writing portable client and server code can be difficult. Because JBossTS has been ported to most of the widely available ORBs, it includes a series of ORB Portability classes and macros. If you write your application using these classes, it should be mostly portable between different ORBs. These classes are described in the separate ORB Portability Manual.

Introduction to the OTS

Basic JBossTS programming involves using the OTS interfaces provided in the `CosTransactions` module, which is specified in `CosTransactions.idl`. This chapter is based on the `OTS Specification1`, specifically with the aspects of OTS that are valuable for developing OTS applications using JBossTS. Where relevant, each section describes JBossTS implementation decisions and runtime choices available to you. These choices are also summarized at the end of this chapter. Subsequent chapters illustrate using these interfaces to construct transactional applications.

3.1. Defining the OTS

The raw `CosTransactions` interfaces reside in package org.omg.CosTransactions. The JBossTS implementations of these interfaces reside in package com.arjuna.CosTransactions and its sub-packages.

You can override many run-time decisions of JBossTS Java properties specified at run-time. The property names are mentioned in the `com.arjuna.ats.jts.common.Environment` class.

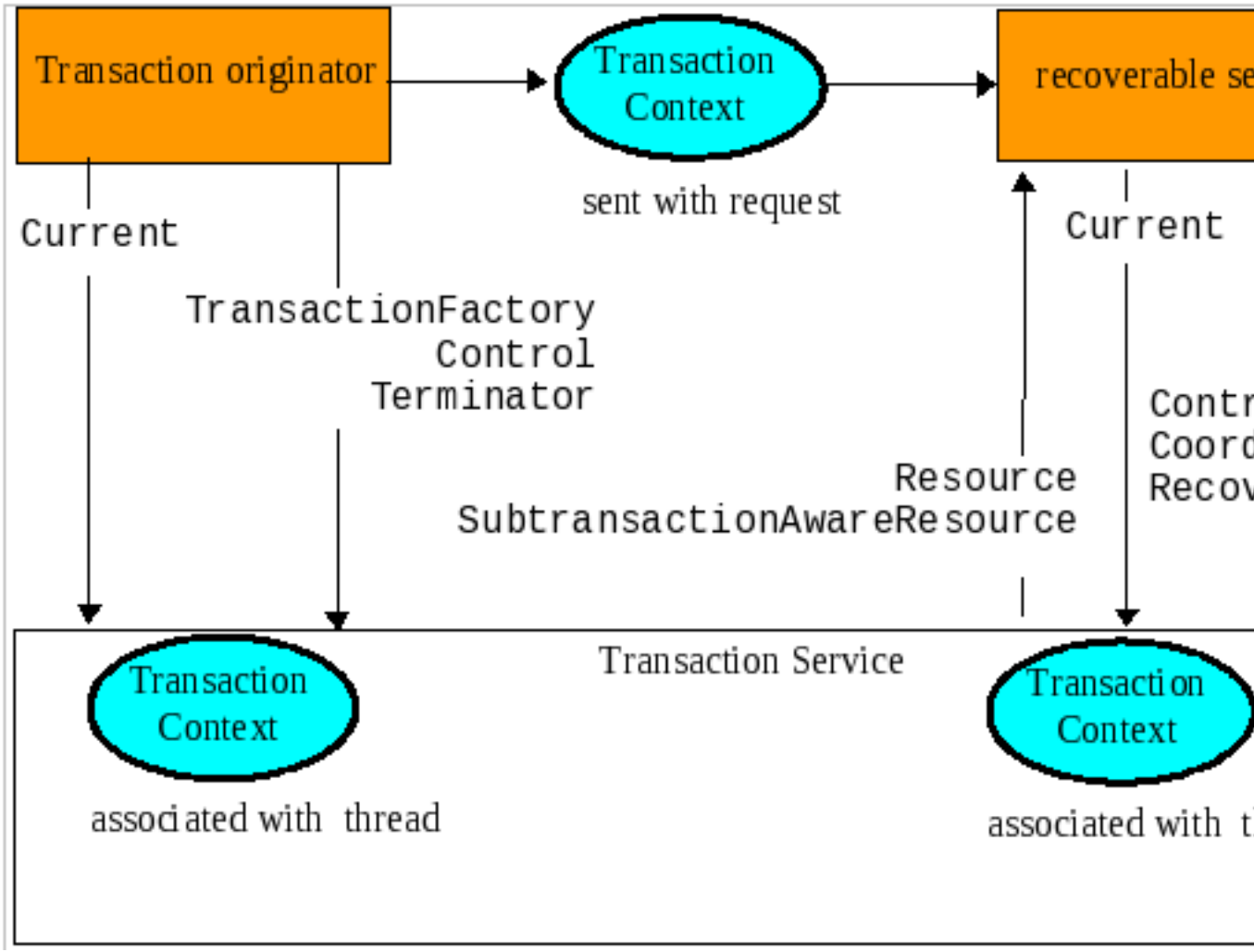

Figure 3.1. OTS architecture

3.2. Action programming models

A client application program can manage a transaction using direct or indirect context management.

- *Indirect context management* means that an application uses the pseudo-object `Current`, provided by the Transaction Service, to associate the transaction context with the application thread of control.
- For *direct context management*, an application manipulates the `Control` object and the other objects associated with the transaction.

An object may require transactions to be either explicitly or implicitly propagated to its operations.

- *Explicit propagation* means that an application propagates a transaction context by passing objects defined by the Transaction Service as explicit parameters. Typically the object is the `PropagationContext` structure.

- *Implicit propagation* means that requests are implicitly associated with the client's transaction, by sharing the client's transaction context. The context is transmitted to the objects without direct client intervention. Implicit propagation depends on indirect context management, since it propagates the transaction context associated with the `Current` pseudo-object. An object that supports implicit propagation should not receive any Transaction Service object as an explicit parameter.

A client may use one or both forms of context management, and may communicate with objects that use either method of transaction propagation. This results in four ways in which client applications may communicate with transactional objects:

Direct Context Management/Explicit Propagation
: The client application directly accesses the `Control` object, and the other objects which describe the state of the transaction. To propagate the transaction to an object, the client must include the appropriate Transaction Service object as an explicit parameter of an operation. Typically, the object is the `PropagationContext` structure.

Indirect Context Management/Implicit Propagation
: The client application uses operations on the `Current` pseudo-object to create and control its transactions. When it issues requests on transactional objects, the transaction context associated with the current thread is implicitly propagated to the object.

Indirect Context Management/Explicit Propagation
: for an implicit model application to use explicit propagation, it can get access to the Control using the get_control operation on the Current pseudo object. It can then use a Transaction Service object as an explicit parameter to a transactional object; for efficiency reasons this should be the PropagationContext structure, obtained by calling get_txcontext on the appropriate Coordinator reference. This is explicit propagation.

Direct Context Management/Implicit Propagation
: A client that accesses the Transaction Service objects directly can use the `resume` pseudo-object operation to set the implicit transaction context associated with its thread. This way, the client can invoke operations of an object that requires implicit propagation of the transaction context.

The main difference between direct and indirect context management is the effect on the invoking thread's transaction context. Indirect context management causes the thread's transaction context to be modified automatically by the OTS. For instance, if method `begin` is called, the thread's notion of the current transaction is modified to the newly-created transaction. When the transaction is terminated, the transaction previously associated with the thread, if one existed, is restored as the thread's context. This assumes that subtransactions are supported by the OTS implementation.

If you use direct management, no changes to the thread's transaction context are made by the OTS, leaving the responsibility to you.

3.3. Interfaces

Table 3.1. Interfaces

Function	Used by	Direct context mgmt	Indirect context mgmt

Create a transaction	Transaction originator	`Factory::create`
		`Control::get_terminator`
		`Control::get_coordinator`
			begin
			set_timeout

Terminate a transaction	Transaction originator (implicit) All (explicit)	`Terminator::commit` `Terminator::rollback`	commit rollback
Rollback transaction	Server	`Terminator::rollback_only`	`rollback_only`
Propagation of transaction to server	Server	Declaration of method parameter	`TransactionalObject`

Client control of transaction propagation to server	All	Request parameters	`get_control`
			`suspend`
			`resume`

Register with a transaction	Recoverable Server	`Coordinator::register_resource`	N/A

Miscellaneous	All	`Coordinator::get_status`
		`Coordinator::get_transaction_name`
		`Coordinator::is_same_transaction`
		`Coordinator::hash_transaction`
		`get_status`
		`get_transaction_name`
			N/A

Note

For clarity, subtransaction operations are not shown

3.4. Transaction factory

The `TransactionFactory` interface allows the transaction originator to begin a top-level transaction. Subtransactions must be created using the `begin` method of `Current`, or the `create_subtransaction` method of the parent's Coordinator.) Operations on the factory and `Coordinator` to create new transactions use direct context management, and therefore do not modify the calling thread's transaction context.

The `create` operation creates a new top-level transaction and returns its `Control` object, which you can use to manage or control participation in the new transaction. Method `create` takes a parameter that is is an application-specific timeout value, in seconds. If the transaction does not complete before this timeout elapses, it is rolled back. If the parameter is `0`, no application-specific timeout is established.

Note

Subtransactions do not have a timeout associated with them.

The Transaction Service implementation allows the `TransactionFactory` to be a separate server from the application, shared by transactions clients, and which manages transactions on their behalf. However, the specification also allows the TransactionFactory to be implemented by an object within each transactional client. This is the default implementation used by JBossTS, because it removes the need for a separate service to be available in order for transactional applications to execute, and therefore reduces a point of failure.

If your applications require a separate transaction manager, set the `OTS_TRANSACTION_MANAGER` environment variable to the value `YES`. The system locates the transaction manager server in a manner specific to the ORB being used. The server can be located in a number of ways.

- Registration with a name server.
- Addition to the ORB's initial references, using a JBossTS specific references file.
- The ORB's specific location mechanism, if applicable.

3.4.1. OTS configuration file

Similar to the `resolve_initial_references`, JBossTS supports an initial reference file where you can store references for specific services, and use these references at runtime. The file, `CosServices.cfg`, consists of two columns, separated by a single space.

- The service name, which is `TransactionService` in the case of the OTS server.
- The IOR

`CosServices.cfg` is usually located in the `etc/` directory of the JBossTS installation. The OTS server automatically registers itself in this file, creating it if necessary, if you use the configuration file mechanism. Stale information is also automatically removed. The Transaction Service locates `CosServices.cfg` at runtime, using the `OrbPortabilityEnvironmentBean` properties `initialReferencesRoot` and `InitialReferencesFile`. `initialReferencesRoot` names a directory, and defaults to the current working directory. `initialReferencesFile` refers to a file within the `initialReferencesRoot`, and defaults to the name `CosServices.cfg`.

3.4.2. Name service

If your ORB supports a name service, and you configure JBossTS to use it, the transaction manager is automatically registered with it.

3.4.3. resolve_initial_references

JBossTS does not support `resolve_initial_references`.

3.4.4. Overriding the default location mechanisms

You can override the default location mechanism with the `RESOLVE_SERVICE` property variable, which can have any of three possible values.

CONFIGURATION_FILE	This is the default option, and directs the system to use the `CosServices.cfg` file.
NAME_SERVICE	JBossTS tries to use a name service to locate the transaction factory. If the ORB does not support the name service mechanism, JBossTS throws an exception.
BIND_CONNECT	JBossTS uses the ORB-specific bind mechanism. If the ORB does not support such a mechanism, JBossTS throws an exception.

If `RESOLVE_SERVICE` is specified when running the transaction factory, the factory registers itself with the specified resolution mechanism.

3.5. Transaction timeouts

As of JBossTS 4.5, transaction timeouts are unified across all transaction components and are controlled by **ArjunaCore**. Refer to the *ArjunaCore Development Guide* for more information.

3.6. Transaction contexts

Transaction contexts are fundamental to the OTS architecture. Each thread is associated with a context in one of three ways.

Null	The thread has no associated transaction.

A transaction ID	The thread is associated with a transaction.

Contexts may be shared across multiple threads. In the presence of nested transactions, a context remembers the stack of transactions started within the environment, so that the context of the thread can be restored to the state before the nested transaction started, when the nested transaction ends. Threads most commonly use object `Current` to manipulate transactional information, which is represented by `Control` objects. `Current` is the broker between a transaction and `Control` objects.

Your application can manage transaction contexts either directly or indirectly. In the direct approach, the transaction originator issues a request to a `TransactionFactory` to begin a new top-level transaction. The factory returns a `Control` object that enables both a `Terminator` interface and a `Coordinator` interface. `Terminator` ends a transaction. `Coordinator` associates a thread with a transaction, or begins a nested transaction. You need to pass each interface as an explicit parameter in invocations of operations, because creating a transaction with them does not change a thread's current context. If you use the factory, and need to set the current context for a thread to the context which its control object returns, use the `resume` method of interface `Current`.

Example 3.1. Interfaces `Terminator`, `Coordinator`, and `Control`

```
interface Terminator
{
    void commit (in boolean report_heuristics) raises (HeuristicMixed, HeuristicHazard);
    void rollback ();
};

interface Coordinator
{
    Status get_status ();
    Status get_parent_status ();
    Status get_top_level_status ();

    RecoveryCoordinator register_resource (in Resource r) raises (Inactive);
    Control create_subtransaction () raises (SubtransactionsUnavailable,
                                             Inactive);

    void rollback_only () raises (Inactive);

    ...
};

interface Control
{
    Terminator get_terminator () raises (Unavailable);
    Coordinator get_coordinator () raises (Unavailable);
};
```

```
interface TransactionFactory
{
    Control create (in unsigned long time_out);
};
```

When the factory creates a transaction, you can specify a timeout value in seconds. If the transaction times out, it is subject to possible roll-back. Set the timeout to `0` to disable application-specific timeout.

The `Current` interface handles implicit context management. Implicit context management provides simplified transaction management functionality, and automatically creates nested transactions as required. Transactions created using `Current` do not alter a thread's current transaction context.

Example 3.2. Interface `Current`

```
interface Current : CORBA::Current
{
    void begin () raises (SubtransactionsUnavailable);
    void commit (in boolean report_heuristics) raises (NoTransaction,
                                                       HeuristicMixed,
                                                       HeuristicHazard);
    void rollback () raises (NoTransaction);
    void rollback_only () raises (NoTransaction);

    . . .

    Control get_control ();
    Control suspend ();
    void resume (in Control which) raises (InvalidControl);
};
```

3.6.1. Nested transactions

Subtransactions are a useful mechanism for two reasons:

fault-tolerance
: If a subtransaction rolls back, the enclosing transaction does not also need to roll back. This preserves as much of the work done so far, as possible.

modularity
: Indirect transaction management does not require special syntax for creating subtransactions. Begin a transaction, and if another transaction is associated with the calling thread, the new transaction is nested within the existing one. If you know that an object requires transactions,

you can use them within the object. If the object's methods are invoked without a client transaction, the object's transaction is top-level. Otherwise, it is nested within the client's transaction. A client does not need to know whether an object is transactional.

The outermost transaction of the hierarchy formed by nested transactions is called the top-level transaction. The inner components are called subtransactions. Unlike top-level transactions, the commits of subtransactions depend upon the commit/rollback of the enclosing transactions. Resources acquired within a subtransaction should be inherited by parent transactions when the top-level transaction completes. If a subtransaction rolls back, it can release its resources and undo any changes to its inherited resources.

In the OTS, subtransactions behave differently from top-level transactions at commit time. Top-level transactions undergo a two-phase commit protocol, but nested transactions do not actually perform a commit protocol themselves. When a program commits a nested transaction, it only informs registered resources of its outcome. If a resource cannot commit, an exception is thrown, and the OTS implementation can ignore the exception or roll back the subtransaction. You cannot roll back a subtransaction if any resources have been informed that the transaction committed.

3.6.2. Transaction propagation

The OTS supports both implicit and explicit propagation of transactional behavior.

- Implicit propagation means that an operation signature specifies no transactional behavior, and each invocation automatically sends transaction context associated with the calling thread.
- Explicit propagation means that applications must define their own mechanism for propagating transactions. This has the following features:
  - A client to control if its transaction is propagated with any operation invocation.
  - A client can invoke operations on both transactional and non-transactional objects within a transaction.

Transaction context management and transaction propagation are different things that may be controlled independently of each other. Mixing of direct and indirect context management with implicit and explicit transaction propagation is supported. Using implicit propagation requires cooperation from the ORB. The client must send current context associated with the thread with any operation invocations, and the server must extract them before calling the targeted operation.

If you need implicit context propagation, ensure that JBossTS is correctly initialized before you create objects. Both client and server must agree to use implicit propagation. To use implicit context propagation, your ORB needs to support filters or interceptors, or the `CosTSPortability` interface.

Implicit context propagation	Property variable `OTS_CONTEXT_PROP_MODE` set to `CONTEXT`.

Interposition	Property variable `OTS_CONTEXT_PROP_MODE` set to `INTERPOSITION`.

Important

Interposition is required to use the JBossTS Advanced API.

3.6.3. Examples

Example 3.3. Simple transactional client using direct context management and explicit transaction propagation

```
{
   ...
   org.omg.CosTransactions.Control c;
   org.omg.CosTransactions.Terminator t;
      org.omg.CosTransactions.PropagationContext pgtx;

   c = transFact.create(0);         // create top-level action

      pgtx = c.get_coordinator().get_txcontext();
   ...
   trans_object.operation(arg, pgtx);     // explicit propagation
   ...
   t = c.get_terminator();          // get terminator
   t.commit(false);              // so it can be used to commit
   ...
}
```

The next example rewrites the same program to use indirect context management and implicit propagation. This example is considerably simpler, because the application only needs to start and either commit or abort actions.

Example 3.4. Indirect context management and implicit propagation

```
{
   ...
   current.begin();           // create new action
   ...
   trans_object2.operation(arg);    // implicit propagation
   ...
   current.commit(false);        // simple commit
   ...
```

```
}
```

The last example illustrates the flexibility of OTS by using both direct and indirect context management in conjunction with explicit and implicit transaction propagation.

Example 3.5. Direct and direct context management with explicitly and implicit propagation

```
{
   ...
   org.omg.CosTransactions.Control c;
   org.omg.CosTransactions.Terminator t;
      org.omg.CosTransactions.PropagationContext pgtx;

   c = transFact.create(0);         // create top-level action
      pgtx = c.get_coordinator().get_txcontext();

   current.resume(c);            // set implicit context
   ...
   trans_object.operation(arg, pgtx);     // explicit propagation
   trans_object2.operation(arg);       // implicit propagation
   ...
   current.rollback();           // oops! rollback
   ...
}
```

3.7. Transaction controls

The `Control` interface allows a program to explicitly manage or propagate a transaction context. An object supporting the `Control` interface is associated with one specific transaction. The `Control` interface supports two operations: `get_terminator` and `get_coordinator`. `get_terminator` returns an instance of the `Terminator` interface. `get_coordinator` returns an instance of the `Coordinator` interface. Both of these methods throw the `Unavailable` exception if the `Control` cannot provide the requested object. The OTS implementation can restrict the ability to use the Terminator and Coordinator in other execution environments or threads. At a minimum, the creator must be able to use them.

Obtain the `Control` object for a transaction when it is created either by using either the `TransactionFactory` or `create_subtransaction` methods defined by the `Coordinator` interface. Obtain a `Control` for the transaction associated with the current thread using the `get_control` or `suspend` methods defined by the `Current` interface.

3.7.1. JBossTS specifics

The transaction creator must be able to use its `Control`, but the OTS implementation decides whether other threads can use `Control`. JBossTS places no restrictions the users of the `Control`.

The OTS specification does not provide a means to indicate to the transaction system that information and objects associated with a given transaction can be purged from the system. In JBossTS, the `Current` interface destroys all information about a transaction when it terminates. For that reason, do not use any `Control` references to the transaction after it commits or rolls back.

However, if the transaction is terminated using the Terminator interface, it is up to the programmer to signal that the transaction information is no longer required: this can be done using the destroyControl method of the OTS class in the com.arjuna.CosTransactions package. Once the program has indicated that the transaction information is no longer required, the same restrictions on using Control references apply as described above. If destroyControl is not called then transaction information will persist until garbage collected by the Java runtime.

In JBossTS, you can propagate `Coordinators` and `Terminators` between execution environments.

3.8. The `Terminator` interface

The `Terminator` interface supports `commit` and `rollback` operations. Typically, the transaction originator uses these operations. Each object supporting the Terminator interface is associated with a single transaction. Direct context management via the Terminator interface does not change the client thread's notion of the current transaction.

The `commit` operation attempts to commit the transaction. To successfully commit, the transaction must not be marked `rollback only`, and all of its must participants agree to commit. Otherwise, the `TRANSACTION_ROLLEDBACK` exception is thrown. If the `report_heuristics` parameter is `true`, the Transaction Service reports inconsistent results using the `HeuristicMixed` and `HeuristicHazard` exceptions.

When a transaction is committed, the coordinator drives any registered `Resources` using their `prepare` or `commit` methods. These Resources are responsible to ensure that any state changes to recoverable objects are made permanent, to guarantee the ACID properties.

When `rollback` is called, the registered `Resources` need to guarantee that all changes to recoverable objects made within the scope of the transaction, and its descendants, is undone. All resources locked by the transaction are made available to other transactions, as appropriate to the degree of isolation the resources enforce.

3.8.1. JBossTS specifics

See *Section 3.7.1, "JBossTS specifics"* for how long `Terminator` references remain valid after a transaction terminates.

When a transaction is committing, it must make certain state changes persistent, so that it can recover if a failure occurs, and continue to commit, or rollback. To guarantee ACID properties, flush these state changes to the persistence store implementation before the transaction proceeds to commit. Otherwise, the application may assume that the transaction has committed, when the state changes may still volatile storage, and may be lost by a subsequent hardware failure. By default, JBossTS makes sure that such state changes are flushed. However, these flushes can impose a significant performance penalty to the application. To prevent transaction state flushes, set the `TRANSACTION_SYNC` variable to `OFF`. Obviously, do this at your own risk.

When a transaction commits, if only a single resource is registered, the transaction manager does not need to perform the two-phase protocol. A single phase commit is possible, and the outcome of the transaction is determined by the resource. In a distributed environment, this optimization represents a significant performance improvement. As such, JBossTS defaults to performing single phase commit in this situation. Override this behavior at runtime by setting the `COMMIT_ONE_PHASE` property variable to `NO`.

3.9. The `Coordinator` interface

The Coordinator interface is returned by the `get_coordinator` method of the `Control` interface. It supports the operations resources need to participate in a transaction. These participants are usually either recoverable objects or agents of recoverable objects, such as subordinate coordinators. Each object supporting the `Coordinator` interface is associated with a single transaction. Direct context management via the Coordinator interface does not change the client thread's notion of the current transaction. You can terminate transaction directly, through the `Terminator` interface. In that case, trying to terminate the transaction a second time using `Current` causes an exception to be thrown for the second termination attempt.

The operations supported by the Coordinator interface of interest to application programmers are:

Table 3.2. Operations supported by the `Coordinator` interface

`get_status`
 	
`get_parent_status`
 	
`get_top_level_status`
	Return the status of the associated transaction. At any given time a transaction can have one of the following status values representing its progress:
	
	StatusActive
	The transaction is currently running, and has not been asked to prepare or marked for rollback.
	
	StatusMarkedRollback
	The transaction is marked for rollback.
	
	StatusPrepared
	The transaction has been prepared, which means that all subordinates have responded `VoteCommit`.

	StatusCommitted The transaction has committed. It is likely that heuristics exist. Otherwise, the transaction would have been destroyed and `StatusNoTransaction` returned. StatusRolledBack The transaction has rolled back. It is likely that heuristics exist. Otherwise. the transaction would have been destroyed and StatusNoTransaction returned. StatusUnknown The Transaction Service cannot determine the current status of the transaction. This is a transient condition, and a subsequent invocation should return a different status. StatusNoTransaction No transaction is currently associated with the target object. This occurs after a transaction completes. StatusPreparing The transaction is in the process of preparing and the final outcome is not known. StatusCommitting The transaction is in the process of committing. StatusRollingBack The transaction is in the process of rolling back.
`is_same_transaction` and others	You can use these operations for transaction comparison. Resources may use these various operations to guarantee that they are registered only once with a specific transaction.

`hash_transaction`
 	
`hash_top_level_tran`
	Returns a hash code for the specified transaction.

`register_resource`	Registers the specified Resource as a participant in the transaction. The `Inactive` exception is raised if the transaction is already prepared. The `TRANSACTION_ROLLEDBACK` exception is raised if the transaction is marked `rollback only`. If the `Resource` is a `SubtransactionAwareResource` and the transaction is a subtransaction, this operation registers the resource with this transaction and indirectly with the top-level transaction when the subtransaction's ancestors commit. Otherwise, the resource is only registered with the current transaction. This operation returns a `RecoveryCoordinator` which this `Resource` can use during recovery. No ordering of registered Resources is implied by this operation. If `A` is

	registered after `B`, the OTS can operate on them in any order when the transaction terminates. Therefore, do not assume such an ordering exists in your implementation.
register_subtran_aware	Registers the specified subtransaction-aware resource with the current transaction, so that it know when the subtransaction commits or rolls back. This method cannot register the resource as a participant in the top-level transaction. The `NotSubtransaction` exception is raised if the current transaction is not a subtransaction. As with `register_resource`, no ordering is implied by this operation.
register_synchronization	Registers the `Synchronization` object with the transaction so that will be invoked before `prepare` and after the transaction completes. Synchronizations can only be associated with top-level transactions, and the `SynchronizationsUnavailable` exception is raised if you try to register a Synchronization with a subtransaction. As with `register_resource`, no ordering is implied by this operation.
rollback_only	Marks the transaction so that the only possible outcome is for it to rollback. The Inactive exception is raised if the transaction has already been prepared/completed.
create_subtransaction	A new subtransaction is created. Its parent is the current transaction. The `Inactive` exception is raised if the current transaction has already been prepared or completed. If you configure the Transaction Service without subtransaction support, the `SubtransactionsUnavailable` exception is raised.

3.9.1. JBossTS specifics

See *Section 3.7.1, "JBossTS specifics"* to control how long `Coordinator` references remain valid after a transaction terminates.

Note

To disable subtransactions, set set the `OTS_SUPPORT_SUBTRANSACTIONS` property variable to `NO`.

3.10. Heuristics

The OTS permits individual resources to make heuristic decisions. *Heuristic* decisions are unilateral decisions made by one or more participants to commit or abort the transaction, without waiting for the consensus decision from the transaction service. Use heuristic decisions with care and only in exceptional circumstances, because they can lead to a loss of integrity in the system.

If a participant makes a heuristic decision, an appropriate exception is raised during commit or abort processing.

Table 3.3. Possible heuristic outcomes

HeuristicRollback	Raised on an attempt to commit, to indicate that the resource already unilaterally rolled back the transaction.
HeuristicCommit	Raised on an attempt to roll back, to indicate that the resource already unilaterally committed the transaction.
HeuristicMixed	Indicates that a heuristic decision has been made. Some updates committed while others rolled back.
HeuristicHazard	Indicates that a heuristic decision may have been made, and the outcome of some of the updates is unknown. For those updates which are known, they either all committed or all rolled back.

HeuristicMixed takes priority over HeuristicHazard. Heuristic decisions are only reported back to the originator if the *`report_heuristics`* argument is set to `true` when you invoke the commit operation.

3.11. Current

The `Current` interface defines operations that allow a client to explicitly manage the association between threads and transactions, using indirect context management. It defines operations that simplify the use of the Transaction Service.

Table 3.4. Methods of `current`

begin	Creates a new transaction and associates it with the current thread. If the client thread is currently associated with a transaction, and the OTS implementation supported nested transactions, the new transaction becomes a subtransaction of that transaction. Otherwise, the new transaction is a top-level transaction. If the OTS implementation does not support nested transactions, the `SubtransactionsUnavailable` exception is thrown. The thread's notion of the current context is modified to be this transaction.
commit	Commits the transaction. If the client thread does not have permission to commit the transaction, the standard exception `NO_PERMISSION` is raised. The effect is the same as performing the `commit` operation on the corresponding `Terminator` object. The client thread's transaction context is returned to its state before the `begin` request was initiated.
rollback	Rolls back the transaction. If the client thread does not have permission to terminate the transaction, the standard exception `NO_PERMISSION` is raised. The effect is the same as performing

	the `rollback` operation on the corresponding `Terminator` object. The client thread's transaction context is returned to its state before the `begin` request was initiated.
rollback_only	Limits the transaction's outcome to rollback only. If the transaction has already been terminated, or is in the process of terminating, an appropriate exception is thrown.
get_status	Returns the status of the current transaction, or exception `StatusNoTransaction` if no transaction is associated with the thread.
set_timeout	Modifies the timeout associated with top-level transactions for subsequent `begin` requests, for this thread only. Subsequent transactions are subject to being rolled back if they do not complete before the specified number of seconds elapses. Default timeout values for transactions without explicitly-set timeouts are implementation-dependent. JBossTS uses a value of `0`, which results in transactions never timing out. There is no interface in the OTS for obtaining the current timeout associated with a thread. However, JBossTS provides additional support for this. See *Section 3.11.1, "JBossTS specifics"*.
get_control	Obtains a `Control` object representing the current transaction. If the client thread is not associated with a transaction, a null object reference is returned. The operation is not dependent on the state of the transaction. It does not raise the `TRANSACTION_ROLLEDBACK` exception.
suspend	Obtains an object representing a transaction's context. If the client thread is not associated with a transaction, a null object reference is returned. You can pass this object to the `resume` operation to re-establish this context in a thread. The operation is not dependent on the state of the transaction. It does not raise the `TRANSACTION_ROLLEDBACK` exception. When this call returns, the current thread has no transaction context associated with it.
resume	Associates the client thread with a transaction. If the parameter is a null object reference, the client thread becomes associated with no transaction. The thread loses association with any previous transactions.

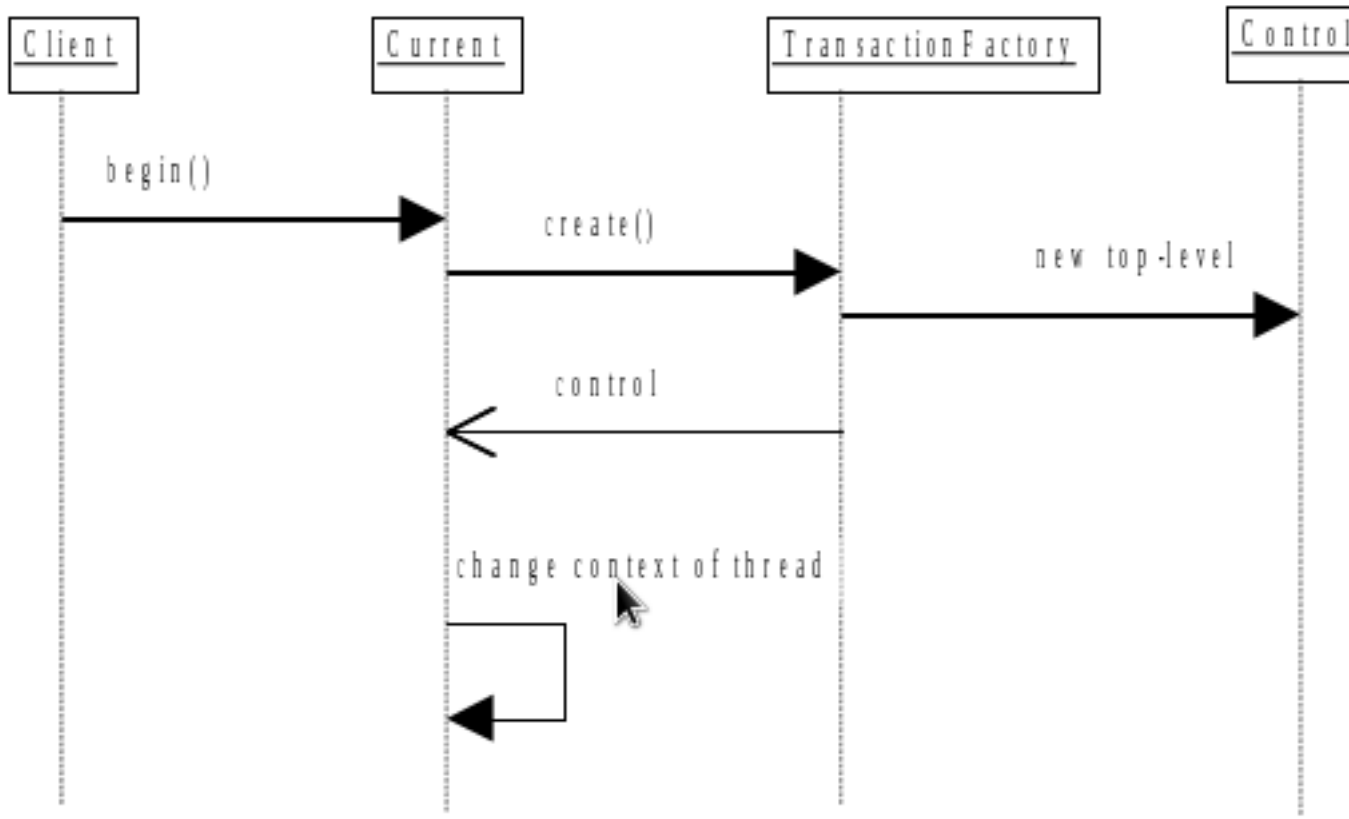

Figure 3.2. Creation of a top-level transaction using `current`

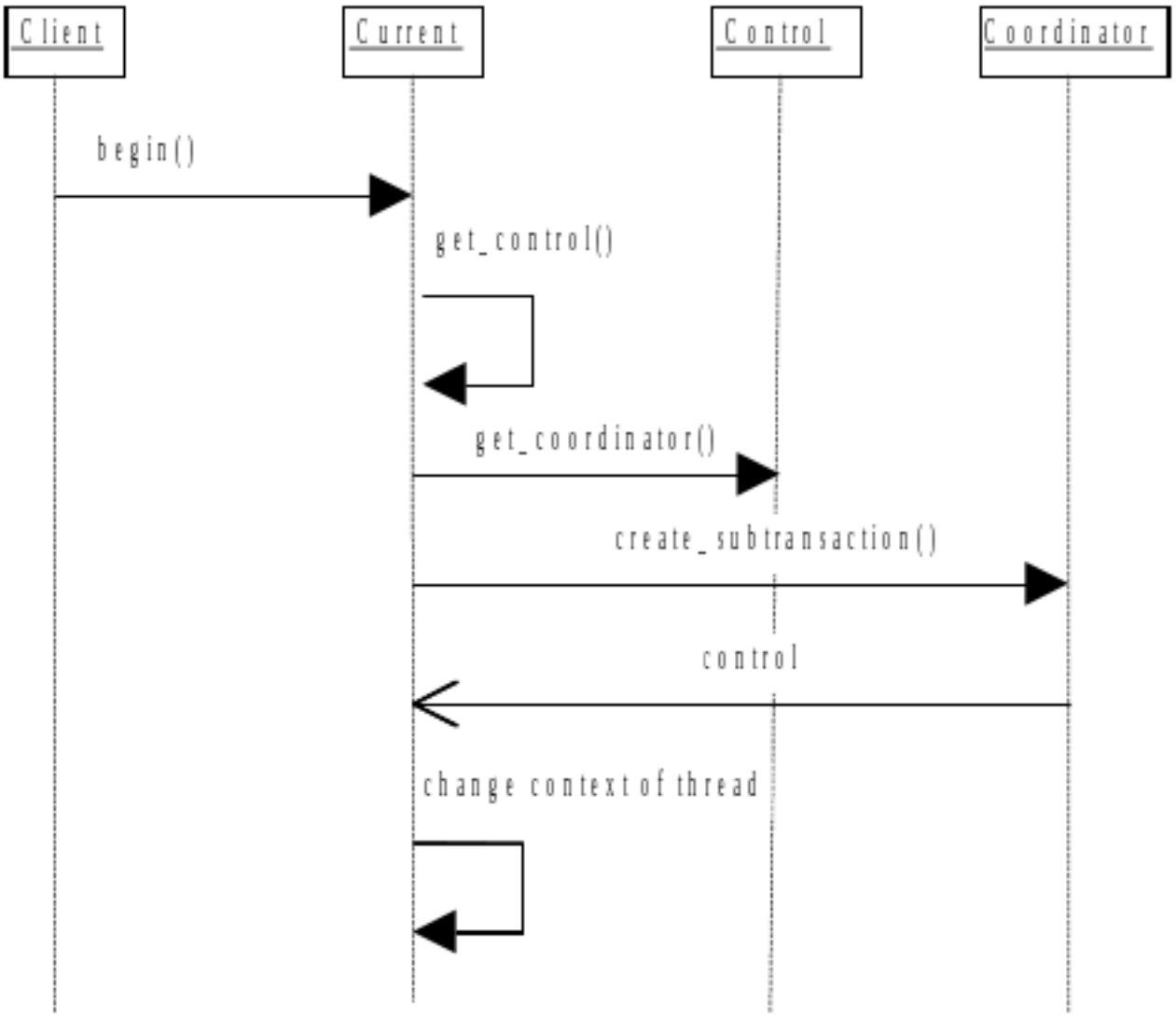

Figure 3.3. Creation of a transaction using `Current`

3.11.1. JBossTS specifics

Ideally, you should Obtain `Current` by using the life-cycle service factory finder. However, very few ORBs support this. JBossTS provides method `get_current` of `Current` for this purpose. This class hides any ORB-specific mechanisms required for obtaining `Current`.

If no timeout value is associated with `Current`, JBossTS associates no timeout with the transaction. The current OTS specification does not provide a means whereby the timeout associated with transaction creation can be obtained. However, JBossTS Current supports a get_timeout method.

By default, the JBossTS implementation of `Current` does not use a separate `TransactionFactory` server when creating new top-level transactions. Each transactional

client has a `TransactionFactory` co-located with it. Override this by setting the `OTS_TRANSACTION_MANAGER` variable to `YES`.

The transaction factory is located in the `bin/` directory of the JBossTS distribution. Start it by executing the OTS script. `Current` locates the factory in a manner specific to the ORB: using the name service, through `resolve_initial_references`, or via the `CosServices.cfg` file. The `CosServices.cfg` file is similar to `resolve_initial_references`, and is automatically updated when the transaction factory is started on a particular machine. Copy the file to each JBossTS instance which needs to share the same transaction factory.

If you do not need subtransaction support, set the `OTS_SUPPORT_SUBTRANSACTIONS` property variable to `NO`. The `setCheckedAction` method overrides the `CheckedAction` implementation associated with each transaction created by the thread.

3.12. Resource

The Transaction Service uses a two-phase commit protocol to complete a top-level transaction with each registered resource.

Example 3.6. Completing a top-level transaction

```
interface Resource
{
    Vote prepare ();
    void rollback () raises (HeuristicCommit, HeuristicMixed,
                             HeuristicHazard);
    void commit () raises (NotPrepared, HeuristicRollback,
                           HeuristicMixed, HeuristicHazard);
    void commit_one_phase () raises (HeuristicRollback, HeuristicMixed,
                                     HeuristicHazard);
    void forget ();
};
```

The `Resource` interface defines the operations invoked by the transaction service. Each `Resource` object is implicitly associated with a single top-level transaction. Do not register a `Resource` with the same transaction more than once. When you tell a `Resource` to prepare, commit, or abort, it must do so on behalf of a specific transaction. However, the `Resource` methods do not specify the transaction identity. It is implicit, since a `Resource` can only be registered with a single transaction.

Transactional objects must use the `register_resource` method to register objects supporting the `Resource` interface with the current transaction. An object supporting the `Coordinator` interface is either passed as a parameter in the case of explicit propagation, or retrieved using operations on the `Current` interface in the case of implicit propagation. If the transaction is nested, the `Resource` is not informed of the subtransaction's completion, and is registered with its parent upon commit.

This example assumes that transactions are only nested two levels deep, for simplicity.

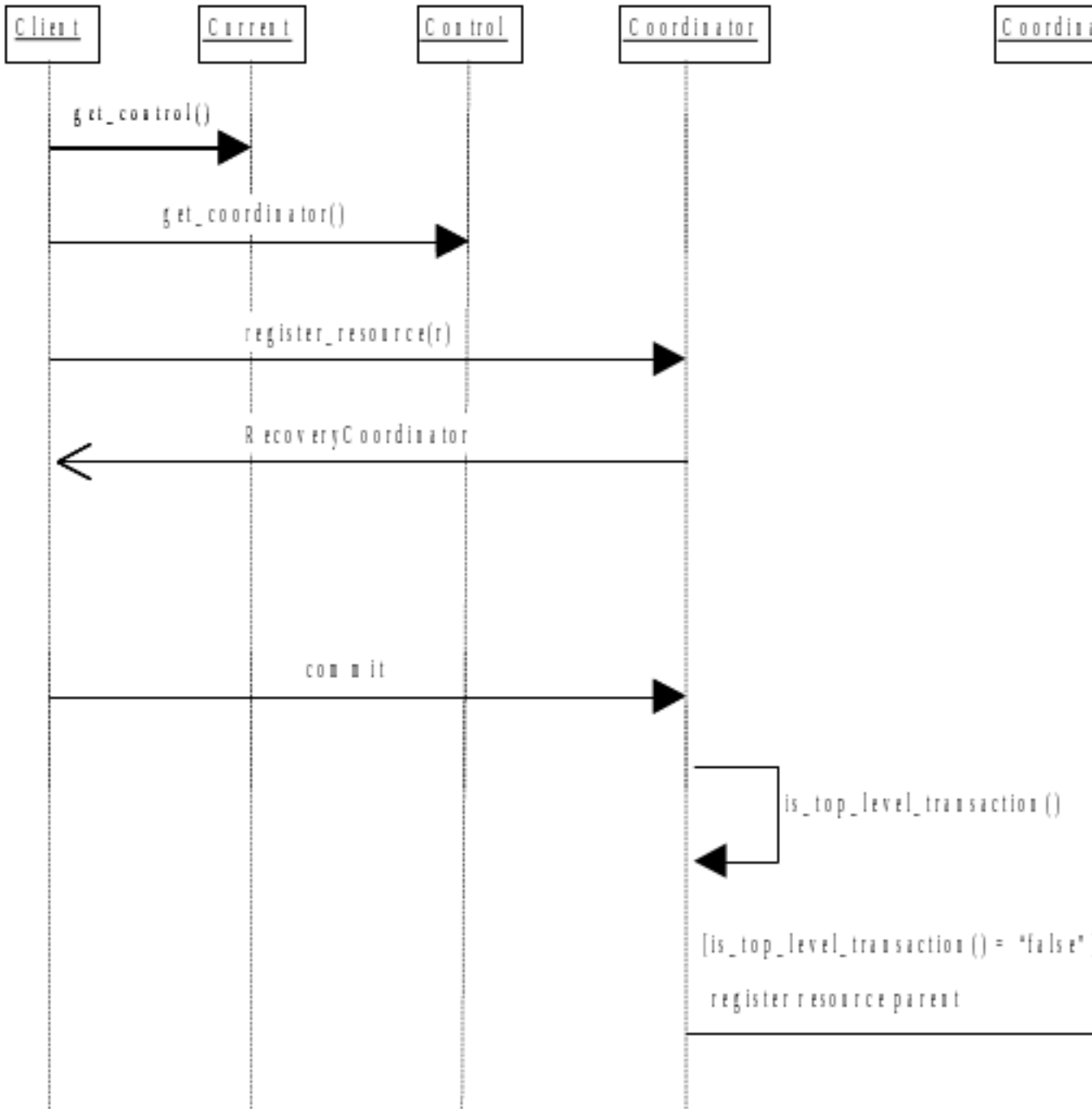

Figure 3.4. `Resource` and nested transactions

Do not register a given `Resource` with the same transaction more than once, or it will receive multiple termination calls. When a `Resource` is directed to prepare, commit, or abort, it needs to link these actions to a specific transaction. Because `Resource` methods do not specify the transaction identity, but can only be associated with a single transaction, you can infer the identity.

A single `Resource` or group of `Resources` guarantees the ACID properties for the recoverable object they represent. A Resource's work depends on the phase of its transaction.

prepare
: If none of the persistent data associated with the resource is modified by the transaction, the Resource can return `VoteReadOnly` and forget about the transaction. It does not need to know the outcome of the second phase of the commit protocol, since it hasn't made any changes.

  If the resource can write, or has already written, all the data needed to commit the transaction to stable storage, as well as an indication that it has prepared the transaction, it can return `VoteCommit`. After receiving this response, the Transaction Service either commits or rolls back. To support recovery, the resource should store the `RecoveryCoordinator` reference in stable storage.

  The resource can return `VoteRollback` under any circumstances. After returning this response, the resource can forget the transaction.

  The `Resource` reports inconsistent outcomes using the `HeuristicMixed` and `HeuristicHazard` exceptions. One example is that a `Resource` reports that it can commit and later decides to roll back. Heuristic decisions must be made persistent and remembered by the `Resource` until the transaction coordinator issues the `forget` method. This method tells the `Resource` that the heuristic decision has been noted, and possibly resolved.

rollback
: The resource should undo any changes made as part of the transaction. Heuristic exceptions can be used to report heuristic decisions related to the resource. If a heuristic exception is raised, the resource must remember this outcome until the forget operation is performed so that it can return the same outcome in case rollback is performed again. Otherwise, the resource can forget the transaction.

commit
: If necessary, the resource should commit all changes made as part of this transaction. As with `rollback`, it can raise heuristic exceptions. The `NotPrepared` exception is raised if the resource has not been prepared.

commit_one_phase
: Since there can be only a single resource, the `HeuristicHazard` exception reports heuristic decisions related to that resource.

forget
: Performed after the resource raises a heuristic exception. After the coordinator determines that the heuristic situation is addressed, it issues `forget` on the resource. The resource can forget all knowledge of the transaction.

3.13. SubtransactionAwareResource

Recoverable objects that need to participate within a nested transaction may support the `SubtransactionAwareResource` interface, a specialization of the `Resource` interface.

Example 3.7. Interface `SubtransactionAwareResource`

```
interface SubtransactionAwareResource : Resource
{
    void commit_subtransaction (in Coordinator parent);
    void rollback_subtransaction ();
};
```

A recoverable object is only informed of the completion of a nested transaction if it registers a `SubtransactionAwareResource`. Register the object with either the `register_resource` of the `Coordinator` interface, or the `register_subtran_aware` method of the `Current` interface. A recoverable object registers Resources to participate within the completion of top-level transactions, and SubtransactionAwareResources keep track of the completion of subtransactions. The `commit_subtransaction` method uses a reference to the parent transaction to allow subtransaction resources to register with these transactions.

SubtransactionAwareResources find out about the completion of a transaction after it terminates. They cannot affect the outcome of the transaction. Different OTS implementations deal with exceptions raised by SubtransactionAwareResources in implementation-specific ways.

Use method `register_resource` or method `register_subtran_aware` to register a SubtransactionAwareResource with a transaction using.

register_resource
: If the transaction is a subtransaction, the resource is informed of its completion, and automatically registered with the subtransaction's parent if the parent commits.

register_subtran_aware
: If the transaction is not a subtransaction, an exception is thrown. Otherwise, the resource is informed when the subtransaction completes. Unlike `register_resource`, the resource is not propagated to the subtransaction's parent if the transaction commits. If you need this propagation, re-register using the supplied parent parameter.

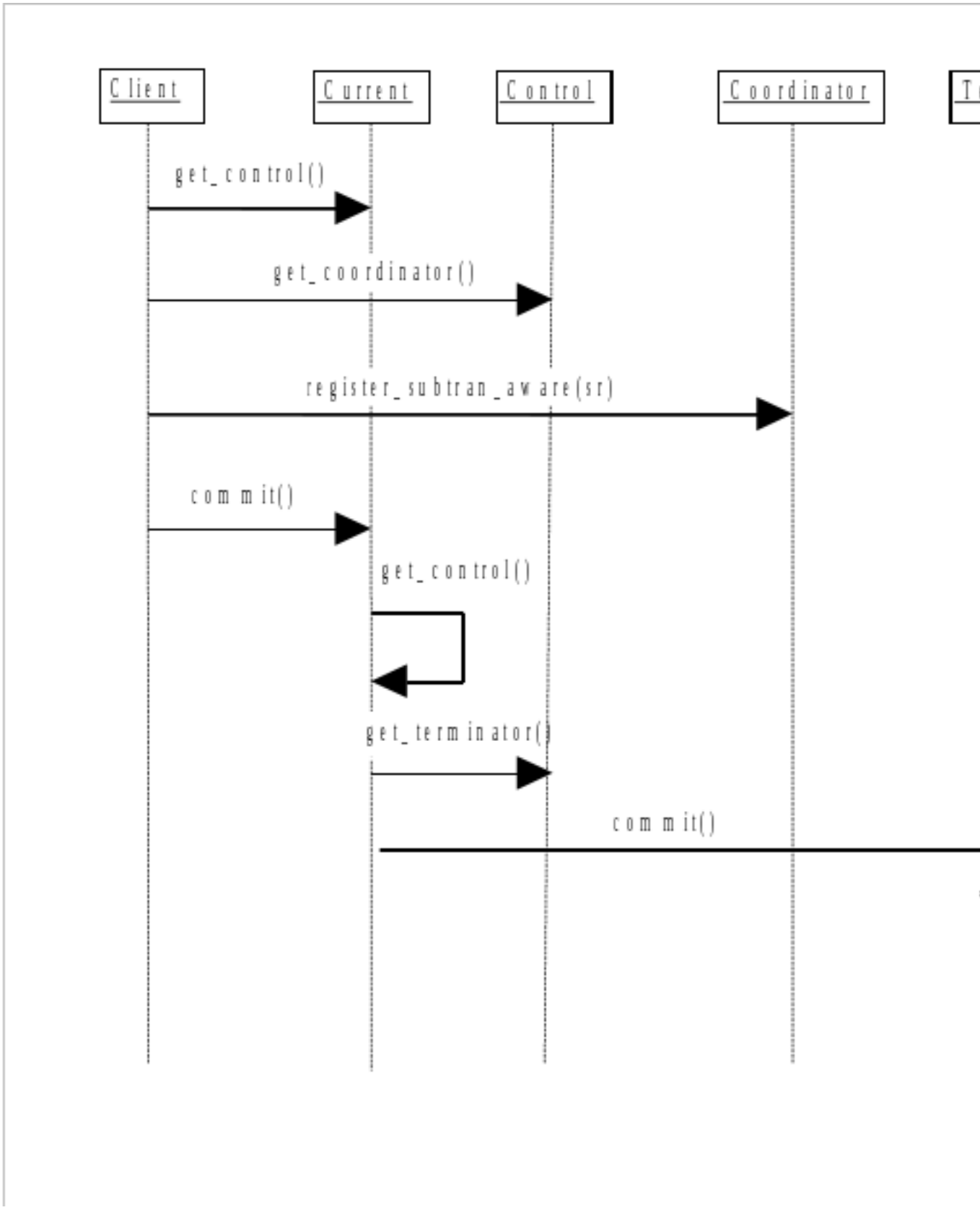

Figure 3.5. Method `register_subtran_aware`

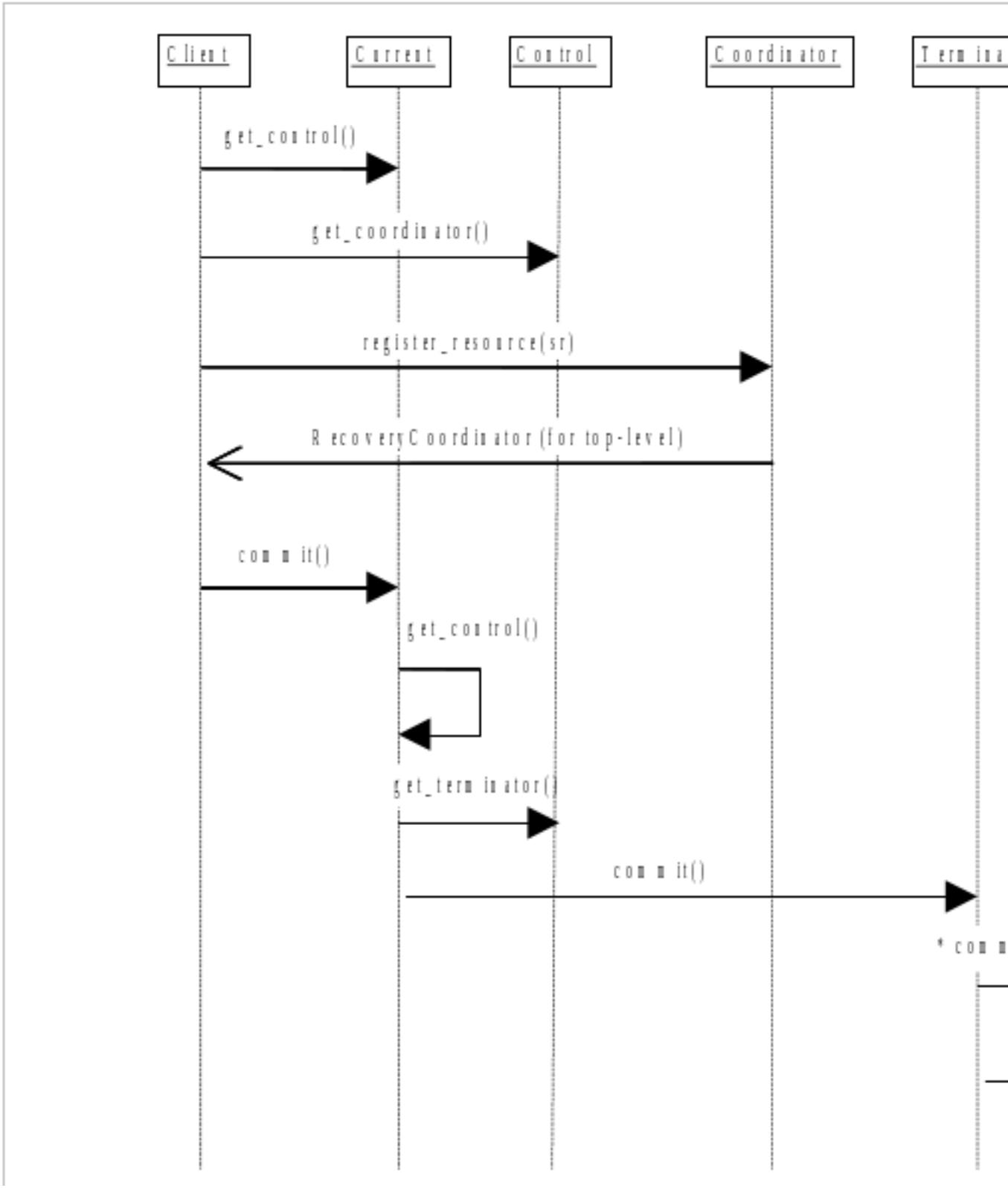

Figure 3.6. Method `register_resource`

In either case, the resource cannot affect the outcome of the transaction completion. It can only act on the transaction's decision, after the decision is made. However, if the resource cannot respond appropriately, it can raise an exception. Thee OTS handles these exceptions in an implementation-specific way.

3.13.1. JBossTS specifics

A `SubtransactionAwareResource` which raises an exception to the commitment of a transaction may create inconsistencies within the transaction if other `SubtransactionAwareResources` think the transaction committed. To prevent this possibility of inconsistency, JBossTS forces the enclosing transaction to abort if an exception is raised.

JBossTS also provides extended subtransaction aware resources to overcome this, and other problems. See Section for further details.

3.14. The `Synchronization` interface

If an object needs notification before a transaction commits, it can register an object which is an implements the `Synchronization` interface, using the `register_synchronization` operation of the `Coordinator` interface. Synchronizations flush volatile state data to a recoverable object or database before the transaction commits. You can only associate Synchronizations with top-level transactions. If you try to associate a Synchronization to a nested transaction, an exception is thrown. Each object supporting the `Synchronization` interface is associated with a single top-level transaction.

Example 3.8. Synchronization

```
interface Synchronization : TransactionalObject
{
   void before_completion ();
   void after_completion (in Status s);
};
```

The method `before_completion` is called before the two-phase commit protocol starts, and `after_completion` is called after the protocol completes. The final status of the transaction is given as a parameter to `after_completion`. If `before_completion` raises an exception, the transaction rolls back. Any exceptions thrown by `after_completion` do not affect the transaction outcome.

The OTS only requires Synchronizations to be invoked if the transaction commits. If it rolls back, registered Synchronizations are not informed.

Given the previous description of `Control`, `Resource`, `SubtransactionAwareResource`, and Synchronization, the following UML relationship diagram can be drawn:

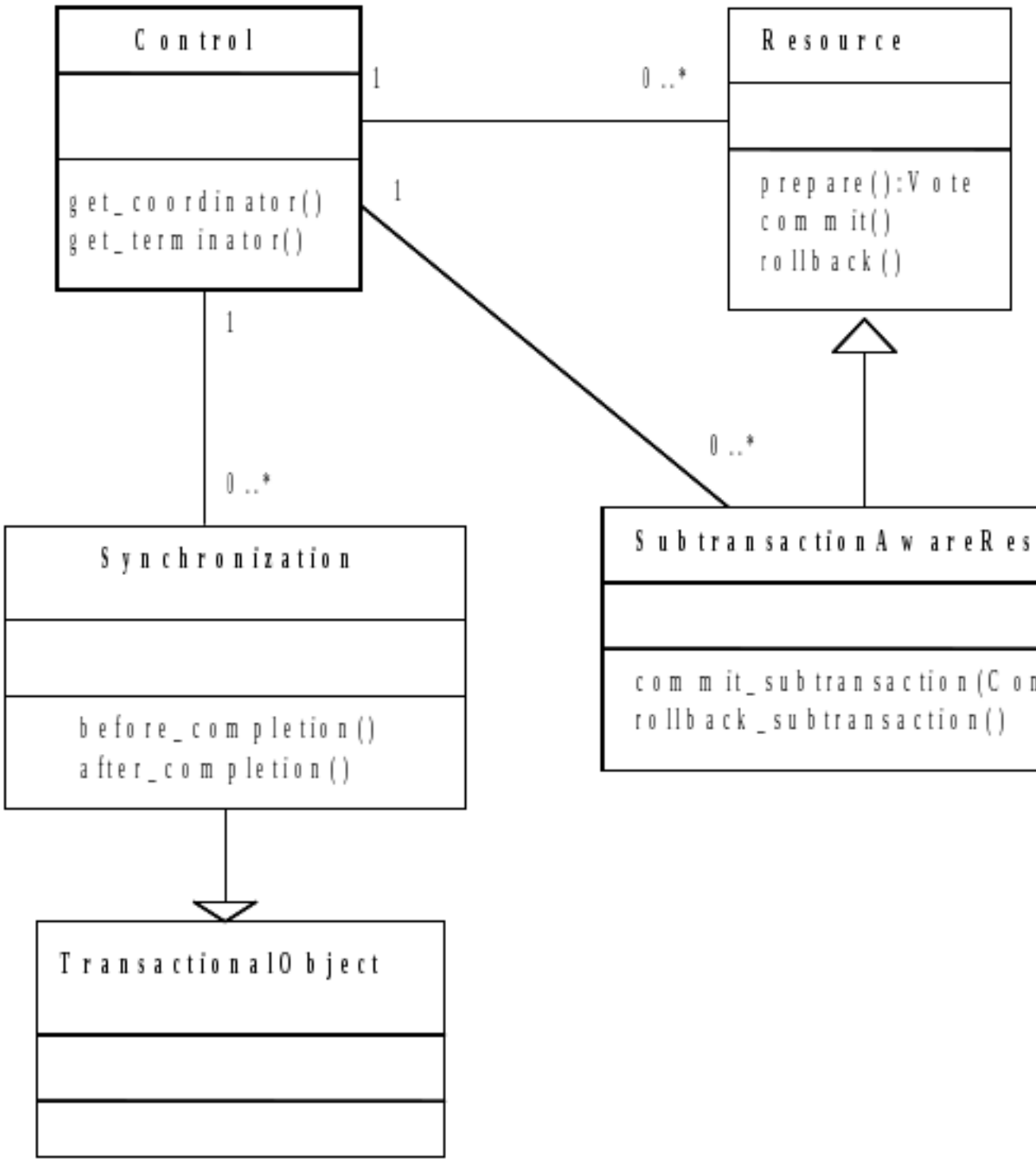

Figure 3.7. Relationship between Control, Resource, SubtransactionAwareResource, and Synchronization

3.14.1. JBossTS specifics

Synchronizations must be called before the top-level transaction commit protocol starts, and after it completes. By default, if the transaction is instructed to roll back, the Synchronizations associated with the transaction is not contacted. To override this, and call Synchronizations regardless of the transaction's outcome, set the `OTS_SUPPORT_ROLLBACK_SYNC` property variable to `YES`.

If you use distributed transactions and interposition, a local proxy for the top-level transaction coordinator is created for any recipient of the transaction context. The proxy looks like a `Resource` or `SubtransactionAwareResource`, and registers itself as such with the actual top-level transaction coordinator. The local recipient uses it to register `Resources` and `Synchronizations` locally.

The local proxy can affect how Synchronizations are invoked during top-level transaction commit. Without the proxy, all Synchronizations are invoked before any Resource or SubtransactionAwareResource objects are processed. However, with interposition, only those Synchronizations registered locally to the transaction coordinator are called. Synchronizations registered with remote participants are only called when the interposed proxy is invoked. The local proxy may only be invoked after locally-registered Resource or SubtransactionAwareResource objects are invoked. With the `OTS_SUPPORT_INTERPOSED_SYNCHRONIZATION` property variable set to `YES`, all Synchronizations are invoked before any Resource or SubtransactionAwareResource, no matter where they are registered.

3.15. Transactions and registered resources

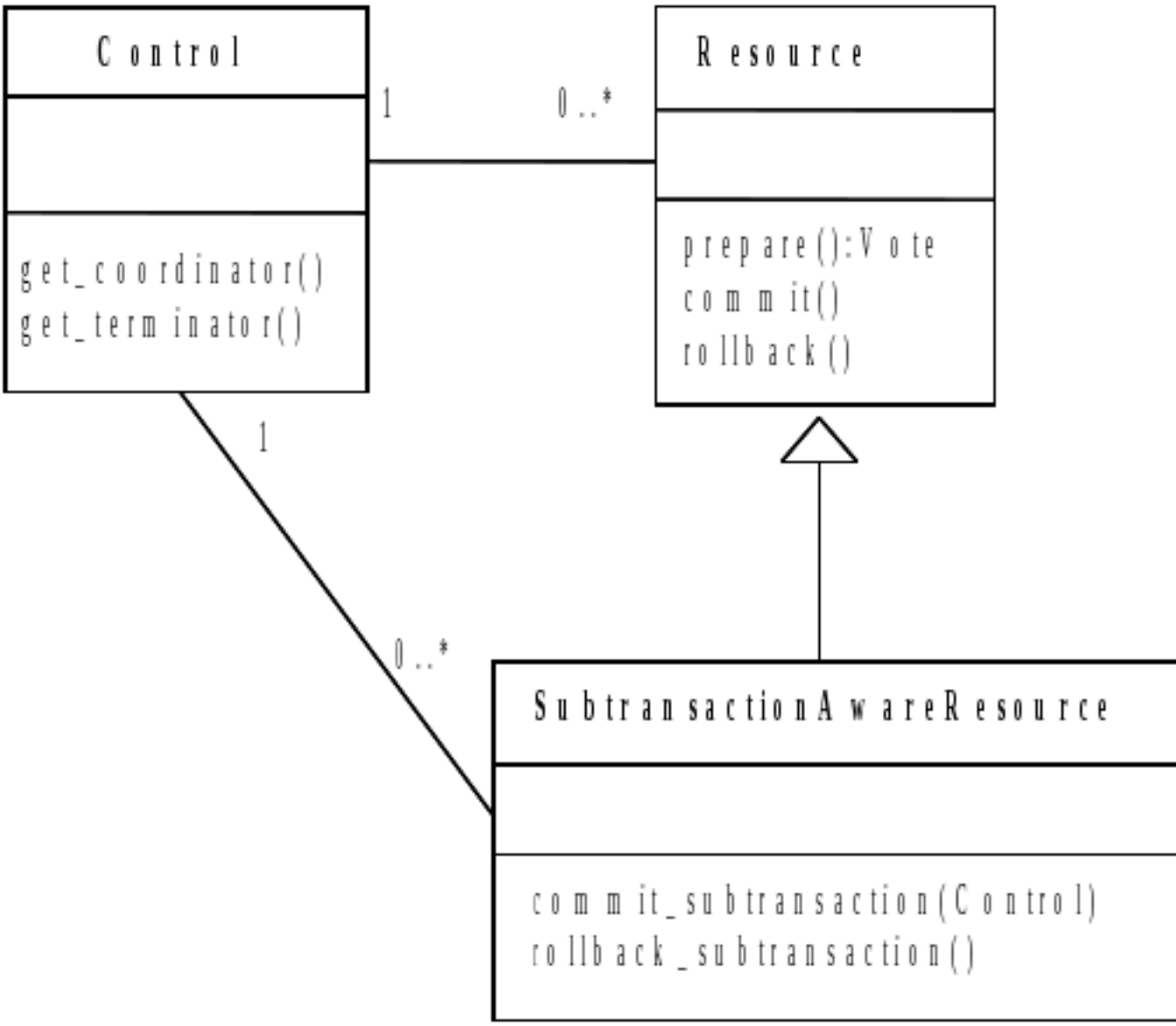

Figure 3.8. Relationship between a transaction `Control` and the resources registered with it

In *Figure 3.9, "Subtransaction commit"*, a subtransaction with both `Resource` and `SubtransactionAwareResource` objects commits. The `SubtransactionAwareResources` were registered using `register_subtran_aware`. The `Resources` do not know the subtransaction terminated, but the `SubtransactionAwareResources` do. Only the `Resources` are automatically propagated to the parent transaction.

Figure 3.9. Subtransaction commit

Figure 3.10, "Subtransaction rollback" illustrates the impact of a subtransaction rolling back. Any registered resources are discarded, and all `SubtransactionAwareResources` are informed of the transaction outcome.

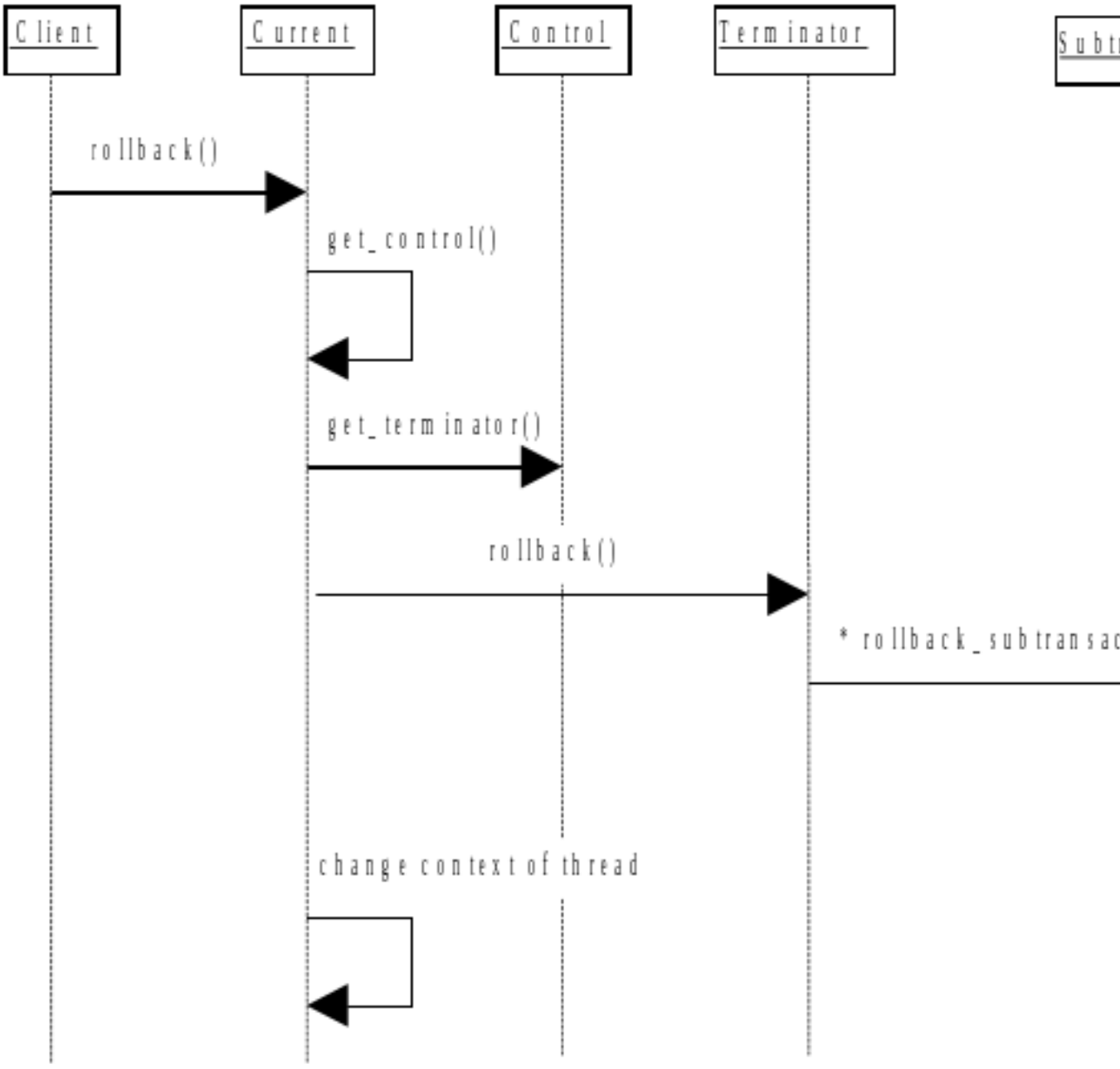

Figure 3.10. Subtransaction rollback

Figure 3.11, "Top-level commit" shows the activity diagram for committing a top-level transaction. Subtransactions within the top-level transaction which have also successfully committed propagate `SubtransactionAwareResources` to the top-level transaction. These `SubtransactionAwareResources` then participate within the two-phase commit protocol. Any registered `Synchronizations` are contacted before `prepare` is called. Because of indirect context

management, when the transaction commits, the transaction service changes the invoking thread’s transaction context.

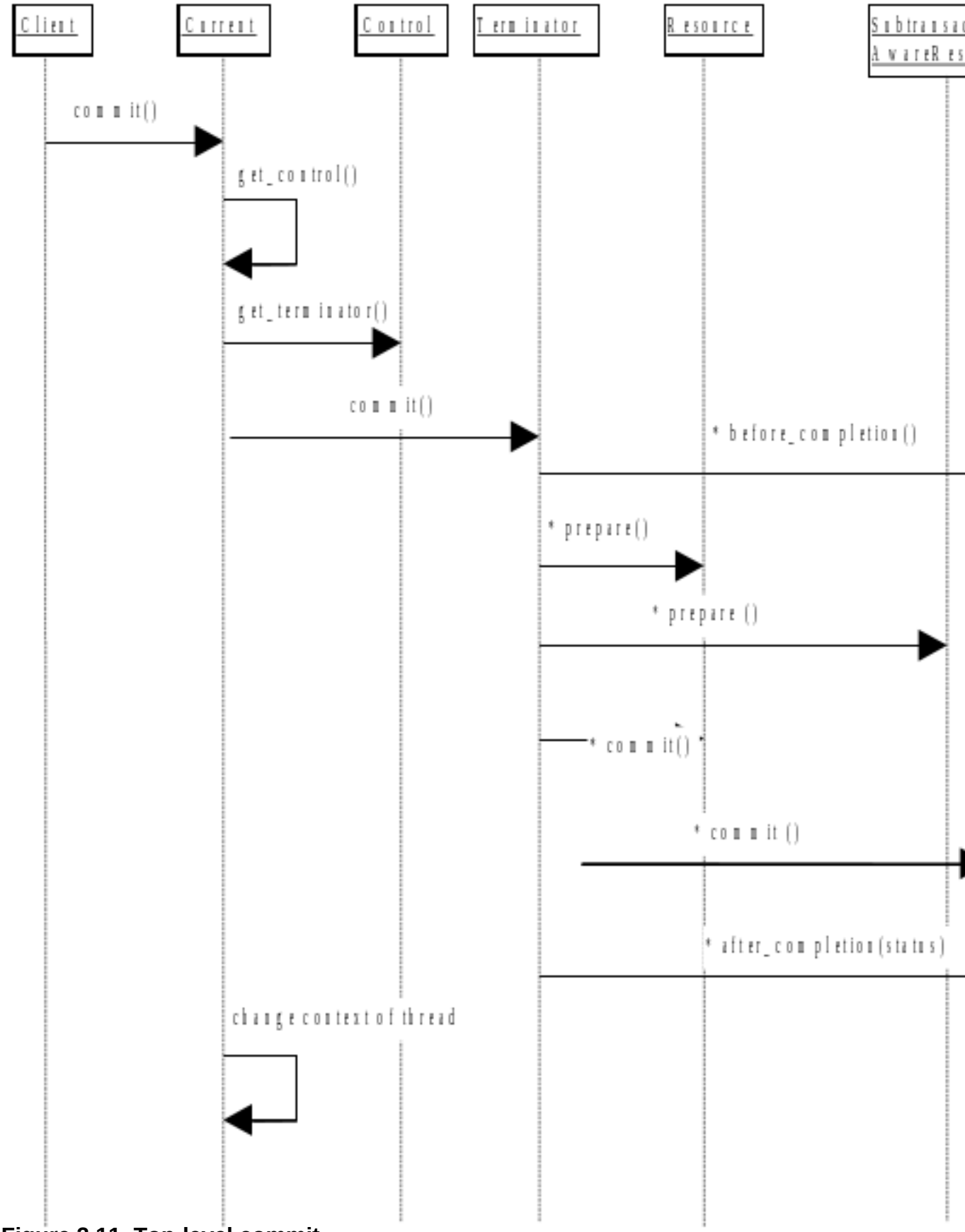

Figure 3.11. Top-level commit

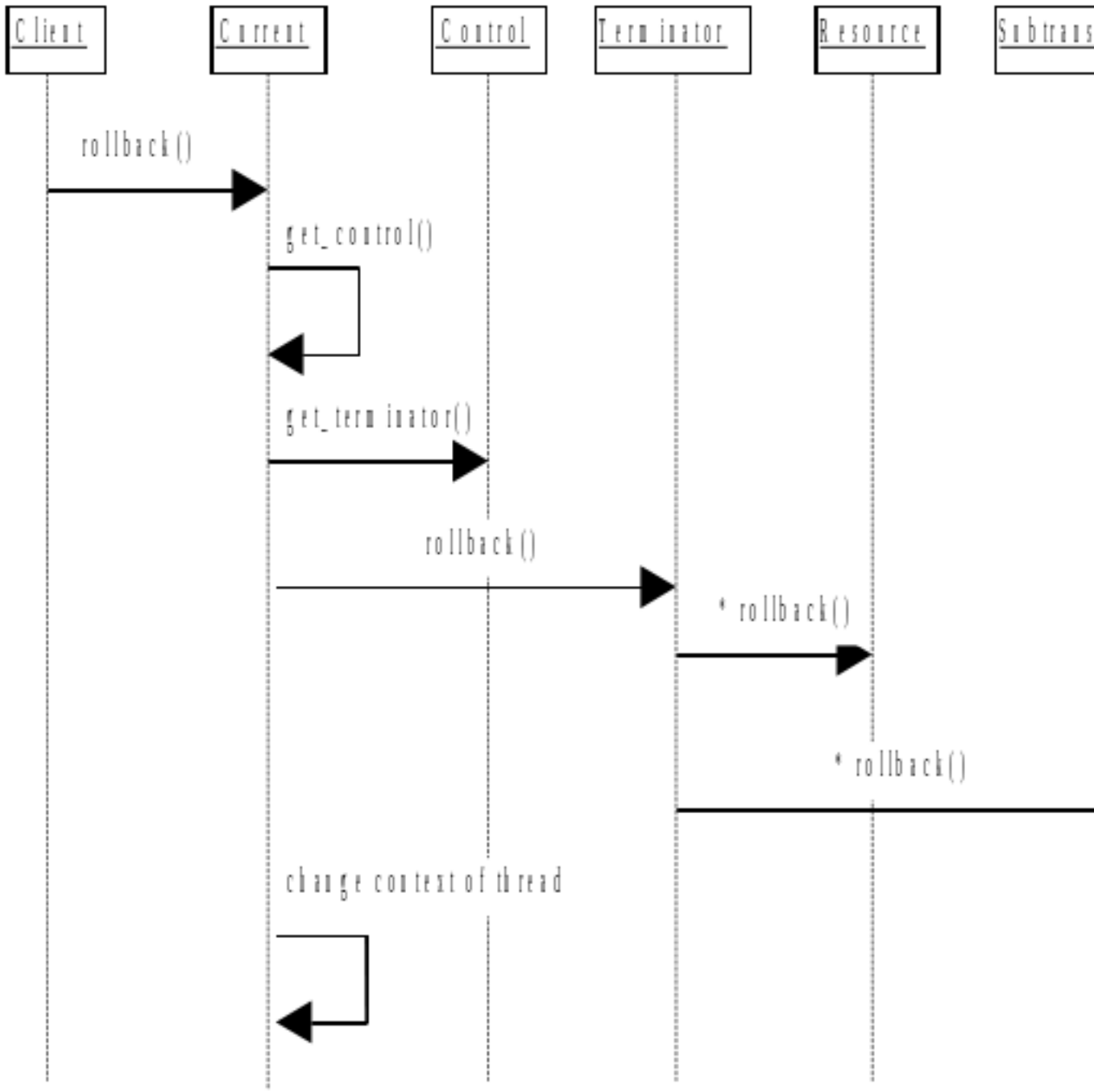

Figure 3.12. Top-level rollback

3.16. The `TransactionalObject` interface

The `TransactionalObject` interface indicates to an object that it is transactional. By supporting this interface, an object indicates that it wants to associate the transaction context associated with the client thread with all operations on its interface. The `TransactionalObject` interface defines no operations.

OTS specifications do not require an OTS to initialize the transaction context of every request handler. It is only a requirement if the interface supported by the target object is derived from `TransactionalObject`. Otherwise, the initial transaction context of the thread is undefined. A transaction service implementation can raise the `TRANSACTION_REQUIRED` exception if a `TransactionalObject` is invoked outside the scope of a transaction.

In a single-address space application, transaction contexts are implicitly shared between clients and objects, regardless of whether or not the objects support the `TransactionalObject` interface. To preserve distribution transparency, where implicit transaction propagation is supported, you can direct JBossTS to always propagate transaction contexts to objects. The default is only to propagate if the object is a `TransactionalObject`. Set the `OTS_ALWAYS_PROPAGATE_CONTEXT` property variable to `NO` to override this behavior.

By default, JBossTS does not require objects which support the `TransactionalObject` interface to invoked within the scope of a transaction. The object determines whether it should be invoked within a transaction. If so, it must throw the `TransactionRequired` exception. Override this default by setting the `OTS_NEED_TRAN_CONTEXT` shell environment variable to `YES`.

Important

Make sure that the settings for `OTS_ALWAYS_PROPAGATE_CONTEXT` and `OTS_NEED_TRAN_CONTEXT` are identical at the client and the server. If they are not identical at both ends, your application may terminate abnormally.

3.17. Interposition

OTS objects supporting interfaces such as the `Control` interface are standard CORBA objects. When an interface is passed as a parameter in an operation call to a remote server, only an object reference is passed. This ensures that any operations that the remote server performs on the interface are correctly performed on the real object. However, this can have substantial penalties for the application, because of the overhead of remote invocation. For example, when the server registers a `Resource` with the current transaction, the invocation might be remote to the originator of the transaction.

To avoid this overhead, your OTS may support interposition. This permits a server to create a local control object which acts as a local coordinator, and fields registration requests that would normally be passed back to the originator. This coordinator must register itself with the original coordinator, so that it can correctly participate in the commit protocol. Interposed coordinators form a tree structure with their parent coordinators.

To use interposition, ensure that JBossTS is correctly initialized before creating objects. Also, the client and server must both use interposition. Your ORB must support filters or interceptors, or the `CosTSPortability` interface, since interposition requires the use of implicit transaction propagation. To use interposition, set the `OTS_CONTEXT_PROP_MODE` property variable to `INTERPOSITION`.

Note

Interposition is not required if you use the JBossTS advanced API.

3.18. RecoveryCoordinator

A reference to a `RecoveryCoordinator` is returned as a result of successfully calling `register_resource` on the transaction's `Coordinator`. Each `RecoveryCoordinator` is implicitly associated with a single `Resource`. It can drive the `Resource` through recovery procedures in the event of a failure which occurs during the transaction.

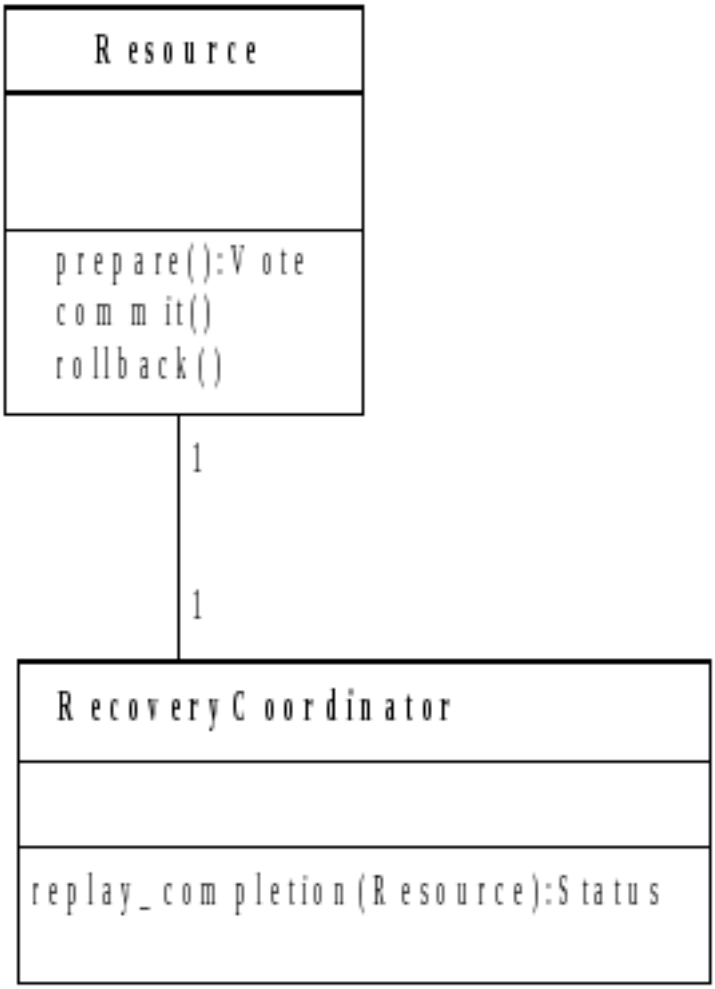

Figure 3.13. `Resource` and `RecoveryCoordinator`

3.19. Checked transaction behavior

The OTS supports both checked and unchecked transaction behavior.

Integrity constraints of checked transactions

- A transaction will not commit until all transactional objects involved in the transaction have completed their transactional requests.
- Only the transaction originator can commit the transaction

Checked transactional behavior is typical transaction behavior, and is widely implemented. Checked behavior requires implicit propagation, because explicit propagation prevents the OTS from tracking which objects are involved in the transaction.

Unchecked behavior allows you to implement relaxed models of atomicity. Any use of explicit propagation implies the possibility of unchecked behavior, since you as the programmer are in control of the behavior. Even if you use implicit propagation, a server may unilaterally abort or commit the transaction using the `Current` interface, causing unchecked behavior.

Some OTS implementations enforce checked behavior for the transactions they support, to provide an extra level of transaction integrity. The checks ensure that all transactional requests made by the application complete their processing before the transaction is committed. A checked Transaction Service guarantees that commit fails unless all transactional objects involved in the transaction complete the processing of their transactional requests. Rolling back the transaction does not require such as check, since all outstanding transactional activities will eventually roll back if they are not directed to commit.

There are many possible implementations of checking in a Transaction Service. One provides equivalent function to that provided by the request and response inter-process communication models defined by X/Open. The X/Open Transaction Service model of checking widely implemented. It describes the transaction integrity guarantees provided by many existing transaction systems. These transaction systems provide the same level of transaction integrity for object-based applications, by providing a Transaction Service interface that implements the X/Open checks.

In X/Open, completion of the processing of a request means that the object has completed execution of its method and replied to the request. The level of transaction integrity provided by a Transaction Service implementing the X/Open model provides equivalent function to that provided by the XATMI and TxRPC interfaces defined by X/Open for transactional applications. X/Open DTP Transaction Managers are examples of transaction management functions that implement checked transaction behavior.

This implementation of checked behavior depends on implicit transaction propagation. When implicit propagation is used, the objects involved in a transaction at any given time form a tree, called the request tree for the transaction. The beginner of the transaction is the root of the tree. Requests add nodes to the tree, and replies remove the replying node from the tree. Synchronous requests, or the checks described below for deferred synchronous requests, ensure that the tree collapses to a single node before commit is issued.

If a transaction uses explicit propagation, the Transaction Service has no way to know which objects are or will be involved in the transaction. Therefore, the use of explicit propagation is not permitted by a Transaction Service implementation that enforces X/Open-style checked behavior.

Applications that use synchronous requests exhibit checked behavior. If your application uses deferred synchronous requests, all clients and objects need to be under the control of a checking Transaction Service. In that case, the Transaction Service can enforce checked behavior, by applying a `reply` check and a `committed` check. The Transaction Service must also apply a `resume` check, so that the transaction is only resumed by applications in the correct part of the request tree.

reply check	Before an object replies to a transactional request, a check is made to ensure that the object has received replies to all the deferred synchronous requests that propagated the transaction in the original request. If this condition is not met, an exception is raised and the transaction is marked as rollback-only. A Transaction Service may check that a reply is issued within the context of the transaction associated with the request.
commit check	Before a commit can proceed, a check is made to ensure that the commit request for the transaction is being issued from the same execution environment that created the transaction, and that the client issuing commit has received replies to all the deferred synchronous requests it made that propagated the transaction.
resume check	Before a client or object associates a transaction context with its thread of control, a check is made to ensure that this transaction context was previously associated with the execution environment of the thread. This association would exist if the thread either created the transaction or received it in a transactional operation.

3.19.1. JBossTS specifics

Where support from the ORB is available, JBossTS supports X/Open checked transaction behavior. However, unless the `OTS_CHECKED_TRANSACTIONS` property variable is set to `YES`, checked transactions are disabled. This is the default setting.

Note

Checked transactions are only possible with a co-located transaction manager.

In a multi-threaded application, multiple threads may be associated with a transaction during its lifetime, sharing the context. In addition, if one thread terminates a transaction, other threads may still be active within it. In a distributed environment, it can be difficult to guarantee that all threads have finished with a transaction when it terminates. By default,

JBossTS issues a warning if a thread terminates a transaction when other threads are still active within it, but allow the transaction termination to continue. You can choose to block the thread which is terminating the transaction until all other threads have disassociated themselves from its context, or use other methods to solve the problem. JBossTS provides the `com.arjuna.ats.arjuna.coordinator.CheckedAction` class, which allows you to override the thread and transaction termination policy. Each transaction has an instance of this class associated with it, and you can implement the class on a per-transaction basis.

Example 3.9. `CheckedAction` implementation

```
public class CheckedAction
{
public CheckedAction ();

public synchronized void check (boolean isCommit, Uid actUid,
                                BasicList list);
};
```

When a thread attempts to terminate the transaction and there active threads exist within it, the system invokes the `check` method on the transaction's `CheckedAction` object. The parameters to the check method are:

isCommit	Indicates whether the transaction is in the process of committing or rolling back.
actUid	The transaction identifier.
list	A list of all of the threads currently marked as active within this transaction.

When `check` returns, the transaction termination continues. Obviously the state of the transaction at this point may be different from that when check was called.

Set the `CheckedAction` instance associated with a given transaction with the `setCheckedAction` method of `Current`.

3.20. Summary of JBossTS implementation decisions

- Any execution environment (thread, process) can use a transaction Control.
- `Controls`, `Coordinators`, and `Terminators` are valid for use for the duration of the transaction if implicit transaction control is used, via `Current`. If you use explicit control, via the `TransactionFactory` and `Terminator`, then use the `destroyControl` method of the OTS class in `com.arjuna.CosTransactions` to signal when the information can be garbage collected.
- You can propagate Coordinators and `Terminators` between execution environments.

- If you try to commit a transaction when there are still active subtransactions within it, JBossTS rolls back the parent and the subtransactions.
- JBossTS includes full support for nested transactions. However, if a resource raises an exception to the commitment of a subtransaction after other resources have previously been told that the transaction committed, JBossTS forces the enclosing transaction to abort. This guarantees that all resources used within the subtransaction are returned to a consistent state. You can disable support for subtransactions by setting the `OTS_SUPPORT_SUBTRANSACTIONS` variable to `NO`.
- Obtain `Current` from the `get_current` method of the OTS.
- A timeout value of zero seconds is assumed for a transaction if none is specified using `set_timeout`.
- by default, `Current` does not use a separate transaction manager server by default. Override this behavior by setting the `OTS_TRANSACTION_MANAGER` environment variable. Location of the transaction manager is ORB-specific.
- Checked transactions are disabled by default. To enable them, set the `OTS_CHECKED_TRANSACTIONS` property to `YES`.

Constructing an OTS application

4.1. Important notes for JBossTS

4.1.1. Initialization

JBossTS must be correctly initialized before you create any application object. To guarantee this, use the `initORB` and `POA` methods described in the *Orb Portability Guide*. Consult the *Orb Portability Guide* if you need direct use of the `ORB_init` and `create_POA` methods provided by the underlying ORB.

4.1.2. Implicit context propagation and interposition

If you need implicit context propagation and interposition, initialize JBossTS correctly before you create any objects. You can only use implicit context propagation on an ORB which supports filters and interceptors, or the `CosTSPortability` interface. You can set `OTS_CONTEXT_PROP_MODE` to `CONTEXT` or `INTERPOSITION`, depending on which functionality you need. If you are using the JBossTS API, you need to use interposition.

4.2. Writing applications using the raw OTS interfaces

Steps to participate in an OTS transaction

- Create `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction or subtransaction. These resources manage the persistence, concurrency control, and recovery for the object. The OTS invokes these objects during the prepare, commit, or abort phase of the transaction or subtransaction, and the Resources perform the work of the application.
- Register `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensure that the object is only registered once within a given transaction. As part of registration, a `Resource` receives a reference to a `RecoveryCoordinator`. This reference must be made persistent, so that the transaction can recover in the event of a failure.
- Correctly propagate resources such as locks to parent transactions and `SubtransactionAwareResource` objects.
- Drive the crash recovery for each resource which was participating within the transaction, in the event of a failure.

The OTS does not provide any `Resource` implementations. You need to provide these implementations. The interfaces defined within the OTS specification are too low-level for most situations. JBossTS is designed to make use of raw *Common Object Services (COS)* interfaces, but provides a higher-level API for building transactional applications and framework. This API automates much of the work involved with participating in an OTS transaction.

4.3. Transaction context management

If you use implicit transaction propagation, ensure that appropriate objects support the `TransactionalObject` interface. Otherwise, you need to pass the transaction contexts as parameters to the relevant operations.

4.3.1. A transaction originator: indirect and implicit

Example 4.1. Indirect and implicit transaction originator

```
...
txn_crt.begin();
// should test the exceptions that might be raised
...
// the client issues requests, some of which involve
// transactional objects;
BankAccount1.makeDeposit(deposit);
...
```

A transaction originator uses indirect context management and implicit transaction propagation. `txn_crt` is a pseudo object supporting the `Current` interface. The client uses the `begin` operation to start the transaction, which becomes implicitly associated with the originator's thread of control.

The program commits the transaction associated with the client thread. The `report_heuristics` argument is set to `false`, so the Transaction Service makes no reports about possible heuristic decisions.

```
...
txn_crt.commit(false);
...
```

4.3.2. Transaction originator: direct and explicit

Example 4.2. Direct and explicit transaction originator

```
...
org.omg.CosTransactions.Control c;
org.omg.CosTransactions.Terminator t;
org.omg.CosTransactions.Coordinator co;
org.omg.CosTransactions.PropagationContext pgtx;

c = TFactory.create(0);
t = c.get_terminator();
```

```
    pgtx = c.get_coordinator().get_txcontext();
    ...
```

This transaction originator uses direct context management and explicit transaction propagation. The client uses a factory object supporting the `CosTransactions::TransactionFactory` interface to create a new transaction, and uses the returned `Control` object to retrieve the `Terminator` and `Coordinator` objects.

The client issues requests, some of which involve transactional objects. This example uses explicit propagation of the context. The `Control` object reference is passed as an explicit parameter of the request. It is declared in the OMG IDL of the interface.

```
    ...
    transactional_object.do_operation(arg, pgtx);
```

The transaction originator uses the `Terminator` object to commit the transaction. The `report_heuristics` argument is set to `false`, so the Transaction Service makes no reports about possible heuristic decisions.

```
    ...
    t.commit(false);
```

4.4. Implementing a transactional client

The `commit` operation of `Current` or the `Terminator` interface takes the boolean `report_heuristics` parameter. If the `report_heuristics` argument is `false`, the commit operation can complete as soon as the `Coordinator` makes the decision to commit or roll back the transaction. The application does not need to wait for the `Coordinator` to complete the commit protocol by informing all the participants of the outcome of the transaction. This can significantly reduce the elapsed time for the commit operation, especially where participant `Resource` objects are located on remote network nodes. However, no heuristic conditions can be reported to the application in this case.

Using the `report_heuristics` option guarantees that the commit operation does not complete until the `Coordinator` completes the commit protocol with all `Resource` objects involved in the transaction. This guarantees that the application is informed of any non-atomic outcomes of the transaction, through one of the exceptions `HeuristicMixed` or `HeuristicHazard`. However, it increases the application-perceived elapsed time for the commit operation.

4.5. Implementing a recoverable server

A Recoverable Server includes at least one transactional object and one resource object, each of which have distinct responsibilities.

4.5.1. Transactional object

The transactional object implements the transactional object's operations and registers a `Resource` object with the `Coordinator`, so that the Recoverable Server's resources, including any necessary recovery, can commit.

The `Resource` object identifies the involvement of the Recoverable Server in a particular transaction. This requires a `Resource` object to only be registered in one transaction at a time. Register a different `Resource` object for each transaction in which a recoverable server is concurrently involved. A transactional object may receive multiple requests within the scope of a single transaction. It only needs to register its involvement in the transaction once. The `is_same_transaction` operation allows the transactional object to determine if the transaction associated with the request is one in which the transactional object is already registered.

The `hash_transaction` operations allow the transactional object to reduce the number of transaction comparisons it has to make. All `Coordinators` for the same transaction return the same hash code. The `is_same_transaction` operation only needs to be called on `Coordinators` with the same hash code as the `Coordinator` of the current request.

4.5.2. Resource object

A `Resource` object participates in the completion of the transaction, updates the resources of the Recoverable Server in accordance with the transaction outcome, and ensures termination of the transaction, including across failures.

4.5.3. Reliable servers

A *Reliable Server* is a special case of a Recoverable Server. A Reliable Server can use the same interface as a Recoverable Server to ensure application integrity for objects that do not have recoverable state. In the case of a Reliable Server, the transactional object can register a `Resource` object that replies `VoteReadOnly` to `prepare` if its integrity constraints are satisfied. It replies `VoteRollback` if it finds a problem. This approach allows the server to apply integrity constraints which apply to the transaction as a whole, rather than to individual requests to the server.

4.5.4. Examples

Example 4.3. Reliable server

```
/*
  BankAccount1 is an object with internal resources. It inherits from both the
 TransactionalObject and the Resource interfaces:
*/
interface BankAccount1:
                CosTransactions::TransactionalObject, CosTransactions::Resource
{
    ...
```

```
    void makeDeposit (in float amt);
    ...
};
/* The corresponding Java class is: */
public class BankAccount1
{
public void makeDeposit(float amt);
    ...
};
/*
  Upon entering, the context of the transaction is implicitly associated with
 the object#s thread. The pseudo object
  supporting the Current interface is used to retrieve the Coordinator object
 associated with the transaction.
*/
void makeDeposit (float amt)
{
    org.omg.CosTransactions.Control c;
    org.omg.CosTransactions.Coordinator co;
    c = txn_crt.get_control();
    co = c.get_coordinator();
    ...
/*
  Before registering the resource the object should check whether it has already
 been registered for the same
  transaction. This is done using the hash_transaction and is_same_transaction
 operations.  that this object registers
  itself as a resource. This imposes the restriction that the object may only
 be involved in one transaction at a
  time. This is not the recommended way for recoverable objects to participate
 within transactions, and is only used as an
  example.  If more parallelism is required, separate resource objects should
 be registered for involvement in the same
  transaction.
*/
    RecoveryCoordinator r;
    r = co.register_resource(this);

    // performs some transactional activity locally
    balance = balance + f;
    num_transactions++;
    ...
    // end of transactional operation
};
```

Example 4.4. Transactional object

```
/*  A BankAccount2 is an object with external resources that inherits from the
 TransactionalObject interface: */
interface BankAccount2: CosTransactions::TransactionalObject
{
    ...
    void makeDeposit(in float amt);
    ...
};

public class BankAccount2
{
public void makeDeposit(float amt);
    ...
}
/*
Upon entering, the context of the transaction is implicitly associated with the
 object#s thread. The makeDeposit
operation performs some transactional requests on external, recoverable servers.
 The objects res1 and res2 are
recoverable objects. The current transaction context is implicitly propagated
 to these objects.
*/
void makeDeposit(float amt)
{
    balance = res1.get_balance(amt);
    balance = balance + amt;
    res1.set_balance(balance);
    res2.increment_num_transactions();
} // end of transactional operation
```

4.6. Failure models

The Transaction Service provides atomic outcomes for transactions in the presence of application, system or communication failures. From the viewpoint of each user object role, two types of failure are relevant:

- A local failure, which affects the object itself.
- An external failure, such as failure of another object or failure in the communication with an object.

The transaction originator and transactional server handle these failures in different ways.

4.6.1. Transaction originator

Local failure

If a Transaction originator fails before the originator issues `commit`, the transaction is rolled back. If the originator fails after issuing commit and before the outcome is reported, the transaction can either commit or roll back, depending on timing. In this case, the transaction completes without regard to the failure of the originator.

External failure

Any external failure which affects the transaction before the originator issues `commit` causes the transaction to roll back. The standard exception `TransactionRolledBack` is raised in the originator when it issues `commit`.

If a failure occurs after commit and before the outcome is reported, the client may not be informed of the outcome of the transaction. This depends on the nature of the failure, and the use of the `report_heuristics` option of `commit`. For example, the transaction outcome is not reported to the client if communication between the client and the `Coordinator` fails.

A client can determine the outcome of the transaction by using method `get_status` on the `Coordinator`. However, this is not reliable because it may return the status `NoTransaction`, which is ambiguous. The transaction could have committed and been forgotten, or it could have rolled back and been forgotten.

An originator is only guaranteed to know the transaction outcome in one of two ways.

- if its implementation includes a `Resource` object, so that it can participate in the two-phase commit procedure.
- The originator and `Coordinator` must be located in the same failure domain.

4.6.2. Transactional server

Local failure

If the Transactional Server fails, optional checks by a Transaction Service implementation may make the transaction to roll back. Without such checks, whether the transaction rolls back depends on whether the commit decision is already made, such as when an unchecked client invokes `commit` before receiving all replies from servers.

External failure

Any external failure affecting the transaction during the execution of a Transactional Server causes the transaction to be rolled back. If the failure occurs while the transactional object's method is executing, the failure has no effect on the execution of this method. The method may terminate normally, returning the reply to its client. Eventually the `TransactionRolledBack` exception is returned to a client issuing `commit`.

Recoverable server

Behavior of a recoverable server when failures occur is determined by the two phase commit protocol between the `Coordinator` and the recoverable server's `Resource` object.

4.7. Summary

When you develop OTS applications which use the raw OTS interfaces, be aware of the following items:

- Create `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction or subtransaction. These resources handle the persistence, concurrency control, and recovery for the object. The OTS invokes these objects during the prepare, commit, and abort phases of the transaction or subtransaction, and the `Resources` then perform all appropriate work.
- Register `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensure that the object is only registered once within a given transaction. As part of registration, a `Resource` receives a reference to a `RecoveryCoordinator`, which must be made persistent so that recovery can occur in the event of a failure.
- For nested transactions, make sure that any propagation of resources, such as locks to parent transactions, are done correctly. You also need to manage propagation of `SubtransactionAwareResource` objects to parents.
- in the event of failures, drive the crash recovery for each `Resource` which participates within the transaction.

The OTS does not provide any `Resource` implementations.

JBossTS interfaces for extending the OTS

This chapter contains a description of the use of the JBossTS classes you can use to extend the OTS interfaces. These advanced interfaces are all written on top of the basic OTS engine described previously, and applications which use them run on other OTS implementations, only without the added functionality.

Features

AtomicTransaction
: Provides a more manageable interface to the OTS transaction than `CosTransactions::Current`. It automatically keeps track of transaction scope, and allows you to create nested top-level transactions in a more natural manner than the one provided by the OTS.

Advanced subtransaction-Resource classes
: Allow nested transactions to use a two-phase commit protocol. These Resources can also be ordered within JBossTS, enabling you to control the order in which `Resources` are called during the commit or abort protocol.

Implicit context propagation between client and server
: Where available, JBossTS uses implicit context propagation between client and server. Otherwise, JBossTS provides an explicit interposition class, which simplifies the work involved in interposition. The JBossTS API, *Transactional Objects for Java (TXOJ)*, requires either explicit or implicit interposition. This is even true in a stand-alone mode when using a separate transaction manager. TXOJ is fully described in the *ArjunaCore Development Guide*.

Note

the extensions to the `CosTransactions.idl` are located in the com.arjuna.ArjunaOTS package and the `ArjunaOTS.idl` file.

5.1. Nested transactions

The OTS implementation of nested transactions is extremely limited, and can lead to the generation of inconsistent results. One example is a scenario in which a subtransaction coordinator discovers part of the way through committing that a resources cannot commit. It may not be able to tell the committed resources to abort.

In most transactional systems which support subtransactions, the subtransaction commit protocol is the same as a top-level transaction's. There are two phases, a `prepare` phase and a `commit`

or `abort` phase. Using a multi-phase commit protocol avoids the above problem of discovering that one resources cannot commit after others have already been told to commit. The `prepare` phase generates consensus on the commit outcome, and the `commit` or `abort` phase enforces the outcome.

JBossTS supports the strict OTS implementation of subtransactions for those resources derived from `CosTransactions::SubtransactionAwareResource`. However, if a resource is derived from `ArjunaOTS::ArjunaSubtranAwareResource`, it is driven by a two-phase commit protocol whenever a nested transaction commits.

Example 5.1. ArjunaSubtranAwareResource

```
interface ArjunaSubtranAwareResource :
    CosTransactions::SubtransactionAwareResource
{
    CosTransactions::Vote prepare_subtransaction ();
};
```

During the first phase of the commit protocol the `prepare_subtransaction` method is called, and the resource behaves as though it were being driven by a top-level transaction, making any state changes provisional upon the second phase of the protocol. Any changes to persistent state must still be provisional upon the second phase of the top-level transaction, as well. Based on the votes of all registered resources, JBossTS then calls either `commit_subtransaction` or `rollback_subtransaction`.

Note

This scheme only works successfully if all resources registered within a given subtransaction are instances of the `ArjunaSubtranAwareResource` interface, and that after a resource tells the coordinator it can prepare, it does not change its mind.

5.2. Extended resources

When resources are registered with a transaction, the transaction maintains them within a list, called the *intentions list*. At termination time, the transaction uses the intentions list to drive each resource appropriately, to commit or abort. However, you have no control over the order in which resources are called, or whether previously-registered resources should be replaced with newly registered resources. The JBossTS interface `ArjunaOTS::OTSAbstractRecord` gives you this level of control.

Example 5.2. OTSAbstractRecord

```
interface OTSAbstractRecord : ArjunaSubtranAwareResource
```

```
{
    readonly attribute long typeId;
    readonly attribute string uid;

    boolean propagateOnAbort ();
    boolean propagateOnCommit ();

    boolean saveRecord ();
    void merge (in OTSAbstractRecord record);
    void alter (in OTSAbstractRecord record);

    boolean shouldAdd (in OTSAbstractRecord record);
    boolean shouldAlter (in OTSAbstractRecord record);
    boolean shouldMerge (in OTSAbstractRecord record);
    boolean shouldReplace (in OTSAbstractRecord record);
};
```

typeId	returns the record type of the instance. This is one of the values of the enumerated type Record_type.
uid	a stringified Uid for this record.
propagateOnAbort	by default, instances of `OTSAbstractRecord` should not be propagated to the parent transaction if the current transaction rolls back. By returning `TRUE`, the instance will be propagated.
propagateOnCommit	returning `TRUE` from this method causes the instance to be propagated to the parent transaction if the current transaction commits. Returning `FALSE` disables the propagation.
saveRecord	returning `TRUE` from this method causes JBossTS to try to save sufficient information about the record to persistent state during commit, so that crash recovery mechanisms can replay the transaction termination in the event of a failure. If `FALSE` is returned, no information is saved.
merge	used when two records need to merge together.
alter	used when a record should be altered.
shouldAdd	returns `true` ii the record should be added to the list, `false` if it should be discarded.
shouldMerge	returns `true` if the two records should be merged into a single record, `false` otherwise.
shouldReplace	returns `true` if the record should replace an existing one, `false` otherwise.

When inserting a new record into the transaction's intentions list, JBossTS uses the following algorithm:

1. if a record with the same type and uid has already been inserted, then the methods `shouldAdd`, and related methods, are invoked to determine whether this record should also be added.
2. If no such match occurs, then the record is inserted in the intentions list based on the `type` field, and ordered according to the uid. All of the records with the same type appear ordered in the intentions list.

`OTSAbstractRecord` is derived from `ArjunaSubtranAwareResource`. Therefore, all instances of `OTSAbstractRecord` inherit the benefits of this interface.

5.3. AtomicTransaction

In terms of the OTS, `AtomicTransaction` is the preferred interface to the OTS protocol engine. It is equivalent to `CosTransactions::Current`, but with more emphasis on easing application development. For example, if an instance of `AtomicTransaction` goes out of scope before it is terminates, the transaction automatically rolls back. `CosTransactions::Current` cannot provide this functionality. When building applications using JBossTS, use `AtomicTransaction` for the added benefits it provides. It is located in the com.arjuna.ats.jts.extensions.ArjunaOTS package.

Example 5.3. AtomicTransaction

```
public class AtomicTransaction
{
    public AtomicTransaction ();
    public void begin () throws SystemException, SubtransactionsUnavailable,
                                NoTransaction;
    public void commit (boolean report_heuristics) throws SystemException,
                                                    NoTransaction, HeuristicMixed,
                                            HeuristicHazard,TransactionRolledBack;
    public void rollback () throws SystemException, NoTransaction;
    public Control control () throws SystemException, NoTransaction;
    public Status get_status () throws SystemException;
    /* Allow action commit to be supressed */
    public void rollbackOnly () throws SystemException, NoTransaction;

    public void registerResource (Resource r) throws SystemException, Inactive;
    public void
        registerSubtransactionAwareResource (SubtransactionAwareResource)
        throws SystemException, NotSubtransaction;
    public void
        registerSynchronization(Synchronization s) throws SystemException,
                                                        Inactive;
};
```

Table 5.1. AtomicTransaction's Methods

begin	Starts an action

commit	Commits an action
rollback	Abort an action

Transaction nesting is determined dynamically. Any transaction started within the scope of another running transaction is nested.

The `TopLevelTransaction` class, which is derived from `AtomicTransaction`, allows creation of nested top-level transactions. Such transactions allow non-serializable and potentially non-recoverable side effects to be initiated from within a transaction, so use them with caution. You can create nested top-level transactions with a combination of the `CosTransactions::TransactionFactory` and the `suspend` and `resume` methods of `CosTransactions::Current`. However, the `TopLevelTransaction` class provides a more user-friendly interface.

`AtomicTransaction` and `TopLevelTransaction` are completely compatible with `CosTransactions::Current`. You an use the two transaction mechanisms interchangeably within the same application or object.

`AtomicTransaction` and `TopLevelTransaction` are similar to `CosTransactions::Current`. They both simplify the interface between you and the OTS. However, you gain two advantages by using `AtomicTransaction` or `TopLevelTransaction`.

- The ability to create nested top-level transactions which are automatically associated with the current thread. When the transaction ends, the previous transaction associated with the thread, if any, becomes the thread's current transaction.
- Instances of `AtomicTransaction` track scope, and if such an instance goes out of scope before it is terminated, it is automatically aborted, along with its children.

5.4. Context propagation issues

When using TXOJ in a distributed manner, JBossTS requires you to use interposition between client and object. This requirement also exists if the application is local, but the transaction manager is remote. In the case of implicit context propagation, where the application object is derived from `CosTransactions::TransactionalObject`, you do not need to do anything further. JBossTS automatically provides interposition. However, where implicit propagation is not supported by the ORB, or your application does not use it, you must take additional action to enable interposition.

The class `com.arjuna.ats.jts.ExplicitInterposition` allows an application to create a local control object which acts as a local coordinator, fielding registration requests that would normally be passed back to the originator. This surrogate registers itself with the original coordinator, so that it can correctly participate in the commit protocol. The application thread context becomes the surrogate transaction hierarchy. Any transaction context currently associated with the thread is lost. The interposition lasts for the lifetime of the explicit interposition object, at which point the application thread is no longer associated with a transaction context. Instead, it is set to `null`.

interposition is intended only for those situations where the transactional object and the transaction occur within different processes, rather than being co-located. If the transaction is created locally to the client, do not use the explicit interposition class. The transaction is implicitly associated with the transactional object because it resides within the same process.

Example 5.4. ExplicitInterposition

```
public class ExplicitInterposition
{
    public ExplicitInterposition ();

    public void registerTransaction (Control control) throws InterpositionFailed, SystemExcepti

    public void unregisterTransaction () throws InvalidTransaction,
                                                SystemException;
};
```

A transaction context can be propagated between client and server in two ways: either as a reference to the client's transaction Control, or explicitly sent by the client. Therefore, there are two ways in which the interposed transaction hierarchy can be created and registered. For example, consider the class Example which is derived from LockManager and has a method increment:

Example 5.5. ExplicitInterposition Example

```
public boolean increment (Control control)
{
    ExplicitInterposition inter = new ExplicitInterposition();

    try
        {
            inter.registerTransaction(control);
        }
    catch (Exception e)
        {
            return false;
        }

    // do real work

    inter.unregisterTransaction();  // should catch exceptions!

    // return value dependant upon outcome
}
```

if the `Control` passed to the `register` operation of `ExplicitInterposition` is `null`, no exception is thrown. The system assumes that the client did not send a transaction context to the server. A transaction created within the object will thus be a top-level transaction.

When the application returns, or when it finishes with the interposed hierarchy, the program should call `unregisterTransaction` to disassociate the thread of control from the hierarchy. This occurs automatically when the `ExplicitInterposition` object is garbage collected. However, since this may be after the transaction terminates, JBossTS assumes the thread is still associated with the transaction and issues a warning about trying to terminate a transaction while threads are still active within it.

Example

This example illustrates the concepts and the implementation details for a simple client/server example using implicit context propagation and indirect context management.

6.1. The basic example

This example only includes a single unit of work within the scope of the transaction. consequently, only a one-phase commit is needed.

The client and server processes are both invoked using the `implicit propagation` and `interposition` command-line options.

For the purposes of this worked example, a single method implements the `DemoInterface` interface. This method is used in the DemoClient program.

Example 6.1. idl interface

```
#include <idl/CosTransactions.idl>
#pragma javaPackage ""


module Demo
{
    exception DemoException {};

    interface DemoInterface : CosTransactions::TransactionalObject
    {
        void work() raises (DemoException);
    };
};
```

6.1.1. Example implementation of the interface

This section deals with the pieces needed to implement the example interface.

6.1.1.1. Resource

The example overrides the methods of the `Resource` implementation class. The `DemoResource` implementation includes the placement of `System.out.println` statements at judicious points, to highlight when a particular method is invoked.

Only a single unit of work is included within the scope of the transaction. Therefore, the `prepare` or `commit` methods should never be invoked, but the `commit_one_phase` method should be invoked.

Example 6.2. DemoResource

```
1    import org.omg.CosTransactions.*;
2    import org.omg.CORBA .SystemException;
3
4    public class DemoResource extends  org.omg.CosTransactions .ResourcePOA
5    {
6        public Vote prepare() throws HeuristicMixed, HeuristicHazard,
7        SystemException
8        {
9            System.out.println("prepare called");
10
11           return Vote.VoteCommit;
12       }
13
14       public void rollback() throws HeuristicCommit, HeuristicMixed,
15       HeuristicHazard, SystemException
16       {
17           System.out.println("rollback called");
18       }
19
20       public void commit() throws NotPrepared, HeuristicRollback,
21       HeuristicMixed, HeuristicHazard, SystemException
22       {
23           System.out.println("commit called");
24       }
25
26       public void commit_one_phase() throws HeuristicHazard, SystemException
27       {
28           System.out.println("commit_one_phase called");
29       }
30
31       public void forget() throws SystemException
32       {
33           System.out.println("forget called");
34       }
35   }
```

6.1.1.2. Transactional implementation

At this stage, the `Demo.idl` has been processed by the ORB's idl compiler to generate the necessary client and server package.

Line 14 returns the transactional context for the `Current` pseudo object. After obtaining a `Control` object, you can derive the Coordinator object (line 16).

Lines 17 and 19 create a resource for the transaction, and then inform the ORB that the resource is ready to receive incoming method invocations.

Line 20 uses the `Coordinator` to register a `DemoResource` object as a participant in the transaction. When the transaction terminates, the resource receives requests to commit or rollback the updates performed as part of the transaction.

Example 6.3. Transactional implementation

```
1    import Demo.*;
2    import org.omg.CosTransactions.*;
3    import com.arjuna.ats.jts.*;
4    import com.arjuna.orbportability.*;
5
6    public class DemoImplementation extends Demo.DemoInterfacePOA
7    {
8        public void work() throws DemoException
9        {
10           try
11           {
12
13               Control control = OTSManager.get_current().get_control();
14
15               Coordinator  coordinator = control.get_coordinator();
16               DemoResource resource    = new DemoResource();
17
18               ORBManager.getPOA().objectIsReady(resource);
19               coordinator.register_resource(resource);
20
21           }
22           catch (Exception e)
23           {
24               throw new DemoException();
25           }
26       }
27
28   }
```

6.1.1.3. Server implementation

First, you need to to initialize the ORB and the POA. Lines 10 through 14 accomplish these tasks.

The servant class `DemoImplementation` contains the implementation code for the `DemoInterface` interface. The servant services a particular client request. Line 16 instantiates a servant object for the subsequent servicing of client requests.

Once a servant is instantiated, connect the servant to the POA, so that it can recognize the invocations on it, and pass the invocations to the correct servant. Line 18 performs this task.

Lines 20 through to 21 registers the service through the default naming mechanism. More information about the options available can be found in the ORB Portability Guide.

If this registration is successful, line 23 outputs a `sanity check` message.

Finally, line 25 places the server process into a state where it can begin to accept requests from client processes.

Example 6.4. DemoServer

```
1    import java.io.*;
2    import com.arjuna.orbportability.*;
3
4    public class DemoServer
5    {
6        public static void main (String[] args)
7        {
8            try
9            {
10               ORB myORB = ORB.getInstance("test").initORB(args, null);
11               RootOA myOA = OA.getRootOA(myORB).myORB.initOA();
12
13               ORBManager.setORB(myORB);
14                   ORBManager.setPOA(myOA);
15
16                   DemoImplementation obj = new DemoImplementation();
17
18               myOA.objectIsReady(obj);
19
20               Services serv = new Services(myORB);
21               serv.registerService(myOA.corbaReference(obj), "DemoObjReference", null);
22
23               System.out.println("Object published.");
24
25               myOA.run();
26           }
27           catch (Exception e)
28           {
29               System.err.println(e);
30           }
31       }
32   }
```

After the server compiles, you can use the command line options defined below to start a server process. By specifying the usage of a filter on the command line, you can override settings in the `TransactionService.properties` file.

Note

if you specify the interposition filter, you also imply usage of implicit context propagation.

6.1.1.4. Client implementation

The client, like the server, requires you to first initialize the ORB and the POA. Lines 14 through 18 accomplish these tasks.

After a server process is started, you can obtain the object reference through the default publication mechanism used to publish it in the server. This is done in lines 20 and 21. Initially the reference is an instance of `Object`. However, to invoke a method on the servant object, you need to narrow this instance to an instance of the `DemoInterface` interface. This is shown in line 21.

Once we have a reference to this servant object, we can start a transaction (line 23), perform a unit of work (line 25) and commit the transaction (line 27).

Example 6.5. DemoClient

```
1    import Demo.*;
2    import java.io.*;
3    import com.arjuna.orbportability.*;
4    import com.arjuna.ats.jts.*;
5    import org.omg.CosTransactions.*;
6    import org.omg.*;
7
8    public class DemoClient
9    {
10       public static void main(String[] args)
11       {
12           try
13           {
14               ORB myORB = ORB.getInstance("test").initORB(args, null);
15               RootOA myOA = OA.getRootOA(myORB).myORB.initOA();
16
17               ORBManager.setORB(myORB);
18               ORBManager.setPOA(myOA);
19
20                   Services serv = new Services(myORB);
21               DemoInterface d = (DemoInterface) DemoInterfaceHelper.narrow(serv.getService('
22
23               OTS.get_current().begin();
24
25               d.work();
26
```

```
27              OTS.get_current().commit(true);
28          }
29          catch (Exception e)
30          {
31              System.err.println(e);
32          }
33      }
34  }
```

6.1.1.5. Sequence diagram

The sequence diagram illustrates the method invocations that occur between the client and server. The following aspects are important:

- You do not need to pass the transactional context as a parameter in method `work`, since you are using implicit context propagation.
- Specifying the use of interposition when the client and server processes are started, by using appropriate filters and interceptors, creates an interposed coordinator that the servant process can use, negating any requirement for cross-process invocations. The interposed coordinator is automatically registered with the root coordinator at the client.
- The resource that commits or rolls back modifications made to the transactional object is associated, or registered, with the interposed coordinator.
- The `commit` invocation in the client process calls the root coordinator. The root coordinator calls the interposed coordinator, which in turn calls the `commit_one_phase` method for the resource.

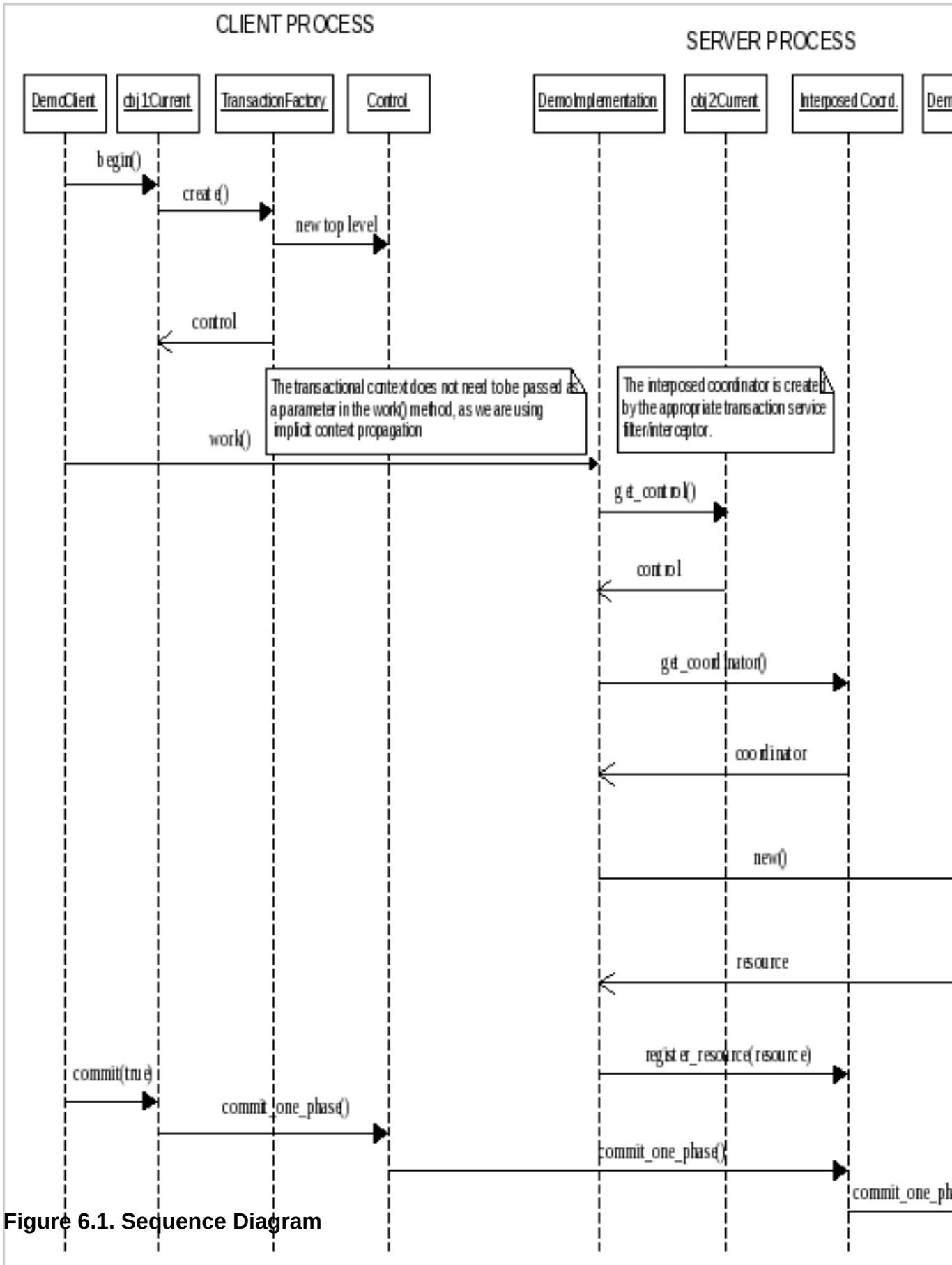

Figure 6.1. Sequence Diagram

6.1.1.6. Interpretation of output

The server process first stringifies the servant instance, and writes the servant IOR to a temporary file. The first line of output is the sanity check that the operation was successful.

In this simplified example, the coordinator object has only a single registered resource. Consequently, it performs a `commit_one_phase` operation on the resource object, instead of performing a `prepare` operation, followed by a `commit` or `rollback`.

The output is identical, regardless of whether implicit context propagation or interposition is used, since interposition is essentially performance aid. Ordinarily, you may need to do a lot of marshaling between a client and server process.

Example 6.6. Server output

```
Object reference written to file
commit_one_phase called
```

6.2. Default settings

These settings are defaults, and you can override them at run-time by using property variables, or in the properties file in the `etc/` directory of the installation.

- Unless a CORBA object is derived from `CosTransactions::TransactionalObject`,you do not need to propagate any context. In order to preserve distribution transparency, JBossTS defaults to always propagating a transaction context when calling remote objects, regardless of whether they are marked as transactional objects. You can override this by setting the `com.arjuna.ats.jts.alwaysPropagateContext` property variable to `NO`.
- If an object is derived from `CosTransactions::TransactionalObject`, and no client context is present when an invocation is made, JBossTS transmits a null context. Subsequent transactions begun by the object are top-level. If a context is required, then set the `com.arjuna.ats.jts.needTranContext` property variable to `YES`, in which case JBossTS raises the `TransactionRequired` exception.
- JBossTS needs a persistent object store, so that it can record information about transactions in the event of failures. If all transactions complete successfully, this object store has no entries. The default location for this must be set using the `ObjectStoreEnvironmentBean.objectStoreDir` variable in the properties file.
- If you use a separate transaction manager for `Current`, its location is obtained from the `CosServices.cfg` file. `CosServices.cfg` is located at runtime by the `OrbPortabilityEnvironmentBean` properties `initialReferencesRoot` and `initialReferencesFile`. The former is a directory, defaulting to the current working directory. The latter is a file name, relative to the directory. The default value is `CosServices.cfg`.

- Checked transactions are not enabled by default. This means that threads other than the transaction creator may terminate the transaction, and no check is made to ensure all outstanding requests have finished prior to transaction termination. To override this, set the `JTSEnvironmentBean.checkedTransactions` property variable to `YES`.
- 

  Note

  As of JBossTS 4.5, transaction timeouts are unified across all transaction components and are controlled by ArjunaCore. The old JTS configuration property com.arjuna.ats.jts.defaultTimeout still remains but is deprecated.

  if a value of `0` is specified for the timeout of a top-level transaction, or no timeout is specified, JBossTS does not impose any timeout on the transaction. To override this default timeout, set the `CoordinatorEnvironmentBean.defaultTimeout` property variable to the required timeout value in seconds.

Failure Recovery

The failure recovery subsystem of JBossTS ensure that results of a transaction are applied consistently to all resources affected by the transaction, even if any of the application processes or the hardware hosting them crash or lose network connectivity. In the case of hardware crashes or network failures, the recovery does not take place until the system or network are restored, but the original application does not need to be restarted. Recovery is handled by the Recovery Manager process. For recover to take place, information about the transaction and the resources involved needs to survive the failure and be accessible afterward. This information is held in the `ActionStore` , which is part of the `ObjectStore` . If the `ObjectStore` is destroyed or modified, recovery may not be possible.

Until the recovery procedures are complete, resources affected by a transaction which was in progress at the time of the failure may be inaccessible. Database resources may report this as as tables or rows held by in-doubt transactions . For TXOJ resources, an attempt to activate the Transactional Object, such as when trying to get a lock, fails.

7.1. Configuring the failure recovery subsystem for your ORB

Although some ORB-specific configuration is necessary to configure the ORB sub-system, the basic settings are ORB-independent. The configuration which applies to JBossTS is in the `RecoveryManager-properties.xml` file and the `orportability-properties.xml` file. Contents of each file are below.

Example 7.1. RecoverManager-properties.xml

```
<entry key="RecoveryEnvironmentBean.recoveryActivatorClassNames">
  com.arjuna.ats.internal.jts.orbspecific.recovery.RecoveryEnablement
</entry>
```

Example 7.2. orportability-properties.xml

```
y="com.arjuna.orbportability.orb.PostInit2">com.arjuna.ats.internal.jts.recovery.RecoveryInit</
entry>
```

These entries cause instances of the named classes to be loaded. The named classes then load the ORB-specific classes needed and perform other initialization. This enables failure recovery for transactions initiated by or involving applications using this property file. The default `RecoveryManager-properties.xml` file and `orportability-properties.xml` with the distribution include these entries.

Important

Failure recovery is NOT supported with the JavaIDL ORB that is part of JDK. Failure recovery is supported for JacOrb only.

To disable recovery, remove or comment out the `RecoveryEnablement` line in the property file.

7.2. JTS specific recovery

7.2.1. XA resource recovery

Recovery of XA resources accessed via JDBC is handled by the `XARecoveryModule` . This module includes both transaction-initiated and resource-initiated recovery.

- Transaction-initiated recovery is possible where the particular transaction branch progressed far enough for a `JTA_ResourceRecord` to be written in the ObjectStore. The record contains the information needed to link the transaction to information known by the rest of JBossTS in the database.
- Resource-initiated recovery is necessary for branches where a failure occurred after the database made a persistent record of the transaction, but before the `JTA_ResourceRecord` was written. Resource-initiated recovery is also necessary for datasources for which it is impossible to hold information in the `JTA_ResourceRecord` that allows the recreation in the RecoveryManager of the `XAConnection` or `XAResource` used in the original application.

Transaction-initiated recovery is automatic. The `XARecoveryModule` finds the `JTA_ResourceRecord` which needs recovery, using the two-pass mechanism described above. It then uses the normal recovery mechanisms to find the status of the transaction the resource was involved in, by running `replay_completion` on the `RecoveryCoordinator` for the transaction branch. Next, it creates or recreates the appropriate `XAResource` and issues `commit` or `rollback` on it as appropriate. The `XAResource` creation uses the same database name, username, password, and other information as the application.

Resource-initiated recovery must be specifically configured, by supplying the `RecoveryManager` with the appropriate information for it to interrogate all the `XADataSources` accessed by any JBossTS application. The access to each `XADataSource` is handled by a class that implements the `com.arjuna.ats.jta.recovery.XAResourceRecovery` interface. Instances of this class are dynamically loaded, as controlled by property `JTAEnvironmentBean.xaResourceRecoveryInstances` .

The `XARecoveryModule` uses the `XAResourceRecovery` implementation to get an `XAResource` to the target datasource. On each invocation of `periodicWorkSecondPass` , the recovery module issues an `XAResource.recover` request. This request returns a list of the transaction identifiers that are known to the datasource and are in an in-doubt state. The list of these in-doubt Xids is compared across multiple passes, using `periodicWorkSecondPass-es` . Any Xid that appears in both lists, and for which no `JTA_ResourceRecord` is found by the intervening

transaction-initiated recovery, is assumed to belong to a transaction involved in a crash before any `JTA_Resource_Record` was written, and a `rollback` is issued for that transaction on the `XAResource` .

This double-scan mechanism is used because it is possible the Xid was obtained from the datasource just as the original application process was about to create the corresponding JTA_ResourceRecord. The interval between the scans should allow time for the record to be written unless the application crashes (and if it does, rollback is the right answer).

An `XAResourceRecovery` implementation class can contain all the information needed to perform recovery to a specific datasource. Alternatively, a single class can handle multiple datasources which have some similar features. The constructor of the implementation class must have an empty parameter list, because it is loaded dynamically. The interface includes an `initialise` method, which passes in further information as a string . The content of the string is taken from the property value that provides the class name. Everything after the first semi-colon is passed as the value of the string. The `XAResourceRecovery` implementation class determines how to use the string.

An `XAResourceRecovery` implementation class, `com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery` , supports resource-initiated recovery for any XADataSource. For this class, the string received in method `initialise` is assumed to contain the number of connections to recover, and the name of the properties file containing the dynamic class name, the database username, the database password and the database connection URL. The following example is for an Oracle 8.1.6 database accessed via the Sequelink 5.1 driver:

```
XAConnectionRecoveryEmpay=com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery;2;OraRecover
```

This implementation is only meant as an example, because it relies upon usernames and passwords appearing in plain text properties files. You can create your own implementations of `XAConnectionRecovery` . See the javadocs and the example com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery .

Example 7.3. XAConnectionRecovery implementation

```
/*
 * Copyright (C) 2000, 2001,
 *
 * Hewlett-Packard,
 * Arjuna Labs,
 * Newcastle upon Tyne,
 * Tyne and Wear,
 * UK.
 *
```

```
 */
package com.arjuna.ats.internal.jdbc.recovery;

import com.arjuna.ats.jdbc.TransactionalDriver;
import com.arjuna.ats.jdbc.common.jdbcPropertyManager;
import com.arjuna.ats.jdbc.logging.jdbcLogger;

import com.arjuna.ats.internal.jdbc.*;
import com.arjuna.ats.jta.recovery.XAConnectionRecovery;
import com.arjuna.ats.arjuna.common.*;
import com.arjuna.common.util.logging.*;

import java.sql.*;
import javax.sql.*;
import javax.transaction.*;
import javax.transaction.xa.*;
import java.util.*;

import java.lang.NumberFormatException;

/**
 * This class implements the XAConnectionRecovery interface for XAResources.
 * The parameter supplied in setParameters can contain arbitrary information
 * necessary to initialise the class once created. In this instance it contains
 * the name of the property file in which the db connection information is
 * specified, as well as the number of connections that this file contains
 * information on (separated by ;).
 *
 * IMPORTANT: this is only an *example* of the sorts of things an
 * XAConnectionRecovery implementor could do. This implementation uses
 * a property file which is assumed to contain sufficient information to
 * recreate connections used during the normal run of an application so that
 * we can perform recovery on them. It is not recommended that information such
 * as user name and password appear in such a raw text format as it opens up
 * a potential security hole.
 *
 * The db parameters specified in the property file are assumed to be
 * in the format:
 *
 * DB_x_DatabaseURL=
 * DB_x_DatabaseUser=
 * DB_x_DatabasePassword=
 * DB_x_DatabaseDynamicClass=
 *
 * DB_JNDI_x_DatabaseURL=
 * DB_JNDI_x_DatabaseUser=
 * DB_JNDI_x_DatabasePassword=
 *
 * where x is the number of the connection information.
```

```
 *
 * @since JTS 2.1.
 */

public class BasicXARecovery implements XAConnectionRecovery
{
    /*
     * Some XAConnectionRecovery implementations will do their startup work
     * here, and then do little or nothing in setDetails. Since this one needs
     * to know dynamic class name, the constructor does nothing.
     */
    public BasicXARecovery () throws SQLException
    {
        numberOfConnections = 1;
        connectionIndex = 0;
        props = null;
    }

    /**
     * The recovery module will have chopped off this class name already.
     * The parameter should specify a property file from which the url,
     * user name, password, etc. can be read.
     */

    public boolean initialise (String parameter) throws SQLException
    {
        int breakPosition = parameter.indexOf(BREAKCHARACTER);
        String fileName = parameter;

        if (breakPosition != -1)
            {
                fileName = parameter.substring(0, breakPosition -1);

                try
                    {
                        numberOfConnections = Integer.parseInt(parameter.substring(breakPositio
                    }
                catch (NumberFormatException e)
                    {
                        //Produce a Warning Message
                        return false;
                    }
            }

        PropertyManager.addPropertiesFile(fileName);

        try
            {
                PropertyManager.loadProperties(true);
```

```
                props = PropertyManager.getProperties();
            }
        catch (Exception e)
            {
                //Produce a Warning Message

                return false;
            }

        return true;
    }

    public synchronized XAConnection getConnection () throws SQLException
    {
        JDBC2RecoveryConnection conn = null;

        if (hasMoreConnections())
            {
                connectionIndex++;

                conn = getStandardConnection();

                if (conn == null)
                    conn = getJNDIConnection();

                if (conn == null)
                    //Produce a Warning message
                    }

        return conn;
    }

    public synchronized boolean hasMoreConnections ()
    {
        if (connectionIndex == numberOfConnections)
            return false;
        else
            return true;
    }

    private final JDBC2RecoveryConnection getStandardConnection () throws SQLException
    {
        String number = new String(""+connectionIndex);
        String url = new String(dbTag+number+urlTag);
        String password = new String(dbTag+number+passwordTag);
        String user = new String(dbTag+number+userTag);
        String dynamicClass = new String(dbTag+number+dynamicClassTag);
        Properties dbProperties = new Properties();
```

```
        String theUser = props.getProperty(user);
        String thePassword = props.getProperty(password);

        if (theUser != null)
            {
                dbProperties.put(ArjunaJDBC2Driver.userName, theUser);
                dbProperties.put(ArjunaJDBC2Driver.password, thePassword);

                String dc = props.getProperty(dynamicClass);

                if (dc != null)
                    dbProperties.put(ArjunaJDBC2Driver.dynamicClass, dc);

                return new JDBC2RecoveryConnection(url, dbProperties);
            }
        else
            return null;
    }

    private final JDBC2RecoveryConnection getJNDIConnection () throws SQLException
    {
        String number = new String(""+connectionIndex);
        String url = new String(dbTag+jndiTag+number+urlTag);
        String password = new String(dbTag+jndiTag+number+passwordTag);
        String user = new String(dbTag+jndiTag+number+userTag);
        Properties dbProperties = new Properties();
        String theUser = props.getProperty(user);
        String thePassword = props.getProperty(password);

        if (theUser != null)
            {
                dbProperties.put(ArjunaJDBC2Driver.userName, theUser);
                dbProperties.put(ArjunaJDBC2Driver.password, thePassword);
                return new JDBC2RecoveryConnection(url, dbProperties);
            }
        else
            return null;
    }
    private int        numberOfConnections;
    private int        connectionIndex;
    private Properties props;
    private static final String dbTag = "DB_";
    private static final String urlTag = "_DatabaseURL";
    private static final String passwordTag = "_DatabasePassword";
    private static final String userTag = "_DatabaseUser";
    private static final String dynamicClassTag = "_DatabaseDynamicClass";
    private static final String jndiTag = "JNDI_";

    /*
```

```
     * Example:
     *
     * DB2_DatabaseURL=jdbc\:arjuna\:sequelink\://qa02\:20001
     * DB2_DatabaseUser=tester2
     * DB2_DatabasePassword=tester
     * DB2_DatabaseDynamicClass=
     *      com.arjuna.ats.internal.jdbc.drivers.sequelink_5_1
     *
     * DB_JNDI_DatabaseURL=jdbc\:arjuna\:jndi
     * DB_JNDI_DatabaseUser=tester1
     * DB_JNDI_DatabasePassword=tester
     * DB_JNDI_DatabaseName=empay
     * DB_JNDI_Host=qa02
     * DB_JNDI_Port=20000
     */

    private static final char BREAKCHARACTER = ';';  // delimiter for parameters
}
```

Multiple recovery domains and resource-initiated recovery

`XAResource.recover` returns the list of all transactions that are in-doubt with in the datasource. If multiple recovery domains are used with a single datasource, resource-initiated recovery sees transactions from other domains. Since it does not have a `JTA_ResourceRecord` available, it rolls back the transaction in the database, if the Xid appears in successive recover calls. To suppress resource-initiated recovery, do not supply an `XAConnectionRecovery` property, or confine it to one recovery domain.

7.2.2. Recovery behavior

Property `OTS_ISSUE_RECOVERY_ROLLBACK` controls whether the `RecoveryManager` explicitly issues a rollback request when `replay_completion` asks for the status of a transaction that is unknown. According to the `presume-abort` mechanism used by OTS and JTS, the transaction can be assumed to have rolled back, and this is the response that is returned to the `Resource` , including a subordinate coordinator, in this case. The `Resource` should then apply that result to the underlying resources. However, it is also legitimate for the superior to issue a rollback, if `OTS_ISSUE_RECOVERY_ROLLBACK` is set to `YES` .

The OTS transaction identification mechanism makes it possible for a transaction coordinator to hold a `Resource` reference that will never be usable. This can occur in two cases:

- The process holding the `Resource` crashes before receiving the commit or rollback request from the coordinator.

- The Resource receives the commit or rollback, and responds. However, the message is lost or the coordinator process has crashed.

In the first case, the RecoveryManager for the Resource ObjectStore eventually reconstructs a new Resource (with a different CORBA object reference (IOR), and issues a replay_completion request containing the new Resource IOR. The RecoveryManager for the coordinator substitutes this in place of the original, useless one, and issues commit to the new reconstructed Resource . The Resource has to have been in a commit state, or there would be no transaction intention list. Until the replay_completion is received, the RecoveryManager tries to send commit to its Resource reference.–This will fail with a CORBA System Exception. Which exception depends on the ORB and other details.

In the second case, the Resource no longer exists. The RecoveryManager at the coordinator will never get through, and will receive System Exceptions forever.

The RecoveryManager cannot distinguish these two cases by any protocol mechanism. There is a perceptible cost in repeatedly attempting to send the commit to an inaccessible Resource . In particular, the timeouts involved will extend the recovery iteration time, and thus potentially leave resources inaccessible for longer.

To avoid this, the RecoveryManager only attempts to send commit to a Resource a limited number of times. After that, it considers the transaction assumed complete . It retains the information about the transaction, by changing the object type in the ActionStore , and if the Resource eventually does wake up and a replay_completion request is received, the RecoveryManager activates the transaction and issues the commit request to the new Resource IOR. The number of times the RecoveryManager attempts to issue commit as part of the periodic recovery is controlled by the property variable COMMITTED_TRANSACTION_RETRY_LIMIT , and defaults to 3 .

7.2.3. Expired entry removal

The operation of the recovery subsystem causes some entries to be made in the ObjectStore that are not removed in normal progress. The RecoveryManager has a facility for scanning for these and removing items that are very old. Scans and removals are performed by implementations of the >com.arjuna.ats.arjuna.recovery.ExpiryScanner . Implementations of this interface are loaded by giving the class names as the value of the property RecoveryEnvironmentBean.expiryScannerClassNames . The RecoveryManager calls the scan method on each loaded ExpiryScanner implementation at an interval determined by the property RecoveryEnvironmentBean.expiryScanInterval . This value is given in hours, and defaults to 12 . A property value of 0 disables any expiry scanning. If the value as supplied is positive, the first scan is performed when RecoveryManager starts. If the value is negative, the first scan is delayed until after the first interval, using the absolute value.

There are two kinds of item that are scanned for expiry:

Contact items	One contact item is created by every application process that uses JBossTS. They contain the information that the RecoveryManager uses to determine if the process that initiated the transaction is still alive, and what the

	transaction status is. The expiry time for these is set by the property `RecoveryEnvironmentBean.transactionStatusManagerExpiryTime` , which is expressed in hours. The default is `12` , and `0` suppresses the expiration. This is the interval after which a process that cannot be contacted is considered to be dead. It should be long enough to avoid accidentally removing valid entries due to short-lived transient errors such as network downtime.
Assumed complete transactions	The expiry time is counted from when the transactions were assumed to be complete. A `replay_completion` request resets the clock. The risk with removing assumed complete transactions it that a prolonged communication outage means that a remote `Resource` cannot connect to the `RecoveryManager` for the parent transaction. If the assumed complete transaction entry is expired before the communications are recovered, the eventual `replay_completion` will find no information and the `Resource` will be rolled back, although the transaction committed. Consequently, the expiry time for assumed complete transactions should be set to a value that exceeds any anticipated network outage. The parameter is `ASSUMED_COMPLETE_EXPIRY_TIME` . It is expressed in hours, with `240` being the default, and `0` meaning never to expire.

Example 7.4. ExpiryScanner properties

```
<entry key="RecoveryEnvironmentBean.expiryScannerClassNames">
  com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner
  com.arjuna.ats.internal.jts.recovery.contact.ExpiredContactScanner
  com.arjuna.ats.internal.jts.recovery.transactions.ExpiredToplevelScanner
  com.arjuna.ats.internal.jts.recovery.transactions.ExpiredServerScanner
</entry>
```

There are two `ExpiryScannner` s for the assumed complete transactions, because there are different types in the ActionStore.

7.2.4. Recovery domains

A key part of the recovery subsystem is that the `RecoveryManager` hosts the OTS `RecoveryCoordinator` objects that handle recovery for transactions initiated in application processes. Information passes between the application process and the `RecoveryManager` in one of three ways:

- `RecoveryCoordinator` object references (IORs) are created in the application process. They contain information identifying the transaction in the object key. They pass the object key to the `Resource` objects, and the `RecoveryManager` receives it.
- The application process and the `RecoveryManager` access the same `jbossts-properties.xml` , and therefore use the same `ObjectStore` .

- The `RecoveryCoordinator` invokes CORBA directly in the application, using information in the contact items. Contact items are kept in the `ObjectStore` .

Multiple recovery domains may useful if you are doing a migration, and separate `ObjectStores` are useful. However, multiple RecoveryManagers can cause problems with XA datasources if resource-initiated recovery is active on any of them.

7.3. Transaction status and `replay_comparison`

When a transaction successfully commits, the transaction log is removed from the system. The log is no longer required, since all registered Resources have responded successfully to the two-phase commit sequence. However, if a `Resource` calls `replay_completion` on the `RecoveryCoordinator` after the transaction it represents commits, the status returned is `StatusRolledback` . The transaction system does not keep a record of committed transactions, and assumes that in the absence of a transaction log, the transaction must have rolled back. This is in line with the `presumed abort protocol` used by the OTS.

JTA and JTS

8.1. Distributed JTA

This guide describes how to use the JTA interfaces for purely local transactions. This is a high-performance implementation, but you can only use it to execute transactions within the same process. If you need support for distributed transactions, the JTA needs to use the JTS. Another advantage of this approach is interoperability with other JTS-compliant transaction systems.

Note

If you use the JTS and JTA interfaces to manage the same transactions, the JTA needs to be configured to be aware of the JTS. Otherwise, local transactions will be unaware of their JTS counterparts.

You need to do this configuration manually, because some applications may be using JBossTS in a purely local manner, or may need to differentiate between transactions managed by JTS and JTA.

Procedure 8.1. Making the JTA interfaces JTS-aware

1. Set `JTAEnvironmentBean.jtaTMImplementation` to `com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple`.
2. Set `JTAEnvironmentBean.jtaUTImplementation` to `com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple`.

Tools

9.1. Introduction

This chapter includes descriptions of JTS specific tools.

9.2. RMIC Extensions

The RMIC extensions allow stubs and tie classes to be generated for transactional RMI-IIOP objects. A transactional object is one which wishes to receive transactional context when one of its methods is invoked. Without transactional object support an RMI-IIOP object won't have transactional context propagated to it when its methods are invoked.

The tool works in two ways: i) via the command line, ii) via ANTs RMIC compiler task. Examples of how to use the tool via these methods are covered in the following sections.

9.2.1. Command Line Usage

As this tool delegates compilation to the Sun RMIC tool it accepts the same command line parameters. So for more details please see it's documentation for details (http://java.sun.com/j2se/1.4.2/docs/tooldocs/tools.html#rmi). The following is an example of how this can be used:

```
java com.arjuna.common.tools.rmictool.RMICTool <parameters>
```

9.2.2. ANT Usage

The RMICTool also acts as a plug-in for the ANT RMIC task. To use the RMICTool simply specify the fully qualified classname as the compiler attribute, e.g.

Example 9.1. Example ANT `rmic` declaration

```
<rmic classname="RMIObject" base="build-
dir"
  compiler="com.arjuna.common.tools.rmictool.RMICTool"
  verify="true" iiop="true" iiopopts="-poa" classpathref="build.classpath" />
```

The RMICTool JAR file must either be specified in your system classpath or it should be copied into the lib directory of your ANT distribution for it to be found.

ORB-specific configuration

10.1. JacORB

Take care to use only the patched version of JacORB shipped with JBossTS. Correct functioning of the transaction system, particularly with regard to crash recovery, is unlikely to work with an unpatched JacORB. For each deployment of JacORB, ensure that the `jacorb.implname` in the `jacorb.properties` file is unique.

Appendix A. IDL definitions

Because of differences between ORBs, and errors in certain ORBs, the idl available with JBossTS may differ from that shown below. You should always inspect the idl files prior to implementation to determine what, if any, differences exist.

Example A.1. CosTransactions.idl

```
#ifndef COSTRANSACTIONS_IDL_
#define COSTRANSACTIONS_IDL_
module CosTransactions
{
    enum Status { StatusActive, StatusMarkedRollback, StatusPrepared,
        StatusCommitted, StatusRolledback, StatusUnknown,
           StatusPreparing, StatusCommitting, StatusRollingBack,
        StatusNoTransaction };

    enum Vote { VoteCommit, VoteRollback, VoteReadOnly };
    // Standard exceptions - some Orb supports them
exception TransactionRequired {};
exception TransactionRolledBack {};
exception InvalidTransaction {};
    // Heuristic exceptions
exception HeuristicRollback {};
       exception HeuristicCommit {};
       exception HeuristicMixed {};
       exception HeuristicHazard {};
    // Exception from ORB
exception WrongTransaction {};
    // Other transaction related exceptions
exception SubtransactionsUnavailable {};
exception NotSubtransaction {};
exception Inactive {};
exception NotPrepared {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
exception SynchronizationUnavailable {};
    // Forward references for later interfaces
interface Control;
interface Terminator;
interface Coordinator;
interface Resource;
interface RecoveryCoordinator;
interface SubtransactionAwareResource;
interface TransactionFactory;
interface TransactionalObject;
```

```
interface Current;
interface Synchronization;
    // Formally part of CosTSInteroperation
struct otid_t
{
    long formatID;
    long bequal_length;
    sequence <octet> tid;
};
struct TransIdentity
       {
           Coordinator coord;
       Terminator term;
           otid_t otid;
       };
struct PropagationContext
      {
       unsigned long timeout;
          TransIdentity currentTransaction;
       sequence <TransIdentity> parents;
       any implementation_specific_data;
      };
      interface Current : CORBA::Current
      {
       void begin () raises (SubtransactionsUnavailable);
       void commit (in boolean report_heuristics) raises (NoTransaction, HeuristicMixed, Heuris
       void rollback () raises (NoTransaction);
       void rollback_only () raises (NoTransaction);

       Status get_status ();
       string get_transaction_name ();
       void set_timeout (in unsigned long seconds);

       Control get_control ();
       Control suspend ();
       void resume (in Control which) raises (InvalidControl);
      };
interface TransactionFactory
      {
       Control create (in unsigned long time_out);
           Control recreate (in PropagationContext ctx);
      };
interface Control
      {
       Terminator get_terminator () raises (Unavailable);
       Coordinator get_coordinator () raises (Unavailable);
      };
interface Terminator
      {
```

```
        void commit (in boolean report_heuristics) raises (HeuristicMixed, HeuristicHazard, Tran
        void rollback ();
       };
       interface Coordinator
       {
        Status get_status ();
        Status get_parent_status ();
        Status get_top_level_status ();

        boolean is_same_transaction (in Coordinator tc);
        boolean is_related_transaction (in Coordinator tc);
        boolean is_ancestor_transaction (in Coordinator tc);
        boolean is_descendant_transaction (in Coordinator tc);
        boolean is_top_level_transaction ();

            unsigned long hash_transaction ();
        unsigned long hash_top_level_tran ();

        RecoveryCoordinator register_resource (in Resource r) raises (Inactive);
        void register_synchronization (in Synchronization sync) raises (Inactive, Synchronizatio
        void register_subtran_aware (in SubtransactionAwareResource r) raises (Inactive, NotSubt

        void rollback_only () raises (Inactive);

        string get_transaction_name ();

        Control create_subtransaction () raises (SubtransactionsUnavailable, Inactive);

        PropagationContext get_txcontext () raises (Unavailable);
       };
       interface RecoveryCoordinator
       {
        Status replay_completion (in Resource r) raises (NotPrepared);
       };
interface Resource
        {
        Vote prepare () raises (HeuristicMixed, HeuristicHazard);
      void rollback () raises (HeuristicCommit, HeuristicMixed, HeuristicHazard);
        void commit () raises (NotPrepared, HeuristicRollback, HeuristicMixed, HeuristicHazard);
        void commit_one_phase () raises (HeuristicHazard);
        void forget ();
       };
interface SubtransactionAwareResource : Resource
       {
        void commit_subtransaction (in Coordinator parent);
        void rollback_subtransaction ();
       };
interface TransactionalObject
       {
```

```
      };
interface Synchronization : TransactionalObject
      {
       void before_completion ();
       void after_completion (in Status s);
      };
};
#endif
```

Example A.2. ArjunaOTS.IDL

```
#ifndef ARJUNAOTS_IDL_
#define ARJUNAOTS_IDL_

#include <idl/CosTransactions.idl>
module ArjunaOTS
{
    exception ActiveTransaction {};
    exception BadControl {};
    exception Destroyed {};
    exception ActiveThreads {};
    exception InterpositionFailed {};

    interface UidCoordinator : CosTransactions::Coordinator
    {
   readonly attribute string uid;
   readonly attribute string topLevelUid;
    };
    interface ActionControl : CosTransactions::Control
    {
        CosTransactions::Control getParentControl ()
                                          raises (CosTransactions::Unavailable,
                                           CosTransactions::NotSubtransaction);
        void destroy () raises (ActiveTransaction, ActiveThreads, BadControl,
                                Destroyed);
    };

    interface ArjunaSubtranAwareResource :
                                    CosTransactions::SubtransactionAwareResource
    {
   CosTransactions::Vote prepare_subtransaction ();
    };
    interface ArjunaTransaction : UidCoordinator, CosTransactions::Terminator
    {
    };

    interface OTSAbstractRecord : ArjunaSubtranAwareResource
```

```
    {
        readonly attribute long typeId;
        readonly attribute string uid;

        boolean propagateOnAbort ();
        boolean propagateOnCommit ();

        boolean saveRecord ();

        void merge (in OTSAbstractRecord record);
        void alter (in OTSAbstractRecord record);

        boolean shouldAdd (in OTSAbstractRecord record);
        boolean shouldAlter (in OTSAbstractRecord record);
        boolean shouldMerge (in OTSAbstractRecord record);
        boolean shouldReplace (in OTSAbstractRecord record);
    };
};
```

References

[OMG95] Copyright © 1995 OMG. OMG. *CORBAservices: Common Object Services Specification*. [OMG Document Number 95-3-31]

[JTA99] Copyright © 1999 Sun Microsystems. Sun Microsystems. *Java Transaction API*.

Appendix B. Revision History

Revision History

Revision 1	Wed Nov 17 2010	MistyStanley-Jones<misty@redhat.com>

Initial conversion to Docbook

Revision 2	Thu Apr 14 2011	TomJenkinson<tom.jenkinson@redhat.com>

Moved some content to main developers guide and added tools information