

JBoss Transactions 4.2.2

JTS Programmers Guide

JBTS-PG-11/2/06



Legal Notices

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

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

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

Copyright

JBoss, Home of Professional Open Source Copyright 2006, JBoss Inc., and individual contributors as indicated by the @authors tag. All rights reserved.

See the copyright.txt in the distribution for a full listing of individual contributors. This copyrighted material is made available to anyone wishing to use, modify, copy, or redistribute it subject to the terms and conditions of the GNU General Public License, v. 2.0. This program is distributed in the hope that it will be useful, but WITHOUT A WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

See the GNU General Public License for more details. You should have received a copy of the GNU General Public License, v. 2.0 along with this distribution; if not, write to the Free Software Foundation, Inc., 51 Franklin Street, Fifth Floor, Boston, * MA 02110-1301, USA.

Software Version

JBoss Transactions 4.2.2

Restricted Rights Legend

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

© Copyright 2006 JBoss Inc.

Contents

About This Guide.....5

What This Guide Contains	5
Audience.....	5
Prerequisites	5
Organization	5
Documentation Conventions	6
Additional Documentation	7
Contacting Us.....	7

An overview of transaction processing.....9

What is a transaction?.....	9
Commit protocol	9
Transactional proxies	10
Nested transactions	10
The Object Transaction Service	11

JBossTS basics.....13

Basics of JBossTS.....	13
Raw OTS	13
Enhanced OTS functionality	14
Advanced application programmer interface	14
<i>JBossTS</i> and the OTS specification	16
Thread class.....	17
ORB portability issues.....	17

An introduction to the OTS.....18

Introduction	18
What is the OTS?	18
Application programming models	19
Interfaces	21
The transaction factory	21
OTS configuration file.....	22
Name Service	23
resolve_initial_references.....	23
ORB specific location mechanism.....	23
Overriding the default location mechanism ..	23
Transaction timeouts.....	23
Transaction contexts	23
Nested transactions	26

Transaction propagation	27
Examples	28
Transaction Controls.....	29
JBossTS specifics.....	29
The Terminator interface	30
JBossTS specifics.....	30
The Coordinator interface.....	31
JBossTS specifics.....	33
Heuristics	33
Current	33
JBossTS specifics.....	36
Statistics gathering	36
Resource	37
SubtransactionAwareResource.....	39
JBossTS specifics.....	42
The Synchronization interface	43
JBossTS specifics.....	44
Transactions and registered resources	45
TransactionalObject interface.....	49
JBossTS specifics.....	50
Interposition.....	50
Asynchronously committing a transaction ...	51
The RecoveryCoordinator	51
Checked transaction behaviour	52
JBossTS specifics.....	54
Summary of JBossTS implementation decisions.....	55

Constructing an OTS application.....56

Important notes for JBossTS	56
Initialisation.....	56
Implicit context propagation and interposition.....	56
Writing applications using the raw OTS interfaces	56
Transaction context management.....	57
A transaction originator: indirect and implicit	57
Transaction originator: direct and explicit ...	58
Implementing a transactional client	58
Implementing a recoverable server	59
Transactional object	59

Resource object	59	Starting the Transaction Service tools	99
Reliable servers	59	Using the Performance Tool.....	101
Example of a recoverable server	60	Using the JMX Browser	102
Example of a transactional object	61	Using Attributes and Operations	103
Failure models	61	Using the Object Store Browser.....	106
Transaction originator.....	61	Object State Viewers (OSV)	107
Transactional server	62	RMIC Extensions	112
Summary.....	63	Command Line Usage	112
		ANT Usage	113
JBossTS interfaces for extending the OTS ...	64	ORB specific configurations.....	114
Introducing	64	Orbix 2000.....	114
Nested transactions	65	Configuring JBossTS.....	116
Extended resources	65	Options.....	116
AtomicTransaction.....	67	IDL Definitions.....	117
Context propagation issues.....	68	Introduction	117
Example.....	70	CosTransactions.idl.....	117
Introduction	70	ArjunaOTS.idl	120
The basic example.....	70	References	122
Resource	71	References	122
Transactional implementation	72	Index	123
Server implementation.....	73		
Client implementation	74		
Sequence diagram	75		
Interpretation of output.....	76		
Default settings.....	77		
Failure recovery	79		
Introduction	79		
Configuring the failure recovery subsystem for your ORB	79		
The Recovery Manager	80		
Important Note	80		
Configuring the Recovery Manager	80		
XA resource recovery	88		
Recovery behaviour	94		
Expired entry removal	95		
Recovery Domains.....	96		
Transaction statuses and replay_completion.	97		
JTA and the JTS	98		
Distributed JTA.....	98		
Tools.....	99		
Introduction	99		

About This Guide

What This Guide Contains

The JTS Programmers Guide contains information on how to use JBoss Transactions 4.2.2. This document provides a detailed look at the design and operation of. It describes the architecture and the interaction of components and within this architecture.

Audience

Although this guide is specifically intended for service developers using JBoss Transactions 4.2.2, it will be useful to anyone who would like to gain an understanding of the transactions and how they function.

Prerequisites

This guide assumes a basic familiarity with Java™ service development and object-oriented programming. A fundamental level of understanding in the following areas will also be useful:

- A general understanding of the APIs, components, and objects that are present in Java applications.
- A general understanding of the Windows and UNIX operating systems.

Organization

This guide contains the following chapters:

- **Chapter 1, An Overview of transaction processing:** gives an brief overview of the transaction processing.
- **Chapter 2, JBossTS basics:** presents JBossTS and describes its features in terms on compliance to JTS/OTS specifications and enhancements it provides in regards to the OTS specification.
- **Chapter 3, An introduction to the OTS:** describes OTS and the programming models from the User point of view. The way JBossTS offers these programming models and JBossTS enhancements are described.

- **Chapter 4, Constructing an OTS application:** describes how to build an OTS application using JBossTS.
- **Chapter 5, JBossTS interfaces for extending the OTS:** contains a description of the use of JBossTS classes that provide extensions to the OTS interfaces.
- **Chapter 6, Example:** illustrates a detailed client/server example.
- **Chapter 7, Failure Recovery:** describes how to configure JBossTS to manage Failure recovery.
- **Chapter 8, JTA and the JTS:** describes how to configure JTA to be aware of JTS.
- **Chapter 9, Tools:** explains how to start and use the tools framework and what tools are available.
- **Chapter 10, ORB specific configurations:** describes how to configure specific ORBs.
- **Chapter 11, Configuring JBossTS:** shows configurations features of JBossTS.

Documentation Conventions

The following conventions are used in this guide:

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

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss Transactions 4.2.2 documentation set:

- *JBoss Transactions 4.2.2 Release Notes*: Provides late-breaking information about JBoss Transactions 4.2.2.
- *JBoss Transactions 4.2.2 Installation Guide*: This guide provides instructions for installing JBoss Transactions 4.2.2.
- *JBoss Transactions 4.2.2 Administration Guide*: Provides guidance for writing applications.
- *JBoss Transactions 4.2.2 Quick Start Guide*: Getting started quickly with the system.
- *JBoss Transactions API Programmer's Guide*: Provides guidance when using the JTA for building transactional applications.
- *TxCore Failure Recovery Guide*: Describes the failure recovery aspects of JBossTS.
- *TxCore Programmer's Guide*: Describes how to write transactional applications using the non-distributed transaction engine at the heart of JBossTS.

Contacting Us

Questions or comments about JBoss Transactions 4.2.2 should be directed to our support team. Send email to support@arjuna.com.

An overview of transaction processing

What is a transaction?

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

Atomic transactions (transactions) possess an "all-or-nothing" property, and are a well-known technique for guaranteeing application consistency in the presence of failures. Transactions possess the following ACID properties:

- *Atomicity*: The transaction completes successfully (commits) or if it fails (aborts) all of its effects are undone (rolled back).
- *Consistency*: Transactions produce consistent results and preserve application specific invariants.
- *Isolation*: Intermediate states produced while a transaction is executing are not visible to others. Furthermore transactions appear to execute serially, even if they are actually executed concurrently.
- *Durability*: The effects of a committed transaction are never lost (except by a catastrophic failure).

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

Commit protocol

A two-phase commit protocol is required to guarantee that all of the action participants either commit or abort any changes made. Figure 1 illustrates the main aspects of the commit protocol: during phase 1, the action coordinator, C, attempts to communicate with all of the action participants, A and B, to determine whether they will commit or abort. An abort reply from any participant acts as a veto, causing the entire action to abort. Based upon these (lack

of) responses, the coordinator arrives at the decision of whether to commit or abort the action. If the action will commit, the coordinator records this decision on stable storage, and the protocol enters phase 2, where the coordinator forces the participants to carry out the decision. The coordinator also informs the participants if the action aborts.

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

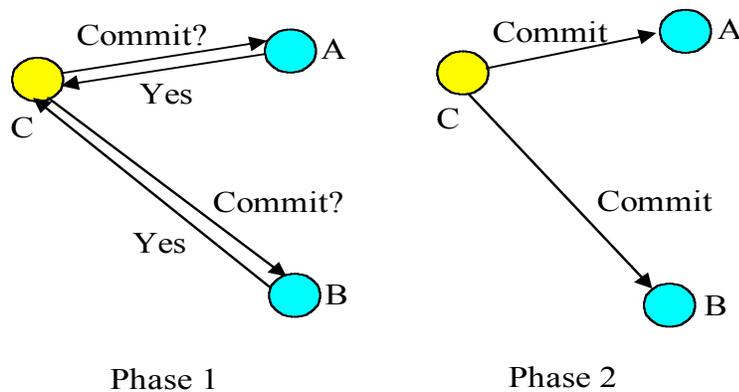


Figure 1: Two-phase commit protocol.

Transactional proxies

The action coordinator maintains a *transaction context* where resources taking part in the action are required to be registered. Such a resource must obey the transaction commit protocol guaranteeing ACID properties; typically this means that the resource will provide specific operations which the action can invoke during the commit/abort protocol. However, it may not be possible to make all resources transactional in this way, e.g., legacy code which cannot be modified. To use these resources within an action it is often possible to provide *transactional proxies*: 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. This requires that the proxy participate within the commit/abort protocol. 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.

Nested transactions

Given a system that provides transactions for certain operations, it is sometimes necessary to 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 the transactions from which

it is composed. The transactions which are contained within the resulting transaction are said to be *nested* (or *subtransactions*), and the resulting transaction is referred to as the *enclosing* transaction. The enclosing transaction is sometimes referred to as the *parent* of a nested (or *child*) transaction. A hierarchical transaction structure can thus result, with the root of the hierarchy being referred to as *the top-level transaction*.

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 transaction(s), i.e., the effects will be recovered if the enclosing transaction aborts, even if the nested transaction has committed.

Subtransactions are a useful mechanism for two reasons:

- *fault-isolation*: if subtransaction rolls back (e.g., because an object it was using fails) then this does not require the enclosing transaction to rollback, thus undoing all of the work performed so far.
- *modularity*: if there is already a transaction associated with a call when a new transaction is begun, then the transaction will be nested within it. Therefore, a programmer who knows that an object require transactions can use them within the object: if the object's methods are invoked without a client transaction, then the object's transactions will simply be top-level; otherwise, they will be nested within the scope of the client's transactions. Likewise, a client need not know that the object is transactional, and can begin its own transaction.

The Object Transaction Service

The CORBA architecture, as defined by the OMG, is a standard derived by an industrial consortium 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), which enables objects to transparently make 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 considered to be necessary for the construction of any distributed application. Of particular relevance to this manual is the Object Transaction Service (OTS).
- Common Facilities, are other useful services that applications may need, but which are not considered to be fundamental such as desktop management and help facilities.

The CORBA architecture is structured to allow both its implementation in, and the 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 supports explicitly reveals such details. As defined in the OMG CORBA Services documentation, object services are a collection of services (interfaces and objects) that support the basic functions for using and implementing objects. Such services are necessary

in the construction of any distributed application and are always independent of an application domain. The document specifies several core services including naming, event management, persistence, concurrency control and transactions.

The OTS specification allows transactions to be nested. However, an implementation need not provide this functionality. Appropriate exceptions are raised if an attempt is made to use nested transactions in this case. *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 such that all objects commit or abort their changes together. However, the OTS does not require all objects to have transactional behaviour. Instead objects can choose not to support transactional operations at all, or to support it for some requests but not others. Transaction information may be propagated between client and server explicitly, or implicitly, giving the programmer finer-grained control over an objects transactionality. Objects supporting (partial) transactional behaviour must have interfaces derived from the `TransactionalObject` interface.

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 by 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 need not necessarily be a recoverable object if its state is actually implemented using other recoverable objects. A simple transactional object need not take part in the commit protocol used to determine the outcome of the transaction since it does not maintain any state itself, having delegated that responsibility to other recoverable objects which will take part in the commit process.

The OTS is simply a *protocol engine* that guarantees that transactional behaviour is obeyed but does not directly support all of the transaction properties given above. As such it requires other co-operating services that implement the required functionality, including:

- *Persistence/Recovery Service*. Required to support the atomicity and durability properties.
- *Concurrency Control Service*. Required to support the isolation properties.

The application programmer is responsible for using appropriate services to ensure that transactional objects have the necessary ACID properties.

Chapter 2

JBossTS basics

Basics of JBossTS

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 will be portable across OTS implementations.

In terms of the OTS specification, *JBossTS* provides:

- 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 1.0 and 2.0 support.
- Full JTA 1.0.1 support.

There are effectively three different levels at which a programmer can approach using *JBossTS*. These will be briefly described in the following sections, and in more detail in subsequent chapters.

Note: because of differences in ORB implementations, *JBossTS* has been written with a separate ORB Portability library which hides these differences; many of the examples used throughout this manual have also been written using this library, and it is therefore recommended that the *ORB Portability Manual* is read first.

Raw OTS

The OTS is actually only a protocol engine for driving registered resources through a two-phase commit protocol. Application programmers are responsible for building and registering

the `Resource` objects which take care of persistence and concurrency control to ensure ACID properties for transactional application objects. The programmer must ensure that `Resources` are registered at appropriate times, and that a given `Resource` is only registered within a single transaction. Therefore, programming at the raw OTS level is extremely basic: the programmer is responsible for many things, including managing persistence and concurrency control on behalf of every transactional object.

Enhanced OTS functionality

The OTS implementation of nested transactions is extremely limited, and can lead to the generation of heuristic-like results: a subtransaction coordinator discovers part way through committing that some resources cannot commit; however, it cannot tell the committed resources to abort. *JBossTS* allows nested transactions to execute a full two-phase commit protocol, thus removing the possibility that some resources will have been committed whereas others will have been rolled back.

When resources are registered with a transaction the programmer has no control over the order in which these resources will be invoked during the commit/abort protocol, or whether previously registered resources should be replaced with newly registered resources, for example, then resources registered with a subtransaction are merged with its parent. *JBossTS* provides an additional `Resource` subtype which gives programmers this control.

Advanced application programmer interface

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. The interfaces defined within the OTS specification are too low-level for most application programmers. Therefore, *JBossTS* comes with *Transactional Objects for Java*, 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, allowing the programmer to concentrate on application development, rather than transaction management.

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.

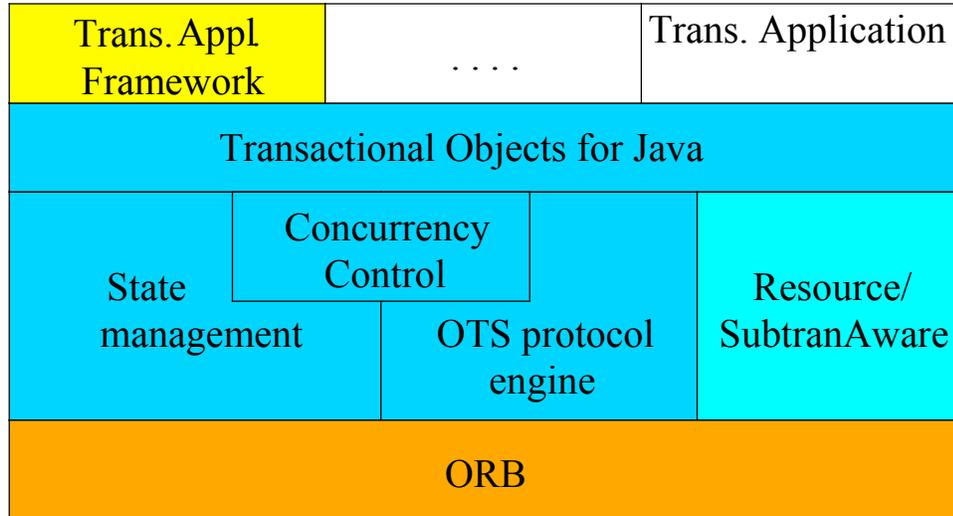


Figure 2: JBossTS structure.

JBossTS exploits object-oriented techniques to present programmers with a toolkit of Java classes from which application classes can inherit to obtain desired properties, such as persistence and concurrency control. These classes form a hierarchy, part of which is shown below.

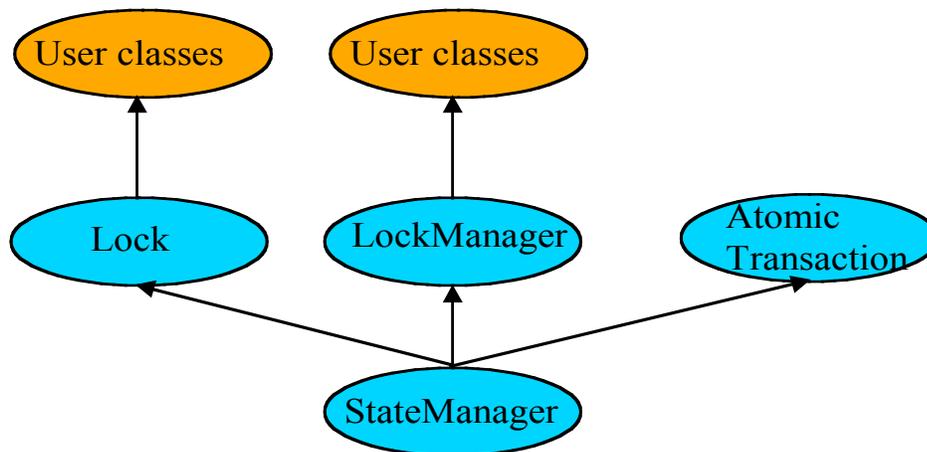


Figure 3: JBossTS class hierarchy.

Apart from specifying the scopes of transactions, and setting appropriate locks within objects, the application programmer does not have any other responsibilities: *JBossTS* guarantees that transactional objects will be registered with, and be driven by, the appropriate transactions, and crash recovery mechanisms are invoked automatically in the event of failures. Using these interfaces, programmers need not worry about either creating or registering `Resource` objects and calling persistence and concurrency control services. *JBossTS* guarantees that appropriate resources will be registered with, and driven by, the transaction. If a transaction is nested, resources will also be 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. In meeting this goal, three system properties were considered highly important:

- *Integration of Mechanisms*: A fault-tolerant distributed system requires a variety of system functions for naming, locating and invoking operations upon objects and also for concurrency control, error detection and recovery from failures. These mechanisms must be integrated such that their use by a programmer is easy and natural.
- *Flexibility*: These mechanisms must be flexible, permitting application specific enhancements, such as type-specific concurrency and recovery control, to be easily produced from existing defaults.
- *Portability*: It should be possible to run *JBossTS* on any ORB.

The system 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.

***JBossTS* and the OTS specification**

The OTS specification is written to allow its implementation in a flexible manner, in order 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 these possible implementations. This section will briefly describe the default behaviour which *JBossTS* provides for certain options. More information can be obtained from relevant sections in the manual.

OTS specification	<i>JBossTS</i> default implementation
If the transaction service chooses to restrict the availability of the transaction context, then it should raise the Unavailable exception.	<i>JBossTS</i> does not restrict the availability of the transaction context
An implementation of the transaction service need not initialise the transaction context for every request.	<i>JBossTS</i> only initialised the transaction context if the interface supported by the target object is derived from 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.	<i>JBossTS</i> 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.	<i>JBossTS</i> allows the termination of a transaction by any client that uses the Terminator interface. In addition, <i>JBossTS</i> does not impose restrictions when clients use the Current interface.
A TransactionFactory is located using the FactoryFinder interface of the life-cycle service.	<i>JBossTS</i> provides multiple ways in which the TransactionFactory can be located.
A transaction service implementation may use the Event Service to report heuristic decisions.	<i>JBossTS</i> does not use the Event Service to report heuristic decisions.
An implementation of the transaction service does not need to support nested transactions.	<i>JBossTS</i> supports nested transactions. To override this, see Section 0.
Synchronization objects are required to be called whenever the transaction commits.	<i>JBossTS</i> allows Synchronizations to be called however the transaction terminates.

A transaction service implementation need not support interposition.

JBossTS supports various types of interposition.

Table 2: JBossTS defaults.

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 (although a thread can only be active within a single transaction at a time). By default, if a thread is created within the scope of a transaction (i.e., the creating thread has a transaction context associated with it), the new thread will not be associated with the transaction. If the thread is to be associated with the transaction then use the `resume` method of either the `AtomicTransaction` class or `Current`.

However, if it is required that newly created threads automatically inherit the transaction context of their parent, then they should be derived from the `OTS_Thread` class:

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

    protected OTS_Thread ();
};
```

The programmer must call the `run` method of `OTS_Thread` at the start of the `run` method of the application thread class. Likewise, it is necessary to call `terminate` prior to exiting the body of the application thread's `run` method:

```
public void run ()
{
    super.run();

    // do my work

    super.terminate();
}
```

ORB portability issues

Although the CORBA specification is a standard, it is written in such a way that there are several different ways in which an ORB can be implemented. As such, writing portable client and server code can be difficult. Because *JBossTS* has been ported to most of the widely available ORBs we believe that we have encountered many of the incompatibilities which can exist between them. As such, in order to make *JBossTS* portable between ORBs we have developed a series of *ORB Portability* classes and macros. If an application is written using these classes then it should be more portable between different *ORBs*. These classes are described in the separate *ORB Portability Manual*.

An introduction to the OTS

Introduction

Basic *JBossTS* programming involves using the OTS interfaces provided in the `CosTransactions` module, specified in `CosTransactions.idl`. This chapter is based on the OTS Specification¹. We shall only consider those aspects of the OTS which are relevant to an application programmer wishing to use *JBossTS*, rather than an OTS implementer. Where relevant, each section will describe *JBossTS* implementation decisions and runtime choices available to the application programmer. These choices are also summarised at the end of this chapter. In subsequent chapters we shall illustrate how these interfaces can be used to construct transactional applications.

What is the OTS?

The raw `CosTransactions` interfaces can be found in the `org.omg.CosTransactions` package. The *JBossTS* implementations of these interfaces are located in the `com.arjuna.CosTransactions` package and its sub packages.

Note: In the following discussion it will be shown how many run-time decisions of *JBossTS* can be overridden using Java properties specified at run-time. The property names are mentioned in the `com.arjuna.ats.jts.common.Environment` class. Section **Error! Reference source not found.** described how these property variables can either be assigned each time the application is executed, or can be placed in a special property file which *JBossTS* reads at runtime.

The fundamental architecture of the OTS is captured in Figure 4. Aspects of this architecture will be described in the rest of the chapter.

¹ Available from <http://www.omg.org>.

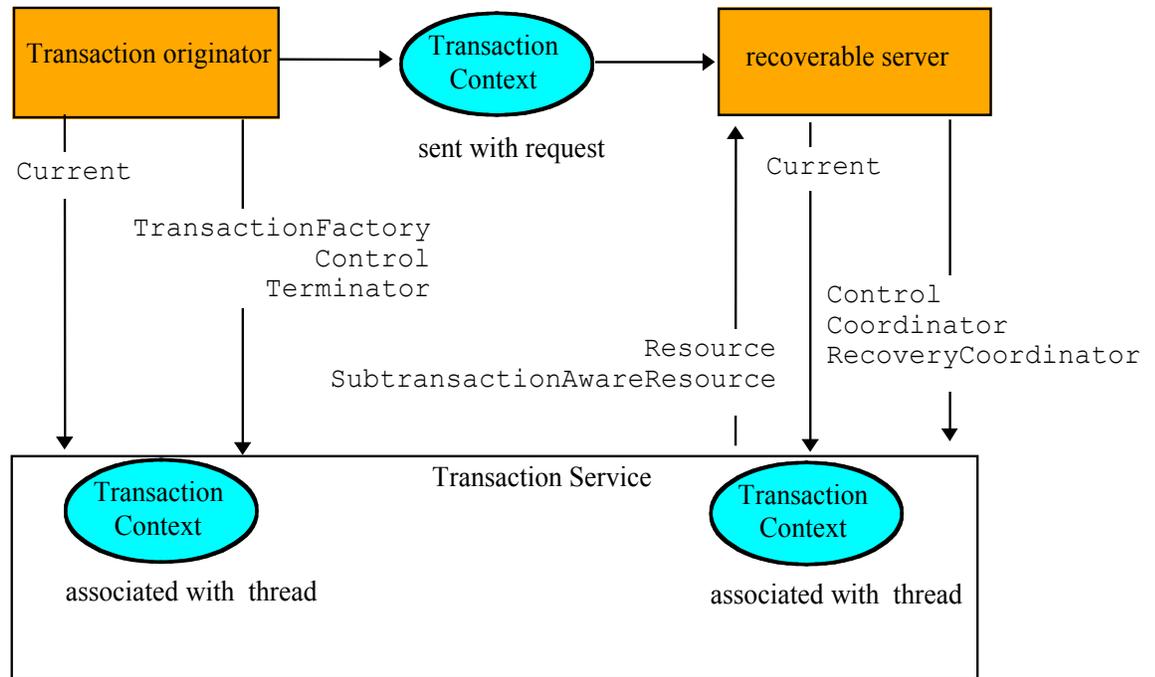


Figure 4: OTS Architecture.

Application programming models

A client application program may use direct or indirect context management to manage a transaction. With indirect context management, an application uses the pseudo object called `Current`, provided by the Transaction Service², to associate the transaction context with the application thread of control. In 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. This should typically be the `PropagationContext` structure.
- *Implicit propagation* means that requests are implicitly associated with the client's transaction; they share the client's transaction context. It is transmitted implicitly 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 would not typically expect to receive any Transaction Service object as an explicit parameter.

² With the release of draft 4 of the specification, `Current` should now be provided by the Orb.

A client may use one or both forms of context management, and may communicate with objects that use either method of transaction propagation. (Details of how to enable implicit propagation were described in Section 0 and Section 0). 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 this should be 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 allows the client to 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. If using indirect (i.e., invoking operations through the Current pseudo object), then the thread's transaction context will be modified automatically by the OTS, e.g., if begin is called then the thread's notion of the current transaction will be modified to the newly created transaction; when that is terminated, the transaction previously associated with the thread (if any) will be restored as the thread's context (assuming subtransactions are supported by the OTS implementation). However, if using direct management, no changes to the threads transaction context are performed by the OTS: the application programmer assumes responsibility for this.

Interfaces

Function	Used by	Direct context management	Indirect ³ context management
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 a transaction	Server	Terminator::rollback_only	rollback_only
Control propagation of transaction to a server	Server	Declaration of method parameter	TransactionalObject interface
Control by client of transaction propagation to a server	All	Request parameters	get_control suspend resume
Become a participant in a transaction	Recoverable Server	Coordinator::register_resource	Not applicable
Miscellaneous	All	Coordinator::get_status Coordinator::get_transaction_name Coordinator::is_same_transaction Coordinator::hash_transaction	get_status get_transaction_name Not applicable Not applicable

Table 3: Use of Transaction Service functionality.

Note: For clarity, subtransaction operations are not shown.

The transaction factory

The `TransactionFactory` interface is provided to allow 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 are direct context management, and as such will not modify the calling thread's transaction context.

The `create` operation creates a new top-level transaction and returns its `Control` object, which can be used to manage or control participation in the new transaction. The parameter to

³ All Indirect context management operations are on the `Current` pseudo-object interface.

`create` is an application specific timeout value, in seconds: if the transaction has not completed before this timeout has elapsed it will be subject to being rolled back. If the parameter is zero, then no application specified timeout is established. (Note, subtransactions do not have a timeout associated with them.) This can be represented in UML as shown below:

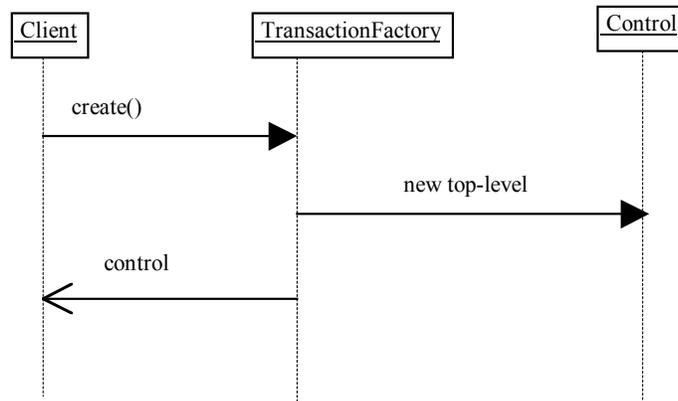


Figure 5: top-level transaction creation (direct mode).

The Transaction Service implementation allows the `TransactionFactory` to be a separate server from the application (e.g., a typical Transaction Monitor) which transaction clients share, and which manages transactions on their behalf. However, the specification also enables the `TransactionFactory` to be implemented by an object within each transactional client. This is the default implementation used by *JBossTS* since it removes the need for a separate service to be available in order for transactional applications to execute.

When running applications which require a separate transaction manager, you must set the `OTS_TRANSACTION_MANAGER` environment variable to have the value `YES`. The system will then locate the transaction manager server in a manner specific to the ORB being used. The server can be located in a number of ways: by being registered with a name server, added to the ORB's initial references, via a JBossTS specific references file, or by the ORB's specific location mechanism (if applicable).

OTS configuration file

Similar to the `resolve_initial_references`, *JBossTS* supports an initial reference file where references for specific services can be stored and used at runtime. The file, `CosServices.cfg`, consists of two columns: the service name (in the case of the OTS server *TransactionService*) and the IOR, separated by a single space. `CosServices.cfg` normally resides in the `etc` directory of the *JBossTS* installation. The OTS server will automatically register itself in this file (creating it if necessary) if this option is being used. Stale information is also automatically removed. The name and location of the file can be overridden using the `INITIAL_REFERENCES_FILE` and `INITIAL_REFERENCES_ROOT` property variables, respectively. For example:

```

INITIAL_REFERENCES_FILE=myFile
INITIAL_REFERENCES_ROOT=c:\\temp
  
```

Name Service

If the ORB you are using supports a name service, and *JBossTS* has been configured to use it, then the transaction manager will automatically be registered with it. There is no further work required

resolve_initial_references

Currently this option is not supported.

ORB specific location mechanism

This configuration option is currently only supported for VisiBroker. At runtime the OTS server supports the following option:

- `-otsname`: when using VisiBroker this is the marker name for the OTS transaction manager object.

Overriding the default location mechanism

It is possible to override the default location mechanism by using the `RESOLVE_SERVICE` property variable. This can have one of the following values:

- `CONFIGURATION_FILE`: the default, this causes the system to use the `CosServices.cfg` file.
- `NAME_SERVICE`: *JBossTS* will attempt to use a name service to locate the transaction factory. If this is not supported, an exception will be thrown.
- `BIND_CONNECT`: *JBossTS* will use the ORB-specific bind mechanism. If this is not supported, an exception will be thrown.

If `RESOLVE_SERVICE` is specified when the transaction factory is run, then the factory will register itself with the specified resolution mechanism.

Transaction timeouts

Refer to the relevant section in the ArjunaCore Programmers Guide.

Transaction contexts

Fundamental to the OTS architecture is the notion of a *transaction context*. Each thread is associated with a context. This association may be null, indicating that the thread has no associated transaction, or it refers to a specific 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 such that when the nested transaction ends the context of the thread can be restored to that in effect before the nested transaction was started. This relationship is shown below in UML, where `Current` is the object most commonly used

by a thread for manipulating its transaction context information (represented by `Control` objects):

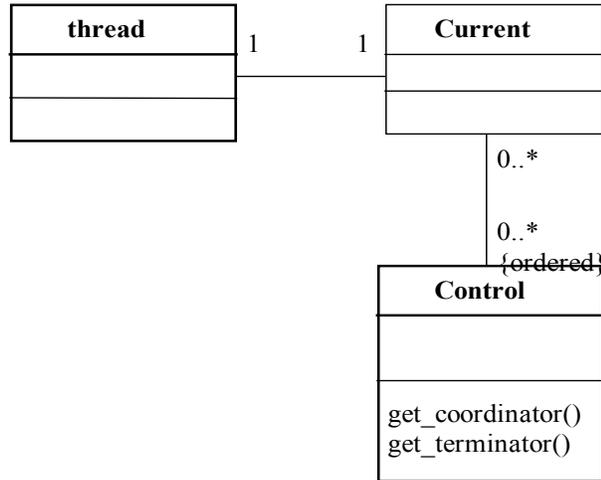


Figure 6: Thread and context relationship.

Management of transaction contexts may be undertaken by an application in either a direct or an indirect manner. 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 two further interfaces to be obtained. These latter interfaces allow an application to end the transaction (via a `Terminator` interface), to become a participant in the transaction, or to start a nested transaction (both via a `Coordinator` interface). These interfaces (shown in detail in Interface 1) are expected to be passed as explicit parameters in operation invocations since transaction creation using these interfaces does not change a thread's current context. If it is necessary to set the current context for a thread to the context represented by the control object returned by the factory the `resume` operation of the `Current` interface must be used.

```

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);
};

```

Interface 1: Direct Context Management Interface.

The relationship between a `Control` and its `Coordinator` and `Terminator` interfaces is shown below:

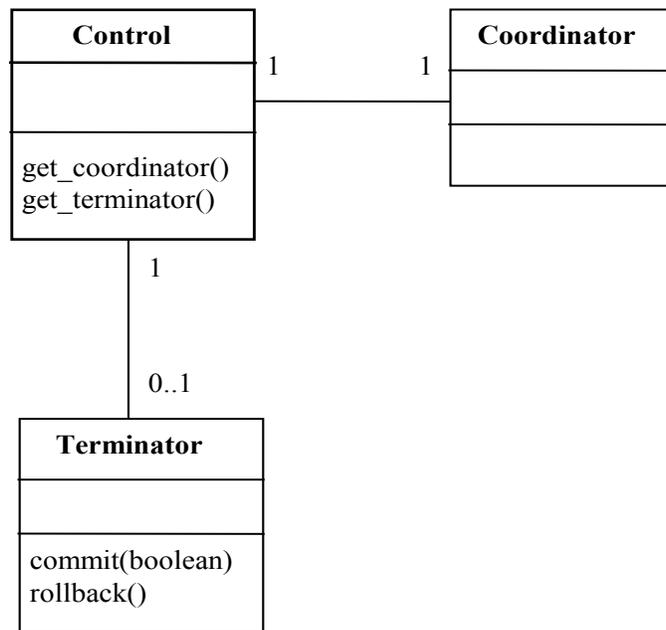


Figure 7: Control relationship.

When a transaction is created by the factory it is possible to specify a timeout value in seconds; if the transaction has not completed within this timeout then it is subject to possible rollback. If the timeout value is zero then no application specific timeout will be set.

In contrast to explicit context management, implicit context management is handled by the `Current` interface (Interface 2) which provides simplified transaction management functionality and automatically creates nested transactions etc. as required. Transactions created using this interface do alter a thread's current transaction context.

```

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);
};

```

Interface 2: Indirect Context Management Interface.

Nested transactions

The provision of nested transaction (subtransactions) by an OTS implementation is optional; *JBossTS supports subtransactions*. Subtransactions are a useful mechanism for two reasons:

- *fault-isolation*: if a subtransaction rolls back (e.g., because an object it was using fails) then this does not require the enclosing transaction to rollback, thus undoing all of the work performed so far.
- *modularity*: if using indirect transaction management (through the `Current` pseudo-object) there is no special syntax for creating subtransactions: a transaction is simply begun, and if there is already a transaction associated with the calling thread then the new transaction will automatically be nested within it. Therefore, a programmer who knows that an object require transactions can use them within the object: if the object's methods are invoked without a client transaction, then the object's transactions will simply be top-level; otherwise, they will be nested within the scope of the client's transactions. Likewise, a client need not know that the object is transactional, and can begin its own transaction.

When nested transactions are provided, the transaction context forms a hierarchy. The outermost transaction of such a hierarchy is typically referred to as the top-level transaction. Unlike top-level transactions, the commits of subtransactions are provisional upon the commit/rollback of the enclosing transactions. Resources acquired within a subtransaction should be inherited (retained) by parent transactions upon the commit of the subtransaction, and (assuming no failures) only released when the top-level transaction completes, i.e., they should be retained for the duration of the top-level transaction. If a subtransaction rolls back, it can release any resources it acquired, and undo any changes to resources it inherited.

Unlike top-level transactions, in the OTS subtransactions behave differently at commit time. Whereas top-level transactions undergo a two-phase commit protocol, nested transactions do not perform any commit protocol: when a program commits a nested transaction then the

transaction is considered committed, and it simply informs any registered resources of its outcome. If a resource cannot commit then it raises an exception, and the OTS implementation is free to ignore this or attempt to rollback the subtransaction. Obviously rolling back a subtransaction may not be possible if some resources have already been told that the transaction has committed.

Transaction propagation

The OTS supports both implicit (system driven) propagation and explicit (application driven) propagation of transactional behaviour. In the implicit case no transactional behaviour is specified in an operation signature and any transaction context associated with the calling thread is automatically sent with each operation invocation. With explicit propagation, applications must define their own mechanism for propagating transactions. This allows:

- 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.

Note that transaction context management and transaction propagation are different things that may be controlled independently of each other. Furthermore, mixing of direct and indirect context management with implicit and explicit transaction propagation is supported. Use of implicit propagation requires co-operation from the ORB, in that the current context associated with the thread must be sent with any operation invocations by a client and extracted by the server prior to actually calling the target operation.

If implicit context propagation is required, then the programmer must ensure that *JBossTS* is correctly initialised prior to objects being created; obviously it is necessary for both client and server to agree to use implicit propagation. Implicit context propagation is only possible on those ORBs which either support filters/interceptors, or the `CostSPortability` interface. Currently this is Orbix 2000. To use implicit transaction propagation, the programmer *must* perform the following:

- Implicit context propagation:
 - set the `OTS_CONTEXT_PROP_MODE` property variable to `CONTEXT`. If using Orbix 2000, see the Orbix 2000 configuration section in the Administrator's Guide.
- Interposition:
 - set the `OTS_CONTEXT_PROP_MODE` property variable to `INTERPOSITION`. If using Orbix 2000, see the Orbix 2000 configuration section in the Administrator's Guide.

If using the *JBossTS* advanced API then interposition is *required*.

Further information on this subject can be found in Chapter 4.

Examples

To aid in comprehension of the above discussions Program 1 illustrates a simple transactional client using both 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
  ...
}
```

Program 1: Simple transactional client (direct/explicit).

In contrast Program 2 shows the same program using indirect context management and implicit propagation. This example is considerably simpler since the application only has to be concerned with starting and then committing or aborting actions

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

Program 2: Simple transactional client (indirect/implicit).

Finally, Program 3 illustrates the potential flexibility of OTS by using both direct and indirect context management in conjunction with explicit and implicit 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();

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

```

Program 3: Mixed transactional client.

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`, which return instances of the `Terminator` and `Coordinator` interfaces, respectively. Both of these methods throw the `Unavailable` exception if the `Control` cannot provide the requested object, e.g., the transaction has terminated. The OTS implementation can restrict the ability for the `Terminator` and `Coordinator` to be used in other execution environments or threads; at a minimum the creator must be able to use them.

The `Control` object for a transaction can be obtained when the transaction is created either using the `TransactionFactory` or the `create_subtransaction` method defined by the `Coordinator` interface. In addition, it is possible to obtain a `Control` for the current transaction (associated with the current thread) using the `get_control` or `suspend` methods defined by the `Current` interface.

JBossTS specifics

The transaction creator (client thread) must be able to use its `Control`, but it is OTS implementation specific as to whether other threads can use this object. In the current version of *JBossTS* no restrictions are placed on the users of the `Control`.

It is implementation dependant as to how long a `Control` remains able to access a transaction after it terminates; in fact, 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*, if using the `Current` interface then all information about a transaction is destroyed when it terminates. Therefore, the programmer should not use any `Control` references to the transaction after issuing the `commit/rollback` operations.

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 the current version of *JBossTS*, both `Coordinators` and `Terminators` can be propagated between execution environments.

The Terminator interface

The `Terminator` interface supports operations to commit or rollback the transaction. Typically, these operations are used by the transaction originator. 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 have been marked rollback only, and all of its participants agree to commit. Otherwise, the `TRANSACTION_ROLLEDBACK` exception is thrown. If the `report_heuristics` parameter is true, the Transaction Service will report inconsistent results using the `HeuristicMixed` and `HeuristicHazard` exceptions.

When a transaction is committed, the coordinator will drive any registered `Resources` using their `prepare/commit` methods. It is the responsibility of these `Resources` to ensure that any state changes to recoverable objects are made permanent to guarantee the ACID properties.

When `rollback` is called, the registered `Resources` are responsible for guaranteeing 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 enforced by the resources.

JBossTS specifics

See the *JBossTS* specific section of `Control` for how long `Terminator` references remain valid after a transaction terminates.

When a transaction is committing it is necessary for it to make certain state changes persistent in order that it can recover in the event of a failure and either continue to commit, or rollback. To guarantee ACID properties, these state changes must be *flushed* to the persistence store implementation before the transaction can proceed to commit; if they are not, the application may assume that the transaction has committed when in fact the state changes may still reside within an operating system cache, and may be lost by a subsequent machine failure. By default, *JBossTS* ensures that such state changes are flushed. However, doing so can impose a

significant performance penalty on the application. To prevent transaction state flushes, set the `TRANSACTION_SYNC` variable to `OFF`.

When a transaction commits, if there is only a single registered resource then the transaction manager need not perform the two-phase protocol: a single phase commit is possible, and the outcome of the transaction will be completely determined by the resource. In a distributed environment this optimisation result in an important performance improvement. Therefore, by default *JBossTS* performs single phase commit in this situation. However, this can be overridden at runtime by setting the `COMMIT_ONE_PHASE` property variable to `NO`.

The Coordinator interface

Returned by the `get_coordinator` method of `Control`, the `Coordinator` interface supports the operations needed by resources to participate in the transaction. These participants are typically 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. Note, it is possible for a transaction to be terminated directly (i.e., through the `Terminator`) and then an attempt to terminate the transaction again through `Current` can be made (or vice versa). In this situation, an exception will be thrown for the subsequent termination attempt.

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

- `get_status`, `get_parent_status`, `get_top_level_status`: these operations 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 has been marked for rollback.
 - *StatusPrepared*: the transaction has been prepared, i.e., all subordinates have responded `VoteCommit`.
 - *StatusCommitted*: the transaction has completed commitment. 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 will ultimately return a different status.
 - *StatusNoTransaction*: no transaction is currently associated with the target object. This will occur after a transaction has completed.

- *StatusPreparing*: the transaction is in the process of preparing and has not yet determined the final outcome.
- *StatusCommitting*: the transaction is in the process of committing.
- *StatusRollingBack*: the transaction is in the process of rolling back.
- *is_same_transaction* et al: these operations can be used 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 has already been prepared. The `TRANSACTION_ROLLEDBACK` exception is raised if the transaction has been marked rollback only. If the `Resource` is a `SubtransactionAwareResource` *and* the transaction is a subtransaction, then this operation registers the resource with this transaction and indirectly with the top-level transaction when the subtransaction's ancestors have committed. Otherwise, the resource will only be registered with the *current* transaction. This operation returns a `RecoveryCoordinator` which can be used by this `Resource` during recovery. Note, there is no ordering of registered `Resources` implied by this operation, i.e., if A is registered after B the OTS is free to operate on them in any order when the transaction terminates. Therefore, `Resources` should not be implemented that assume (or require) such an ordering to exist.
- *register_subtran_aware*: registers the specified subtransaction aware resource with the current transaction only such that it will be informed when the subtransaction commits or rolls back. This method cannot be used to 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 such that it will be invoked prior to prepare and after the transaction has completed. `Synchronizations` can only be associated with top-level transactions, and an exception (`SynchronizationsUnavailable`) will be raised if an attempt is made 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 whose parent is the current transaction. The `Inactive` exception is raised if the current transaction has already been prepared/completed. An implementation of the Transaction Service need not support nested transactions, in which case the `SubtransactionsUnavailable` exception is raised.

JBossTS specifics

See *JBossTS* specific section of `Control` for how long `Coordinator` references remain valid after a transaction terminates.

JBossTS supports subtransactions. If this is not required, then set the `OTS_SUPPORT_SUBTRANSACTIONS` property variable to `NO`.

Heuristics

The OTS permits individual servers/resources to make so-called *Heuristic* decisions. Such 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. Heuristic decisions should be used with care and only in exceptional circumstances since there is the possibility that the decision will differ from that determined by the transaction service and will thus lead to a loss of integrity in the system. If a heuristic decision is made by a participant then an appropriate exception is raised during commit/abort processing. The possible heuristic exceptions are:

- *HeuristicRollback*
Raised on an attempted commit operation invocation to indicate that the resource has already unilaterally rolled back the transaction.
- *HeuristicCommit*
Raised on an attempted rollback operation invocation to indicate that the resource has already unilaterally committed the transaction.
- *HeuristicMixed*
Indicates that a heuristic decision has been made in which some updates have committed while others have been rolled back.
- *HeuristicHazard*
Indicates that a heuristic decision may have been made, and the disposition of some of the updates is unknown. For those updates which are known they have either all been committed or all rolled back.

Heuristics are ordered such that `HeuristicMixed` takes priority over `HeuristicHazard`. Heuristic decisions are only reported back to the originator if the `report_heuristics` argument was set to true when the `commit` operation was invoked.

Current

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

- `begin`: a new transaction is created, and associated with the current thread. If the client thread is currently associated with a transaction, and the OTS implementation supported nested transactions, the new transaction is 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 may be thrown. The thread's notion of the current context will be modified to this transaction.

- `commit`: the transaction commits; 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 transaction context is returned to the state prior to the `begin` request.
- `rollback`: the transaction rolls back; 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 transaction context is returned to the state prior to the `begin` request.
- `rollback_only`: the transaction is modified so the only possible outcome is for it to rollback. If the transaction has already been terminated (or is in the process of terminating) an appropriate exception will be thrown.
- `get_status`: returns the status of the current transaction, or `StatusNoTransaction` if there is no transaction associated with the thread.
- `set_timeout`: modifies the timeout associated with *top-level transactions* for subsequent `begin` requests *for this thread only*. Subsequent transactions will be subject to being rolled back if they have not completed after the specified number of seconds. It is implementation dependant as to what timeout value will be used for a transaction if one is not explicitly specified prior to `begin`. *JBossTS* uses a value of zero, i.e., no timeout will be associated with the transaction. There is no interface in the OTS for obtaining the current timeout associated with a thread. However, *JBossTS* provides additional support for this; see the *JBossTS* specific section.
- `get_control`: if the client thread is not associated with a transaction, a null object reference is returned. Otherwise, a `Control` object is returned that represents the current transaction. The operation is not dependent on the state of the transaction; in particular, it does not raise the `TRANSACTION_ROLLEDBACK` exception.
- `suspend`: if the client thread is not associated with a transaction, a null object reference is returned. Otherwise, an object that represents the transaction context is returned. This object can be given to the `resume` operation to re-establish this context in a thread. The operation is not dependent on the state of the transaction; in particular, it does not raise the `TRANSACTION_ROLLEDBACK` exception. When this call returns, the current thread has no transaction context associated with it.
- `resume`: if the parameter is a null object reference, the client thread becomes associated with no transaction. Otherwise, if the parameter is valid in the current execution environment, the client thread becomes associated with that transaction. Any previous transaction will be forgotten by the thread.

If we consider the creation of a top-level transaction using the `Current` pseudo-object, the course of events within the OTS can be represented as follows:

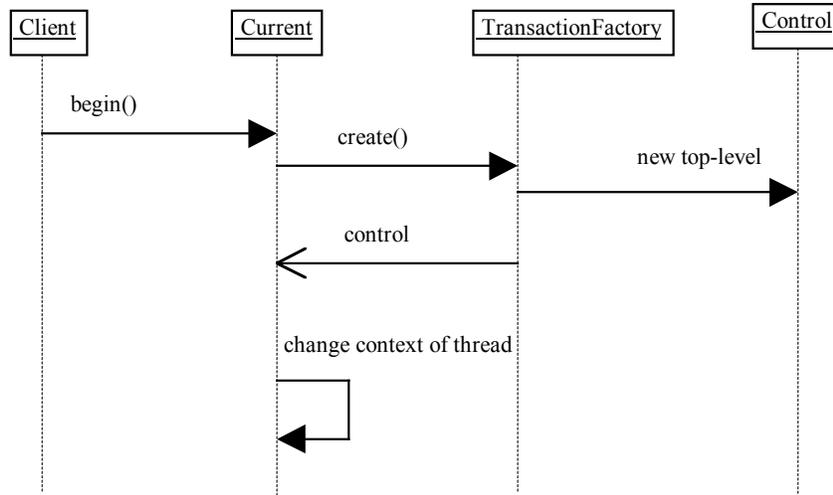


Figure 8: top-level transaction creation (indirect).

Likewise, creation of a subtransaction through Current can be represented as shown below:

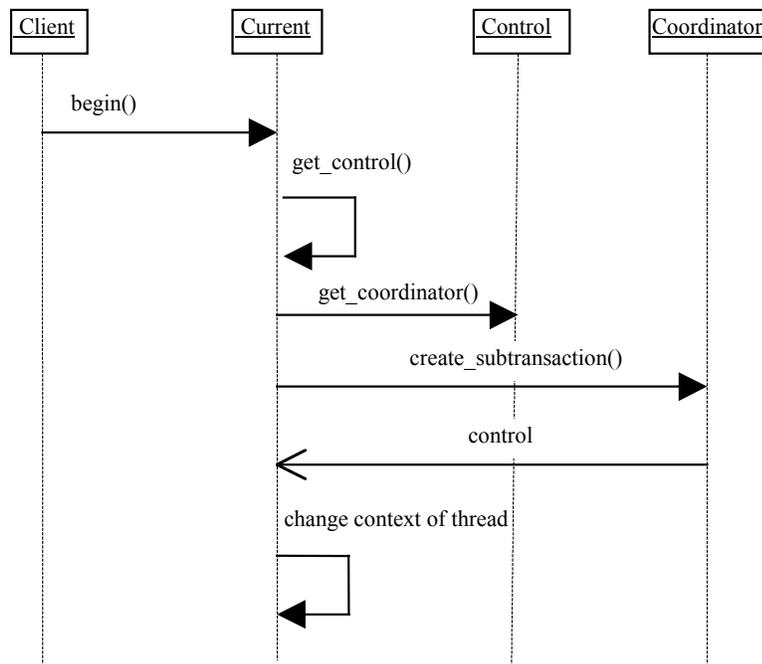


Figure 9: subtransaction creation (indirect).

Given the descriptions of `Current`, `indirect context management`, `Resource`, `SubtransactionAwareResource` and `Synchronization`, we can consider the course of events involved in terminating a top-level transaction and a subtransaction. These are illustrated in the following diagrams.

JBossTS specifics

The pseudo-object should be obtained using factory finder of the life-cycle service. However, very few ORBs support this. Therefore, in the current implementation of JBossTS the programmer should use the `get_current` method on the *JBossTS* class `OTS`. This class hides any ORB specific mechanisms required for obtaining `Current`.

If no timeout value has previously been associated with `Current` by a thread then *JBossTS* uses a default value of zero, i.e., no timeout will be associated with the transaction. To override this default behaviour, see Section 0. 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 its own `TransactionFactory` which is co-located with it. By setting the `OTS_TRANSACTION_MANAGER` variable to `YES` this can be overridden at runtime.

The transaction factory is located in the `/bin` directory of the *JBossTS* distribution, and can be started by executing the `OTS` script. `Current` locates the factory in an ORB specific manner, e.g., using the name service, through `resolve_initial_references`, or via the `CosServices.cfg` file located in the `/etc` directory of the *JBossTS* distribution. This file is similar to `resolve_initial_references`, and is automatically updated when the transaction factory is started on a particular machine. This file must be copied to the installation of all machines which require to share the same transaction factory.

JBossTS supports subtransactions. If this is not required, then set the `OTS_SUPPORT_SUBTRANSACTIONS` property variable to `NO`.

The `setCheckedAction` method can be used to override the `CheckedAction` implementation associated with each transaction the thread creates.

Statistics gathering

By default, the *JBossTS* does not maintain any history information about transactions. However, by setting the `com.arjuna.ats.arjuna.coordinator.enableStatistics` property variable to `YES`, the transaction service will maintain information about the number of transactions created, and their outcomes. This information can be obtained during the execution of a transactional application via the `com.arjuna.TxCore.Atomic.TxStats` class:

```
public class TxStats
{
    /**
     * Returns the number of transactions (top-level and nested)
     * created so far.
     */

    public static int numberOfTransactions ();
```

```

    /**
     * Returns the number of nested (sub) transactions created so far.
     */

    public static int numberOfNestedTransactions ();

    /**
     * Returns the number of transactions which have terminated with
     * heuristic outcomes.
     */

    public static int numberOfHeuristics ();

    /**
     * Returns the number of committed transactions.
     */

    public static int numberOfCommittedTransactions ();

    /**
     * Returns the number of transactions which have rolled back.
     */

    public static int numberOfAbortedTransactions ();
}

```

Resource

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

```

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 ();
};

```

Interface 3: The Resource Interface.

The `Resource` interface defines the operations invoked by the transaction service. Each `Resource` object is implicitly associated with a single top-level transaction. A given `Resource` should not be registered with the same transaction more than once. This is because when a `Resource` is told to prepare/commit/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 register objects that support the `Resource` interface with the current transaction using the `register_resource` method of the transaction's

Coordinator interface. An object supporting the `Coordinator` interface will either be passed as a parameter (if explicit propagation is being used) or may be retrieved using operations on the `Current` interface (if implicit propagation is used). If the transaction is a subtransaction, then the `Resource` will not be informed of the subtransaction's completion, and will be registered with its parent upon commit. This is illustrated below, where for simplicity we assume the hierarchy is only 2 deep.

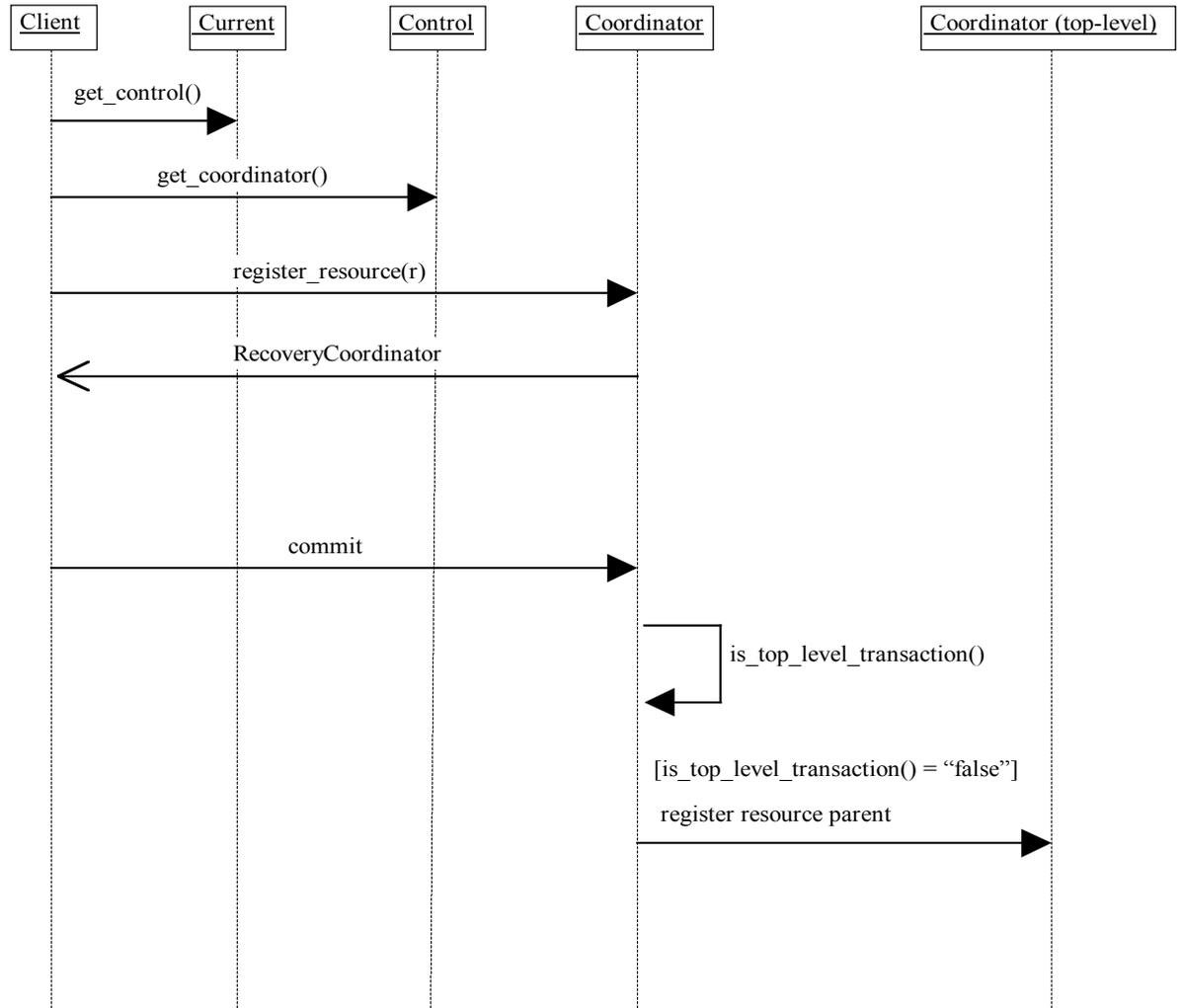


Figure 10: Registering a Resource with a transaction.

A given `Resource` should not be registered with the same transaction more than once or it will receive multiple termination calls. It *must* not be registered with more than one transaction. This is because when a `Resource` is told to prepare/commit/abort it must do so on behalf of a specific transaction; however, the `Resource` methods do not specify the transaction identity. Therefore, the identity is implicit since the `Resource` should only be associated with a single transaction.

A single `Resource` or group of `Resources` are responsible for guaranteeing the ACID properties for the recoverable object they represent. The work `Resources` should perform can be summarized for each phase of the commit protocol:

- *prepare*: if no persistent data associated with the resource has been modified within the transaction, then the `Resource` can return `VoteReadOnly` and forget about the transaction; it need not be contacted during the second phase of the commit protocol since it has made no state changes to either commit or roll back. If the resource is able to 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 will eventually either commit or rollback. 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. For example, if the `Resource` said it could commit and then decides it cannot, and must rollback. Heuristic decisions must be made persistent and remembered by the `Resource` until the transaction coordinator issues the `forget` method; this essentially tells the `Resource` that the heuristic decision has been noted (and possibly resolved).
- *rollback*: if necessary, 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, heuristic exceptions can be raised. 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 is used to report heuristic decisions related to that resource. See Section 0 for how to disable the use of the one-phase commit protocol.
- *forget*: this operation is performed if the resource raised a heuristic exception. Once the coordinator has determined that the heuristic situation has been addressed, it will issue `forget` on the resource. The resource can then forget all knowledge of the transaction.

SubtransactionAwareResource

An OTS implementation can support subtransactions. Recoverable objects that wish to participate within a nested transaction may support the `SubtransactionAwareResource` interface, a specialization of the `Resource` interface.

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

Only by registering a `SubtransactionAwareResource` will a recoverable object be informed of the completion of a nested transaction. Registration is performed using either the `register_resource` or `register_subtran_aware` methods of the `Coordinator` or `Current` interfaces. Generally a recoverable object will register `Resources` to participate within the completion of top-level transactions, and `SubtransactionAwareResources` to be notified of the completion of subtransactions. The `commit_subtransaction` method is passed a reference to the parent transaction in order to allow subtransaction resources to register with these transactions, e.g., to perform propagation of locks.

It is important to realise that `SubtransactionAwareResources` are informed of the completion of a transaction *after* it has terminated, i.e., they cannot affect the outcome of the transaction. It is implementation specific as to how the OTS implementation will deal with any exceptions raised by `SubtransactionAwareResources`.

A `SubtransactionAwareResource` is registered with a transaction using either the `register_resource` method, or the `register_subtran_aware` method. Both methods have subtly different requirements and effects:

- `register_resource`: if the transaction is a subtransaction then the resource will be informed of its completion, and automatically registered with the subtransaction's parent if it commits.
- `register_subtran_aware`: if the transaction is not a subtransaction, then an exception will be thrown. Otherwise, the resource will be informed of the completion of the subtransaction. However, unlike `register_resource`, it will *not* be propagated to the subtransaction's parent if the transaction commits. If the resource requires this it must re-register using the supplied parent parameter.

Both of these registration techniques are illustrated in the following diagrams. Figure 11 shows how a `SubtransactionAwareResource` is registered with a subtransaction using the `register_subtran_aware` method:

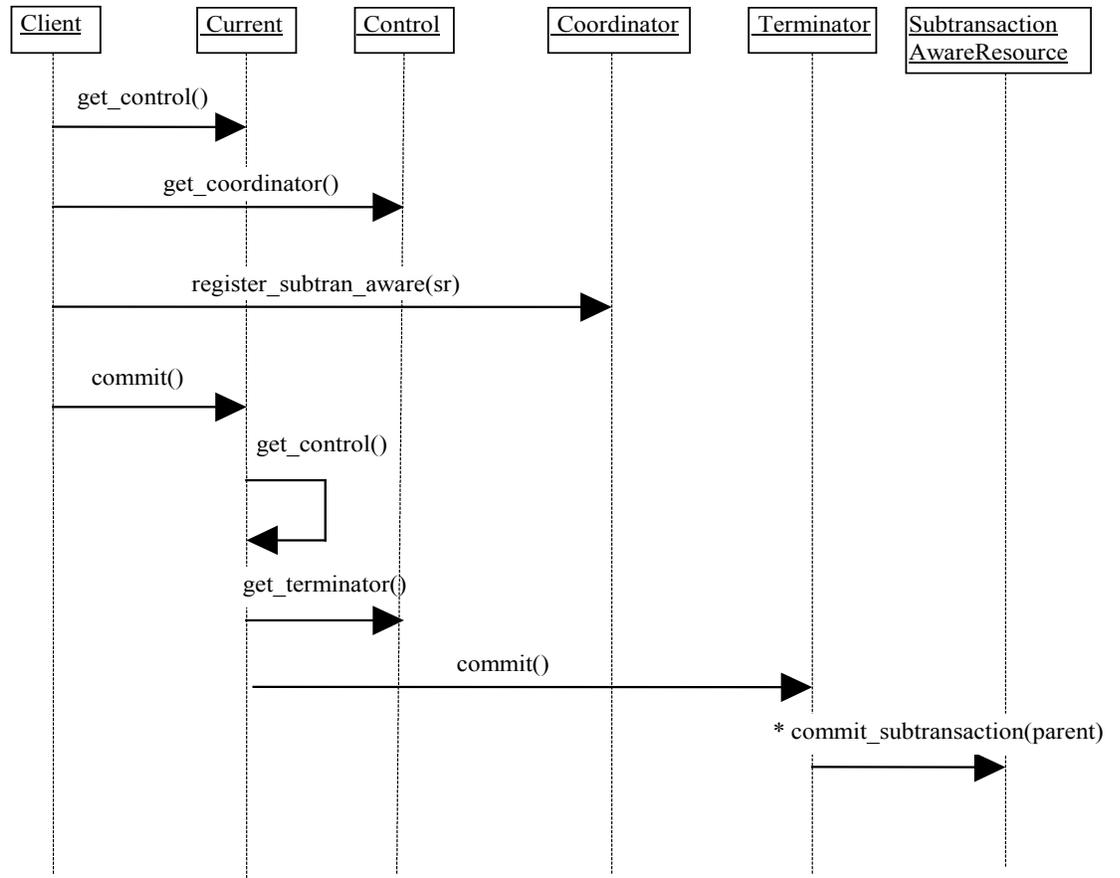


Figure 11: Registering a SubtransactionAwareResource with a subtransaction.

Figure 12 illustrates the mechanisms involved when a SubtransactionAwareResource is registered using the `register_resource` operation:

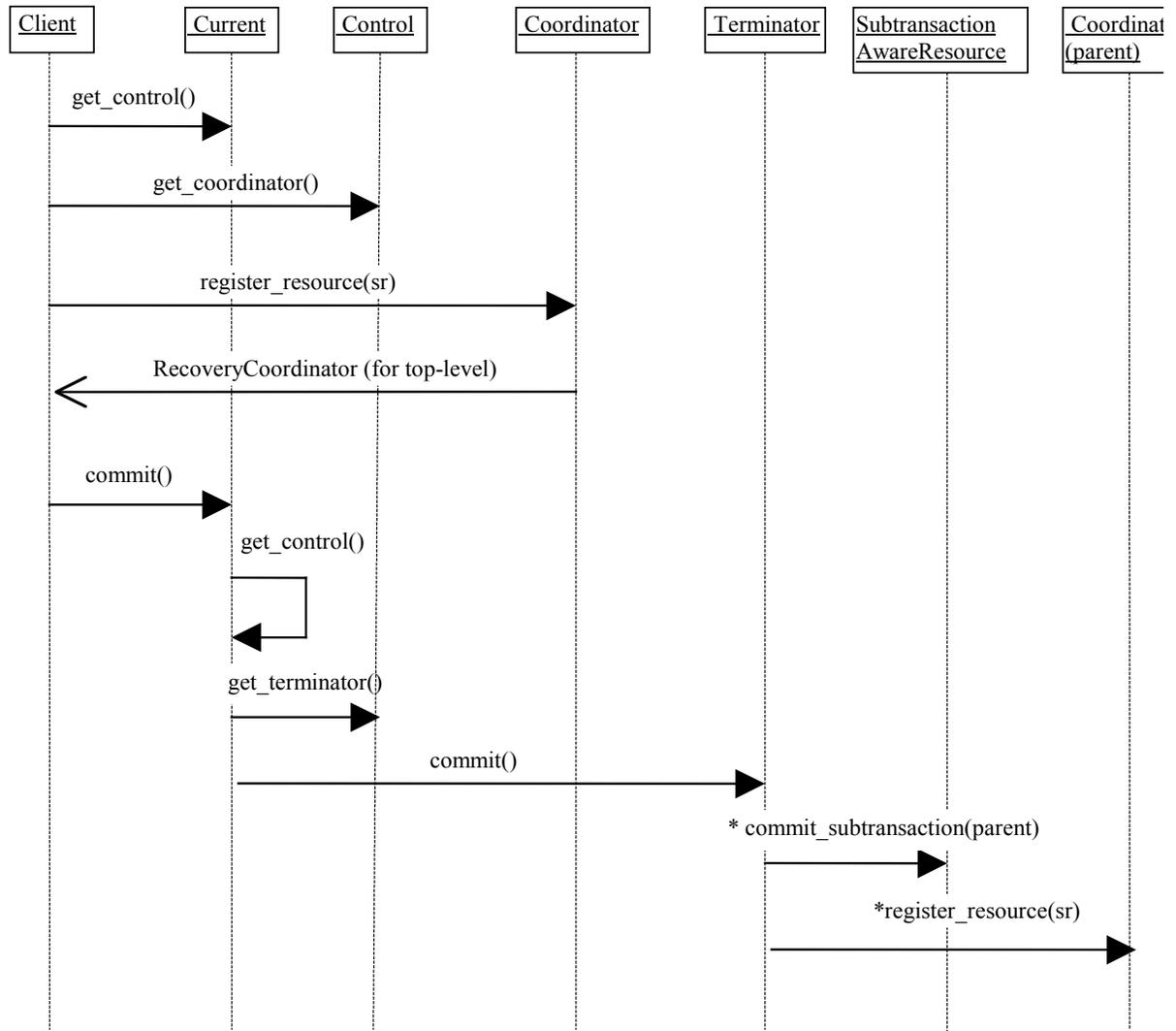


Figure 12: Registering a SubtransactionAwareResource with a subtransaction as a Resource.

In either case, the resource cannot effect the outcome of the transaction completion. It is only informed of the transaction decision, and should attempt to act accordingly. However, if the resource cannot (e.g., it cannot commit when `commit_subtransaction` is called) then it can raise an exception, and the OTS will deal with it in an implementation specific manner (for example, it is valid for the OTS to ignore such exceptions).

JBossTS specifics

A `SubtransactionAwareResource` which raises an exception to the commitment of a transaction may result in inconsistencies within the transaction if other `SubtransactionAwareResources` have previously been told that the transaction committed. Therefore, *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 0 for further details.

The Synchronization interface

If an object wishes to be informed that a transaction is about to commit, it can register an object which is an instance of the Synchronization interface, using the `register_synchronization` operation of the Coordinator interface. Synchronizations are typically employed to flush volatile (cached) state, which may be being used to improve performance of an application, to a recoverable object or database prior to the transaction committing. Note, Synchronizations can only be associated with top-level transactions, and an exception will be raised if an attempt is made to register a Synchronization with a subtransaction. Each object supporting the Synchronization interface is associated with a single top-level transaction.

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

The method `before_completion` is called prior to the start of the two-phase commit protocol, and `after_completion` is called after the protocol has completed (the final status of the transaction is given as a parameter). If `before_completion` raises an exception, the transaction will rollback. Any exceptions thrown by `after_completion` will have no effect on the transaction outcome. The OTS only requires Synchronizations to be invoked if the transaction commits; if it rolls back, registered Synchronizations will not be informed. See Section 0 for further details.

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

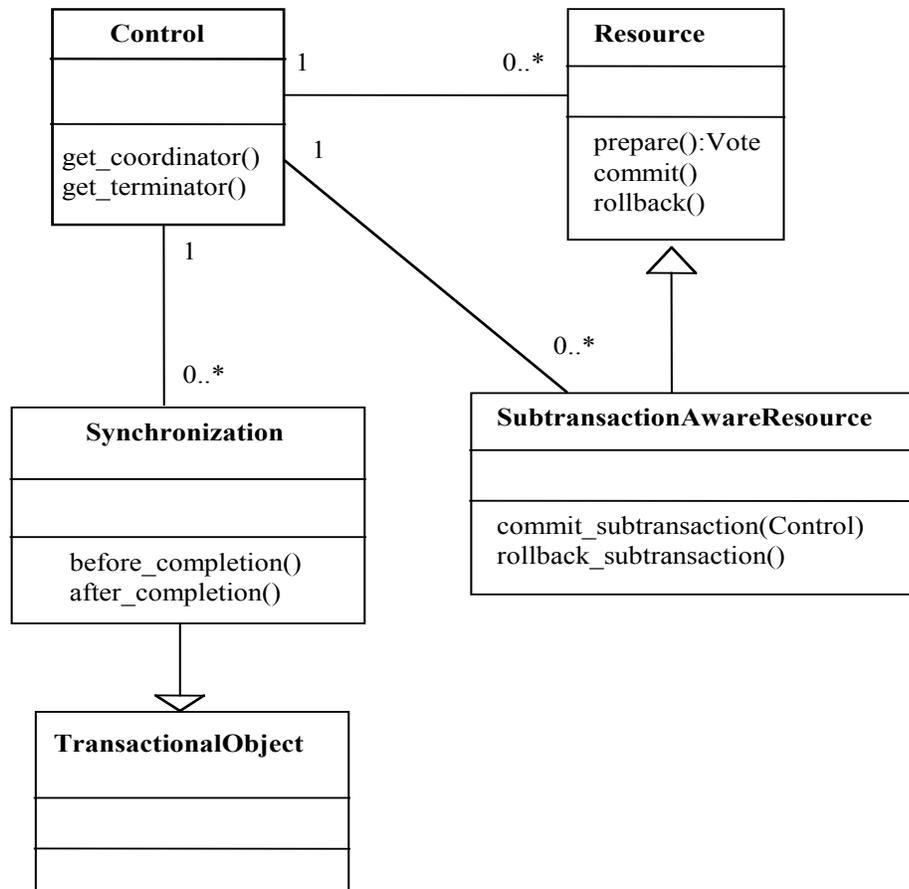


Figure 13: Transaction relationship.

JBossTS specifics

Synchronizations are required to be called prior to the start of the top-level transaction commit protocol, and after it completes. However, if the transaction is told to rollback (e.g., by the application program invoking the `rollback` method on the `Current` pseudo-object), the Synchronizations associated with the transaction will not be contacted. To override this such that Synchronizations are called however the transaction terminates, set the `OTS_SUPPORT_ROLLBACK_SYNC` property variable to `YES`.

When using distributed transactions and interposition, a local proxy for the top-level transaction coordinator will be created for any recipient of the transaction context. The proxy looks like a `Resource/SubtransactionAwareResource` and registers itself as such with the actual top-level transaction coordinator, and is used by the local recipient for registering Resources and Synchronizations locally. See Section 0 for further details. This can affect how Synchronizations are invoked during top-level transaction commit: normally all Synchronizations are invoked before any Resource/SubtransactionAwareResource objects are processed. However, with interposition, only those Synchronizations registered locally to the transaction coordinator will be called; Synchronizations registered with remote participants will only be called when the interposed proxy is invoked, which may be after locally registered

Resource/SubtransactionAwareResource objects. *JBossTS* provides a mechanism whereby all Synchronizations are invoked before any Resource/SubtransactionAwareResource, no matter where they are registered. To enable this feature set the `OTS_SUPPORT_INTERPOSED_SYNCHRONIZATION` property variable to YES.

Transactions and registered resources

The relationship between a transaction `Control` and resources registered with it is shown in the following diagram.

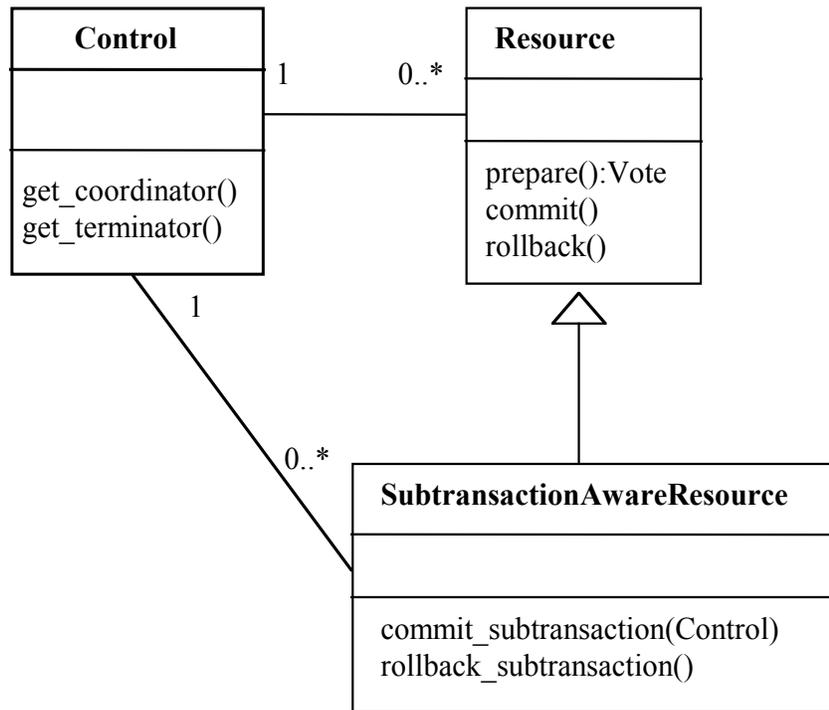


Figure 14: Control and Resource relationship.

Figure 15 shows the course of events when committing a subtransaction which has had both Resource and SubtransactionAwareResource objects registered with it; we assume that the SubtransactionAwareResources were registered using `register_subtran_aware`. The Resources are not informed of the termination of the subtransaction, whereas the SubtransactionAwareResources are. However, only the Resources are automatically propagated to the parent transaction.

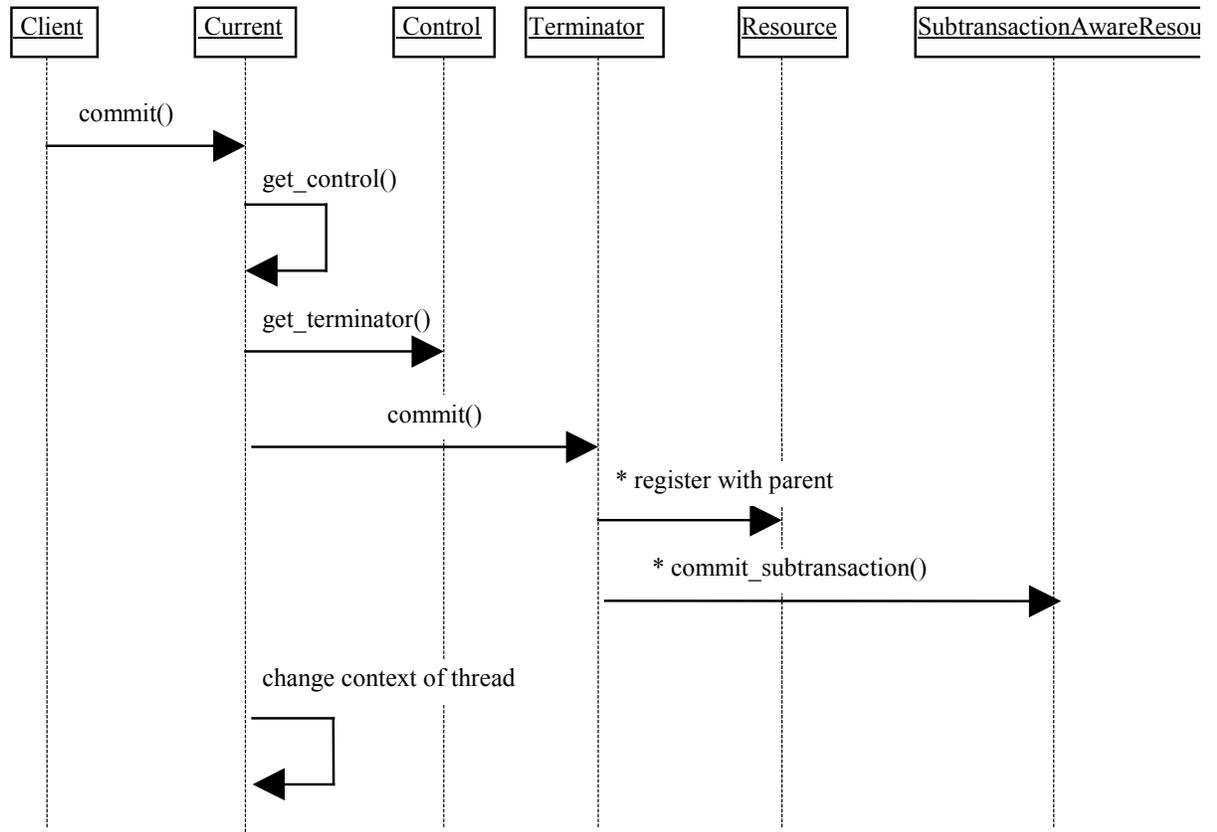


Figure 15: Subtransaction commit.

Figure 16 illustrates what happens when the subtransaction rolls back. Any registered resources are discarded (not shown), and `SubtransactionAwareResources` are informed of the transaction outcome.

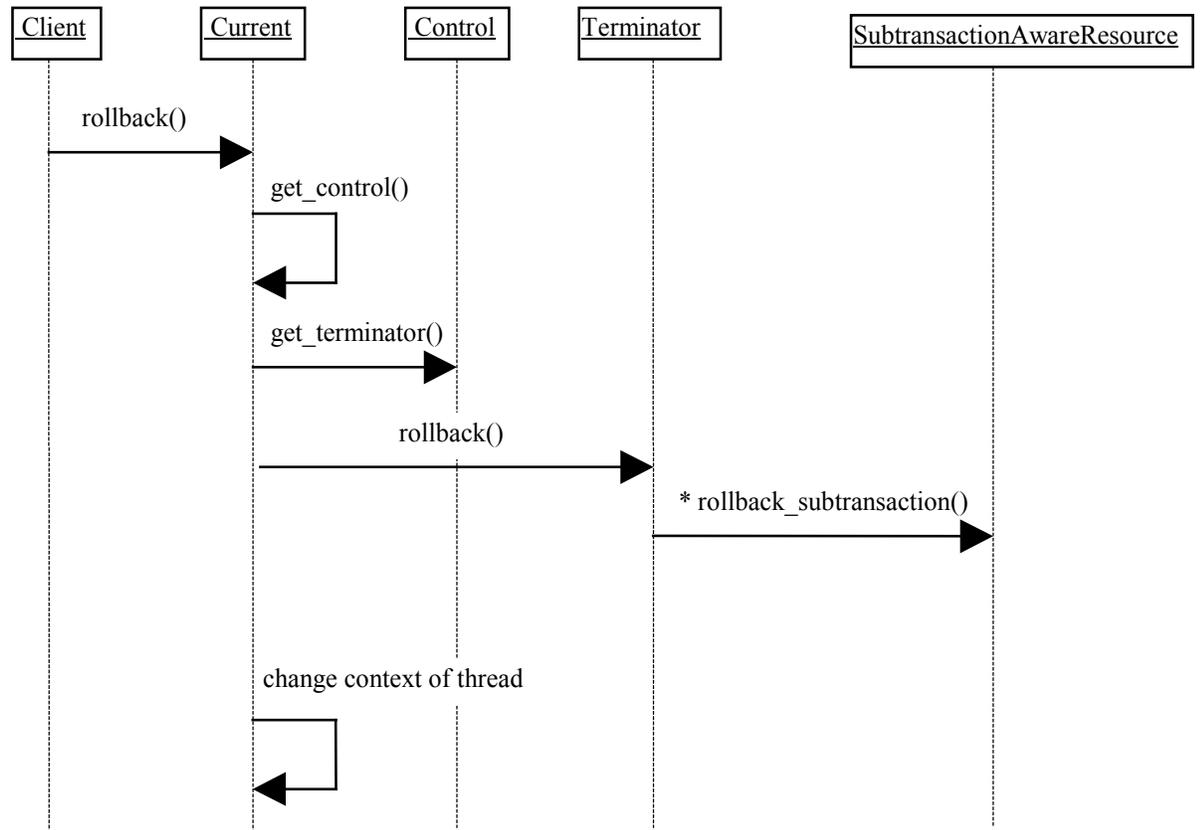


Figure 16: Subtransaction rollback.

Figure 17 shows the activity diagram for committing a top-level transaction; those subtransactions within the top-level transaction which have also successfully committed will have propagated `SubtransactionAwareResources` to the top-level transaction, and these will then participate within the two-phase commit protocol. As can be seen, however, prior to `prepare` being called, any registered `Synchronizations` are first contacted. Because we are using indirect context management, when the transaction commits, the transaction service changes the invoking thread's transaction context.

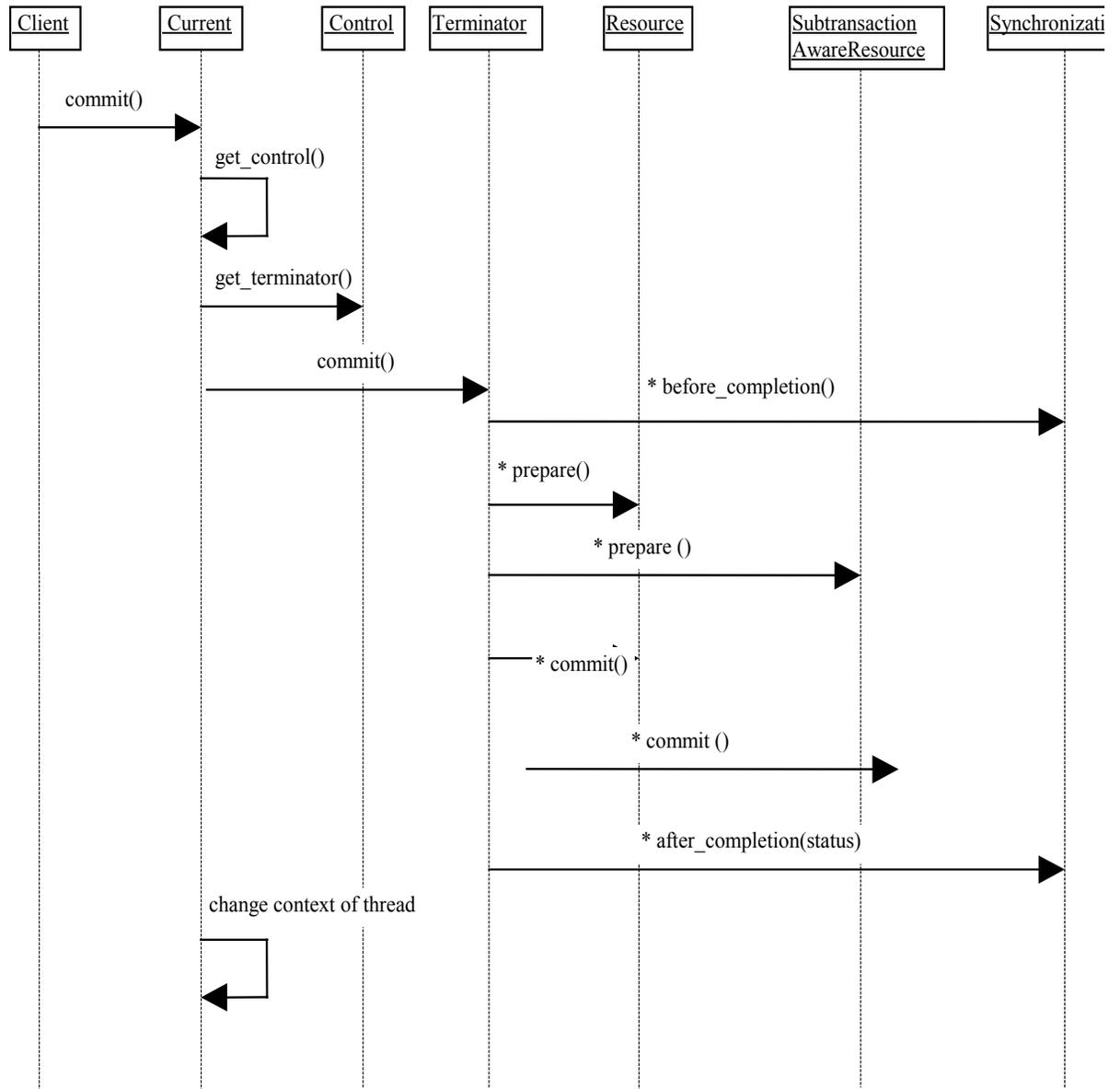


Figure 17: Top-level transaction commit.

Figure 18 shows rolling back a top-level transaction.

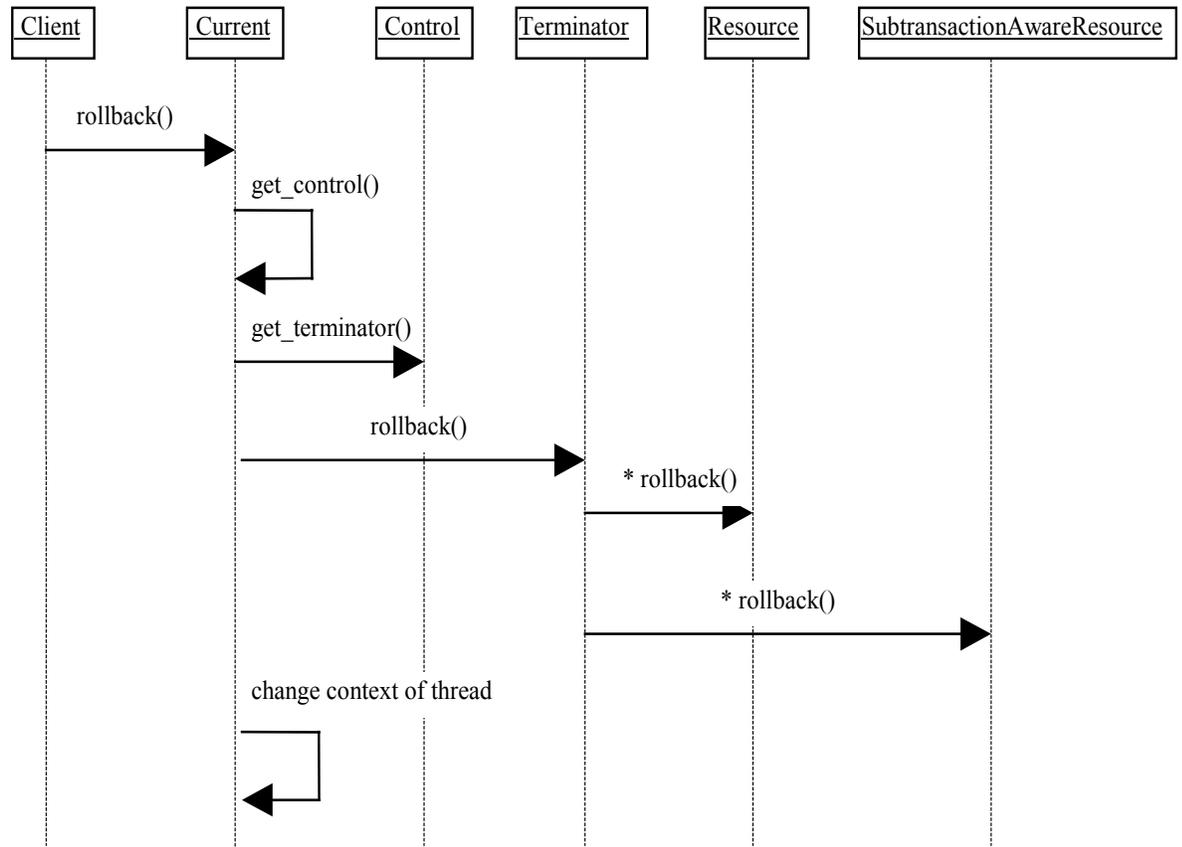


Figure 18: Top-level transaction rollback.

TransactionalObject interface

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

An OTS implementation is not required to initialise the transaction context of every request handler. It is required to do so only 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, i.e., the transaction context is null.

In a single-address space application (i.e., all objects reside within the same process), transaction contexts are implicitly shared between “clients” and objects, regardless of whether or not the objects support the `TransactionalObject` interface. Therefore, in order to preserve distribution transparency, where implicit transaction propagation is supported *JBossTS* can be made to always propagate transaction contexts to objects. The default, which

is only to propagate if the object is a `TransactionalObject`, can be overridden by setting the environment variable `OTS_ALWAYS_PROPAGATE_CONTEXT` to `NO`.

By default, *JBossTS* does not require that objects supporting the `TransactionalObject` interface are invoked within the scope of a transaction. Rather, this it is left up to the object to determine whether it should be invoked within a transaction; if so, it should throw the `TransactionRequired` exception. This can be overridden by setting the `OTS_NEED_TRAN_CONTEXT` shell environment variable to `YES`.

Note: it is important to ensure that the settings for `OTS_ALWAYS_PROPAGATE_CONTEXT` and `OTS_NEED_TRAN_CONTEXT` are identical at the client and the server. Failure to do this may result in abnormal termination of the application, depending upon which ORB is being used.

JBossTS specifics

In a single-address space application (i.e., all objects reside within the same process), transaction contexts are implicitly shared between “clients” and objects, regardless of whether or not the objects support the `TransactionalObject` interface. Therefore, in order to preserve distribution transparency, where implicit transaction propagation is supported *JBossTS* will always propagate transaction contexts to objects. The default can be overridden by setting the environment variable `OTS_ALWAYS_PROPAGATE_CONTEXT` to `NO`.

By default, *JBossTS* does not require that objects supporting the `TransactionalObject` interface are invoked within the scope of a transaction. Rather, this it is left up to the object to determine whether it should be invoked within a transaction; if so, it should throw the `TransactionRequired` exception. This can be overridden by setting the `OTS_NEED_TRAN_CONTEXT` shell environment variable to `YES`.

Interposition

OTS objects supporting the interfaces such as the `Control` interface are simply standard CORBA objects. This implies that when an interface is passed as a parameter in some 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 due to the overheads of remote invocation. For example, when the server registers a `Resource` with the current transaction that has the potential to be a remote invocation back to the originator of the transaction.

To avoid this overhead, an implementation of the OTS may support *interposition*. This permits a server to create a local control object which acts as a local coordinator fielding registration requests that would normally have been passed back to the originator. This surrogate must register itself with the original coordinator to enable it to correctly participate in the commit protocol. Interposed coordinators effectively form a tree structure with their parent coordinators.

If interposition is required, then the programmer must ensure that *JBossTS* is correctly initialized prior to objects being created; obviously it is necessary for both client and server to

agree to use interposition. Interposition is only possible on those ORBs which either support filters/interceptors, or the `CostSPortability` interface, since interposition implicitly requires the use of implicit transaction propagation. (Currently this is Orbix 2000). The programmer *must* perform the following:

- set the `OTS_CONTEXT_PROP_MODE` property variable to `INTERPOSITION`. If using Orbix 2000, see the Orbix 2000 configuration section in the *Administrator's Guide*.

If using the *JBossTS* advanced API then interposition is *required*.

Asynchronously committing a transaction

By default, *JBossTS* executes the commit protocol of a top-level transaction in a synchronous manner, i.e., all registered resources will be told to prepare in order by a single thread, and then they will be told to commit or rollback. This has several possible disadvantages:

- In the case of many registered resources, the prepare operating can logically be invoked in parallel on each resource. The disadvantage is that if an “early” resource in the list of registered resource forces a rollback during prepare, possibly many prepare operations will have been made needlessly.
- In the case where heuristic reporting is not required by the application, the second phase of the commit protocol can be done asynchronously, since its success or failure is not important.

Therefore, *JBossTS* provides runtime options to enable possible threading optimizations. By setting the `ASYNC_PREPARE` environment variable to `YES`, during the prepare phase a separate thread will be created for each registered participant within the transaction. By setting `ASYNC_COMMIT` to `YES`, a separate thread will be created to complete the second phase of the transaction *if knowledge about heuristics outcomes is not required*, i.e., `report_heuristics` is set to `FALSE`.

The RecoveryCoordinator

A reference to a `RecoveryCoordinator` is returned as a result of successfully calling `register_resource` on the transaction `Coordinator`. This object, which is implicitly associated with a single `Resource`, can be used to drive the `Resource` through recovery procedures in the event of a failure occurring during the transaction.

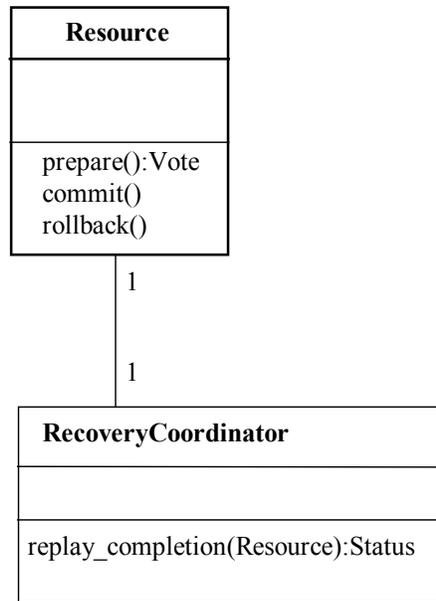


Figure 19: Resource and RecoveryCoordinator relationship.

Checked transaction behaviour

The OTS supports both checked and unchecked transaction behaviour. Checked transactions have a number of integrity constraints including:

- Ensuring that a transaction will not commit until all transactional objects involved in the transaction have completed their transactional requests.
- Ensuring that only the transaction originator can commit the transaction

In fact, checked transactional behaviour might be best described as classical transaction behaviour and is widely implemented. Checked behaviour is only possible if implicit propagation is used since the use of explicit propagation prevents the OTS from tracking which objects are involved in the transaction with any certainty.

In contrast, unchecked behaviour allows relaxed models of atomicity to be implemented. Any use of explicit propagation implies the possibility of unchecked behaviour since it is the application programmer's responsibility to ensure the correct behaviour. Note that even using implicit propagation unchecked behaviour may still be possible since a server could unilaterally abort or commit the transaction via the `Current` interface.

Some Transaction Service implementations will enforce checked behaviour for the transactions they support, to provide an extra level of transaction integrity. The purpose of the checks is to ensure that all transactional requests made by the application have completed their processing before the transaction is committed. A checked Transaction Service guarantees that commit will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. Rolling back the transaction does not require such a check, since all outstanding transactional activities will eventually rollback if they are not told to commit.

There are many possible implementations of checking in a Transaction Service. One provides equivalent function to that provided by the request/response inter-process communication models defined by X/Open. The X/Open Transaction Service model of checking is particularly important because it is widely implemented. It describes the transaction integrity guarantees provided by many existing transaction systems. These transaction systems will 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 of checking 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 behaviour.

This implementation of checked behaviour depends on *implicit transaction propagation*. When implicit propagation is used, the objects involved in a transaction at any given time may be represented as a tree, the request tree for the transaction. The beginner of the transaction is the root of the tree. Requests add nodes to the tree, 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 cannot know which objects are or will be involved in the transaction; that is, a request tree cannot be constructed or assured. Therefore, the use of explicit propagation is not permitted by a Transaction Service implementation that enforces X/Open-style checked behaviour.

Applications that use synchronous requests implicitly exhibit checked behaviour. For applications that use deferred synchronous requests, in a transaction where all clients and objects are in the domain of a checking Transaction Service, the Transaction Service can enforce this property by applying a *reply check* and a *commit check*. The Transaction Service must also apply a *resume check* to ensure that the transaction is only resumed by application programs in the correct part of the request tree.

- *reply check*: before allowing an object to reply to a transactional request, a check is made to ensure that the object has received replies to all its 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, that is, it cannot be successfully committed. A Transaction Service may check that a reply is issued within the context of the transaction associated with the request.
- *commit check*: before allowing commit to proceed, a check is made to ensure that i) the commit request for the transaction is being issued from the same execution environment that created the transaction, and ii) the client issuing commit has received replies to all the deferred synchronous requests it made that caused the propagation of the transaction.
- *resume check*: before allowing a client or object to associate 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 would be

true if the thread either created the transaction or received it in a transactional operation.

JBossTS specifics

Where support from the ORB is available, *JBossTS* supports X/Open checked transaction behaviour. However, unless the `OTS_CHECKED_TRANSACTIONS` property variable is set to `YES` this is disabled by default.

Note: checked transactions are only possible if using a co-located transaction manager, i.e., the use of a separate transaction manager processes does not allow checked transactions to be provided by the system.

In a multi-threaded application, multiple threads may be associated with a transaction during its lifetime, i.e., the thread's share the context. In addition, it is possible that 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 is terminated. By default, JBossTS will issue a warning if a thread terminates a transaction when other threads are still active within it; however, it will allow the transaction termination to continue. Other solutions to this problem are possible, e.g., blocking the thread which is terminating the transaction until all other threads have disassociated themselves from the transaction context. Therefore, JBossTS provides the `com.arjuna.ats.arjuna.coordinator.CheckedAction` class, which allows the thread/transaction termination policy to be overridden. Each transaction has an instance of this class associated with it, and application programmers can provide their own implementations on a per transaction basis.

```
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 are active threads within it, the system will invoke 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 will continue. Obviously the state of the transaction at this point may be different from that when `check` was called, e.g., the transaction may subsequently have been committed.

The `CheckedAction` instance associated with a given transaction is set using the `setCheckedAction` method of `Current`.

Summary of JBossTS implementation decisions

The following list summarizes the run-time and compile-time design decisions used by *JBossTS*.

- 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 using explicit control (via the `TransactionFactory` and `Terminator`), use the `destroyControl` method of the `OTS` class in `com.arjuna.CosTransactions` to signal when the information can be garbage collected.
- `Coordinators` and `Terminators` can be propagated between execution environments.
- if an attempt is made to commit a transaction when there are still active subtransactions within it, *JBossTS* will rollback the parent and the subtransactions.
- there is 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 will guarantee that all resources used within the subtransaction will be returned to a consistent state. Support for subtransactions can be disabled at runtime by setting the `OTS_SUPPORT_SUBTRANSACTIONS` variable to `NO`.
- `Current` should be obtained using the `get_current` method of the `OTS`.
- a timeout value of zero seconds is assumed for a transaction if not specified using `set_timeout`.
- by default, `Current` does not use a separate transaction manager server. This can be overridden by setting the `OTS_TRANSACTION_MANAGER` environment variable. How the `OTS` locates the transaction manager is ORB specific. See the chapter on configuring *JBossTS*.
- checked transactions are not enabled by default. To enable them, set the `OTS_CHECKED_TRANSACTIONS` property to `YES`.

Constructing an OTS application

Important notes for JBossTS

The following are important notes for programmers when building any application using *JBossTS* (i.e., those which use the raw OTS interfaces or the extended *JBossTS* API.)

Initialisation

It is important that JBossTS is correctly initialised prior to any application object being created. In order to guarantee this, the programmer should use the `initORB` and `initBOA/initPOA` methods described in the Orb Portability Guide. Consult the Orb Portability Guide if direct use of the `ORB_init` and `BOA_init/create_POA` methods provided by the underlying ORB is required.

Implicit context propagation and interposition

If implicit context propagation and interposition are required, then the programmer must ensure that *JBossTS* is correctly initialised prior to objects being created. Implicit context propagation is only possible on those ORBs which either support filters/interceptors, or the `CostSPortability` interface. Depending upon which type of functionality is required, the programmer must perform the following:

- Implicit context propagation:
 - set the `OTS_CONTEXT_PROP_MODE` property variable to `CONTEXT`. If using Orbix 2000, see the Orbix 2000 configuration section in the Administrator's Guide.
- Interposition:
 - set the `OTS_CONTEXT_PROP_MODE` property variable to `INTERPOSITION`. If using Orbix 2000, see the Orbix 2000 configuration section in the Administrator's Guide.

If using the *JBossTS* API the interposition *must* be used.

Writing applications using the raw OTS interfaces

To participate within an OTS transaction, a programmer must be concerned with:

- creating `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction/subtransaction. These resources

are responsible for the persistence, concurrency control, and recovery for the object. The OTS will invoke these objects during the prepare/commit/abort phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.

- registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.
- ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions are correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.
- in the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction.

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer. The interfaces defined within the OTS specification are too low-level for most application programmers. Therefore, we have designed *JBossTS* to make use of raw Common Object Services interfaces but provide a higher-level API for building transactional applications and frameworks. This API automates much of the above activities concerned with participating in an OTS transaction.

Transaction context management

If implicit transaction propagation is being used the programmer should ensure that appropriate objects support the `TransactionalObject` interface; otherwise, the transaction contexts must be explicitly passed as parameters to the relevant operations.

A transaction originator: indirect and implicit

In the code fragments below, 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:

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

The program commits the transaction associated with the client thread. The `report_heuristics` argument is set to `false` so no report will be made by the Transaction Service about possible heuristic decisions.

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

Transaction originator: direct and explicit

In the following example, a 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.

```
...
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();
...
```

The client issues requests, some of which involve transactional objects, in this case explicit propagation of the context is used. 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 no report will be made by the Transaction Service about possible heuristic decisions.

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

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` has made its decision to commit or rollback the transaction. The application is not required 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 will not complete until the `Coordinator` has completed the commit protocol with all `Resource` objects involved in the transaction. This guarantees that the application will be informed of any non-atomic outcomes of the transaction via the `HeuristicMixed` or

HeuristicHazard exceptions, but increases the application-perceived elapsed time for the commit operation.

Implementing a recoverable server

A Recoverable Server includes at least one transactional object and one resource object. The responsibilities of each of these objects are explained in the following sections.

Transactional object

The responsibilities of the transactional object are to implement the transactional object's operations, and to register a Resource object with the Coordinator so commitment of the Recoverable Server's resources, including any necessary recovery, can be completed.

The Resource object identifies the involvement of the Recoverable Server in a particular transaction. This means a Resource object may only be registered in one transaction at a time. A different resource object must be registered 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 need only be done on Coordinators which have the same hash code as the Coordinator of the current request.

Resource object

The responsibilities of a Resource object are to participate in the completion of the transaction, to update the Recoverable Server's resources in accordance with the transaction outcome, and ensure termination of the transaction, including across failures. The protocols that the Resource object must follow were described in [Chapter 4](#).

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 (e.g., all debits have a corresponding credit), or replies `VoteRollback` if there is 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.

Example of a recoverable 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.

Note: 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 of a 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

```

Failure models

The Transaction Service provides atomic outcomes for transactions in the presence of application, system or communication failures. This section describes the behaviour of application entities when failures occur.

From the viewpoint of each user object role, two types of failure are relevant: a failure affecting the object itself (local failure) and a failure external to the object (external failure), such as failure of another object or failure in the communication with that object.

Transaction originator

Local failure

A failure of a Transaction originator prior to the originator issuing commit will cause the transaction to be rolled back. A failure of the originator after issuing commit and before the outcome is reported may result in either commitment or rollback of the transaction depending

on timing; in this case completion of the transaction takes place without regard to the failure of the originator.

External failure

Any external failure affecting the transaction prior to the originator issuing commit will cause the transaction to be rolled back; the standard exception `TransactionRolledBack` will be raised in the originator when it issues commit.

A failure after commit and before the outcome has been reported will mean that the client may not be informed of the transaction outcome, depending on the nature of the failure, and the use of the `report_heuristics` option of `commit`. For example, the transaction outcome will not be reported to the client if communication between the client and the `Coordinator` fails.

A client may use `get_status` on the `Coordinator` to determine the transaction outcome. However, this is not reliable because the status `NoTransaction` is ambiguous: it could mean that the transaction committed and has been forgotten, or that the transaction rolled back and has been forgotten.

If an originator needs to know the transaction outcome, including in the case of external failures, then either the originator's implementation must include a `Resource` object so that it will participate in the two-phase commit procedure (and any recovery), or the originator and `Coordinator` must be located in the same failure domain (for example, the same execution environment).

Transactional server

Local failure

If the Transactional Server fails then optional checks by a Transaction Service implementation may cause the transaction to be rolled back; without such checks, whether the transaction rolls back depends on whether the commit decision has already been made (this would be the case where 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 will cause the transaction to be rolled back. If this 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 will be returned to a client issuing commit.

Recoverable server

Behaviour 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. This

protocol, including the local and external failure models and the required behaviour of the `Resource`, has been described previously.

Summary

In summary, when developing OTS applications using the raw OTS interfaces, the programmer must be concerned with:

- creating `Resource` and `SubtransactionAwareResource` objects for each object which will participate within the transaction/subtransaction. These resources are responsible for the persistence, concurrency control, and recovery for the object. The OTS will invoke these objects during the prepare/commit/abort phase of the (sub)transaction, and the `Resources` must then perform all appropriate work.
- registering `Resource` and `SubtransactionAwareResource` objects at the correct time within the transaction, and ensuring that the object is only registered once within a given transaction. As part of registration a `Resource` will receive a reference to a `RecoveryCoordinator` which must be made persistent so that recovery can occur in the event of a failure.
- ensuring that, in the case of nested transactions, any propagation of resources such as locks to parent transactions are correctly performed. Propagation of `SubtransactionAwareResource` objects to parents must also be managed.
- in the event of failures, the programmer or system administrator is responsible for driving the crash recovery for each `Resource` which was participating within the transaction.

The OTS does not provide any `Resource` implementations. These must be provided by the application programmer or the OTS implementer.

JBossTS interfaces for extending the OTS

Introducing

This chapter contains a description of the use of the *JBossTS* classes which provide extensions to the OTS interfaces. These advanced interfaces are all written on top of the basic OTS engine described previously and have been designed so that applications written using them will continue to operate on other OTS implementations, but without the added functionality.

These features can be summarized below:

- `AtomicTransaction` class, which provides a more manageable interface to the OTS transaction than `CosTransactions::Current`, and automatically keeps track of transaction scope. In addition, it allows the creation of *nested top-level transactions* in a more natural manner than that provided by the OTS.
- advanced subtransaction-`Resource` classes which allow nested transactions to use a two-phase commit protocol, giving all of the benefits of using nested transactions. These `Resources` can also be ordered within *JBossTS*, enabling the programmer to control when a `Resource` will be called during the commit/abort protocol with respect to other `Resources`.
- where available, *JBossTS* uses implicit context propagation between client and server. However, where this is not available an explicit interposition class is provided which simplifies the work required by the programmer to do interposition. The *JBossTS* API, *Transactional Objects for Java*, outlined in Chapter 2 requires either explicit or implicit interposition, even in a stand-alone mode when using a separate transaction manager. *Transactional Objects for Java* is fully described in the `TxCore` manuals.

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

Nested transactions

As was mentioned previously, the OTS implementation of nested transactions is extremely limited, and can lead to the generation of inconsistent results: a subtransaction coordinator discovers part way through committing that some resources cannot commit; however, it cannot 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, i.e., there are two phases, a prepare and a commit/abort. Using a multi-phase commit protocol avoids the above problem of discovering that some resources cannot commit whereas others have already been told to commit. The first (prepare) phase is used to generate consensus on the commit outcome, and the second phase is used to enforce this 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` then it will be driven by a two-phase commit protocol whenever a nested transaction commits:

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

Interface 4: `ArjunaSubtranAwareResource`.

During the first phase of the commit protocol the `prepare_subtransaction` method will be called, and the resource can behave as though it were being driven by a top-level transaction, i.e., it should make any state changes provisional upon the second phase of the protocol. Any changes to persistent state must still be provisional upon the commit/abort of the top-level transaction. Based upon the votes of all registered resources, *JBossTS* will then either call `commit_subtransaction` or `rollback_subtransaction`.

Note: this scheme can only work successfully if all resources registered within a given subtransaction are instances of the `ArjunaSubtranAwareResource`, and that once a resource has told the coordinator it can prepare it will not negate this decision.

Extended resources

When resources are registered with a transaction, the transaction maintains them within a list (the intentions list) so that at termination time it can drive each resource appropriately, i.e., to commit or abort. However, the application programmer has no control over the order in which resources will be called, or whether previously registered resources should be replaced with newly registered resources. The *JBossTS* interface `ArjunaOTS::OTSAbstractRecord` gives programmers this control.

```

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);
};

```

Interface 5: OTSAbstractRecord.

The attributes and methods will now be described:

- `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`: by returning `TRUE` from this method, the instance will be propagated to the parent transaction if the current transaction commits; `FALSE` means it will not be propagated.
- `saveRecord`: by returning `TRUE` from this method *JBossTS* will attempt to save sufficient information about the record to persistent store during commit to enable crash recovery mechanisms to replay the transaction termination in the event of a failure. If `FALSE` is returned then no information will be saved.
- `merge`: used when two records need to merge together.
- `alter`: used when a record should be altered.
- `shouldAdd`: returns true is the record should be added to the list, false if it should be discarded.
- `shouldMerge`: returns true is the two records should be merged into single record, false if it should be discarded.
- `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: if a record with the same `type` and `uid` has already been inserted, then the methods `shouldAdd` etc. will be invoked to determine whether this record should also be added. If no such match occurs, then the record will be inserted in the intentions list based on

the `type` field, and ordered according to the `uid`, i.e., all of the records with the same `type` will appear ordered in the intentions list.

Note: `OTSAbstractRecord` is derived from `ArjunaSubtranAwareResource`. Therefore, all instances of `OTSAbstractRecord` will also obtain the benefits of this interface.

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 terminated then it will automatically rollback the transaction. This is something which `CosTransactions::Current` cannot do. When building applications using *JBossTS*, programmers are encouraged to use `AtomicTransaction` for the added benefits it provides. It is located in the `com.arjuna.ats.jts.extensions.ArjunaOTS` package.

```
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 suppressed */

    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;
};
```

`AtomicTransaction` provides operations to start an action (`begin`); commit an action (`commit`); and abort an action (`rollback`). Transaction nesting is determined dynamically; that is, any transaction started within the scope of another running transaction is deemed to be nested.

The `TopLevelTransaction` class, which is derived from `AtomicTransaction`, allows the creation of nested top-level transactions. Such transactions allow non-serializable and potentially non-recoverable side effects to be initiated from within a transaction and should be used with caution. Nested top-level transactions can be created using 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.

Note: `AtomicTransaction` and `TopLevelTransaction` are completely compatible with `CosTransactions::Current`, i.e., the two transaction mechanisms can be used interchangeably within the same application/object.

`AtomicTransaction/TopLevelTransaction` are similar to `CosTransactions::Current`, with the intention of simplifying the interface between application programmer and the OTS. However, by using `AtomicTransaction/TopLevelTransaction` the programmer has the following advantages:

- 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, will become the thread's current transaction.
- instances of `AtomicTransaction` track scope and if such an instance goes out of scope before being terminated then it will be automatically aborted, aborting any children it may have.

Context propagation issues

When using Transactional Objects for Java in a distributed manner, *JBossTS* requires interposition to be used between client and object; this is also true if the application is local (i.e., client is co-located with object) but the transaction manager is remote. In the case of implicit context propagation, i.e., where the application object is derived from `CosTransactions::TransactionalObject`, then the application programmer need take no further action; *JBossTS* will automatically provide interposition. However, where implicit propagation is not supported by the *ORB*, or is not being used by the application, the programmer must take additional action to enable interposition.

The class `com.arjuna.ats.jts.ExplicitInterposition` is provided to allow an application to create a local control object which acts as a local coordinator fielding registration requests that would normally have been passed back to the originator. This surrogate registers itself with the original coordinator to enable it to correctly participate in the commit protocol. The application thread context is modified to become the surrogate transaction hierarchy; any transaction context currently associated with the thread will be lost. The interposition lasts for the lifetime of the explicit interposition object, at which point the application thread will no longer be associated with a transaction context, i.e., it will be set to null.

Note: interposition is intended only for those situations where the transactional object and the transaction occur within *different* processes, i.e., are not co-located. If the transaction is created locally to the client (invoker of the transactional object), i.e., is not managed by a separate transaction manager, then the explicit interposition class should not be used. The

transaction will be implicitly associated with the transactional object because it resides within the same process.

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

public void registerTransaction (Control control) throws
InterpositionFailed, SystemException;

public void unregisterTransaction () throws InvalidTransaction,
SystemException;

};
```

There are two ways in which a transaction context can be propagated between client and server: either as a reference to the client's transaction `Control`, or the client could send the transaction context explicitly. 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`:

```
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
}
```

Note: if the `Control` passed to the `register` operation of `ExplicitInterposition` is null, then no exception will be thrown. The system will simply assume 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 prior to this when it has finished 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 has terminated, *JBossTS* will assume the thread is still associated with the transaction and will issue a warning about trying to terminate a transaction while threads are still active within it.

Example

Introduction

The following example illustrates the concepts and the implementation details for a simple client/server example that uses implicit context propagation and indirect context management.

The basic example

It is relatively simplistic in that only a single unit of work is included within the scope of the transaction; consequently, a two phase commit is not required, but rather a one phase commit.

It demonstrates the invocation of the client and server processes using both the implicit propagation command line option, and also the interposition command line option.

The idl interface for this example is as follows. Line 1 includes the CosTransactions.idl.

For the purposes of this worked example, we have defined a single method (see line 12) for the DemoInterface interface. We will use this method in the DemoClient program.

```
1  #include <idl/CosTransactions.idl>
2  #pragma javaPackage ""
3
4
5  module Demo
6  {
7      exception DemoException {};
8
9      interface DemoInterface : CosTransactions::TransactionalObject
10     {
11         void work() raises (DemoException);
12     };
13 };
```

We shall now describe an example implementation of this interface.

Resource

Here, we have overridden 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 has been invoked.

As mentioned previously, only a single unit of work is included within the scope of the transaction; consequently, we should not expect the `prepare()` at line 6, or the `commit()` at line 19 to be invoked. However, we should expect the `commit_one_phase()` method at line 25 to be called.

```
1  import org.omg.CosTransactions.*;
2  import org.omg.CORBA .SystemException;
3
4  public class DemoResource extends org.omg.CosTransactions
   ._ResourceImplBase
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,
27     SystemException
28     {
29         System.out.println("commit_one_phase called");
30     }
31
32     public void forget() throws SystemException
33     {
34         System.out.println("forget called");
35     }
36 }
```

Transactional implementation

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

Line 14 returns the transactional context for the `Current` pseudo object. Once we have a `Control` object, we 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 is terminated, the resource will receive requests to commit or rollback the updates performed as part of the transaction.

```
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 ._DemoInterfaceImplBase
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             OAIInterface.objectIsReady(resource);
19             coordinator.register_resource(resource);
20
21         }
22         catch (Exception e)
23         {
24             throw new DemoException();
25         }
26     }
27 }
28 }
```

Server implementation

The first requirement is to initialise the ORB and the BOA. Lines 10 and 11 accomplish these tasks.

It is the servant class `DemoImplementation` that contains the implementation code for the `DemoInterface` interface. Furthermore, it is ultimately the responsibility of the servant to service a particular client request. Line 13 instantiates a servant object for the subsequent servicing of client requests.

Once a servant has been instantiated, we can connect the servant to the ORB, so that the ORB can recognize the invocations on it, and pass them to the correct servant. Line 15 performs this task.

Lines 17 through to 21 stringify the servant object to an IOR, and write to a temporary file. This IOR will be subsequently used to construct the object in the `DemoClient` program (see section 3.11.4 for further details).

If this stringification has been successful, line 23 will output a ‘sanity check’ message.

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

```

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             ORBInterface.initORB(args, null);
11             OAIInterface.initBOA();
12
13             DemoImplementation obj = new DemoImplementation();
14
15             OAIInterface.objectIsReady(obj);
16
17             String ref = ORBInterface.orb().object_to_string(obj);
18             BufferedWriter file =
19             new BufferedWriter(new
20             FileWriter("DemoObjReference.tmp"));
21             file.write(ref);
22             file.close();
23
24             System.out.println("Object reference written to file");
25
26             ORBInterface.run();
27         }
28         catch (Exception e)
29         {
30             System.err.println(e);
31         }
32     }

```

After the server has compiled, we can use the following command line options, as defined below, to start a server process. By specifying the usage of a filter on the command line, we will effectively override any contrary setting in the `TransactionService.properties` file.

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

Client implementation

Our client, like the server, requires us to firstly initialize the ORB and the BOA. Lines 14 and 15 accomplish these tasks.

Once a server process has been started, we can assume, in this simplistic example, that the server has stringified the servant object that will service our client requests, and saved the IOR for this object in a temporary file. Lines 17 through to 22 open this file, extract the IOR as a string and then close the file.

Once we have the IOR, we can reconstruct the servant object. Initially, this string to object conversion returns an instance of `Object` (see line 24). However, if we want to invoke a method on the servant object, it is necessary for us to narrow this instance to an instance of the `DemoInterface` interface (line 26).

Once we have a reference to this servant object, we can start a transaction (line 28), perform a unit of work (line 30) and commit the transaction (line 32).

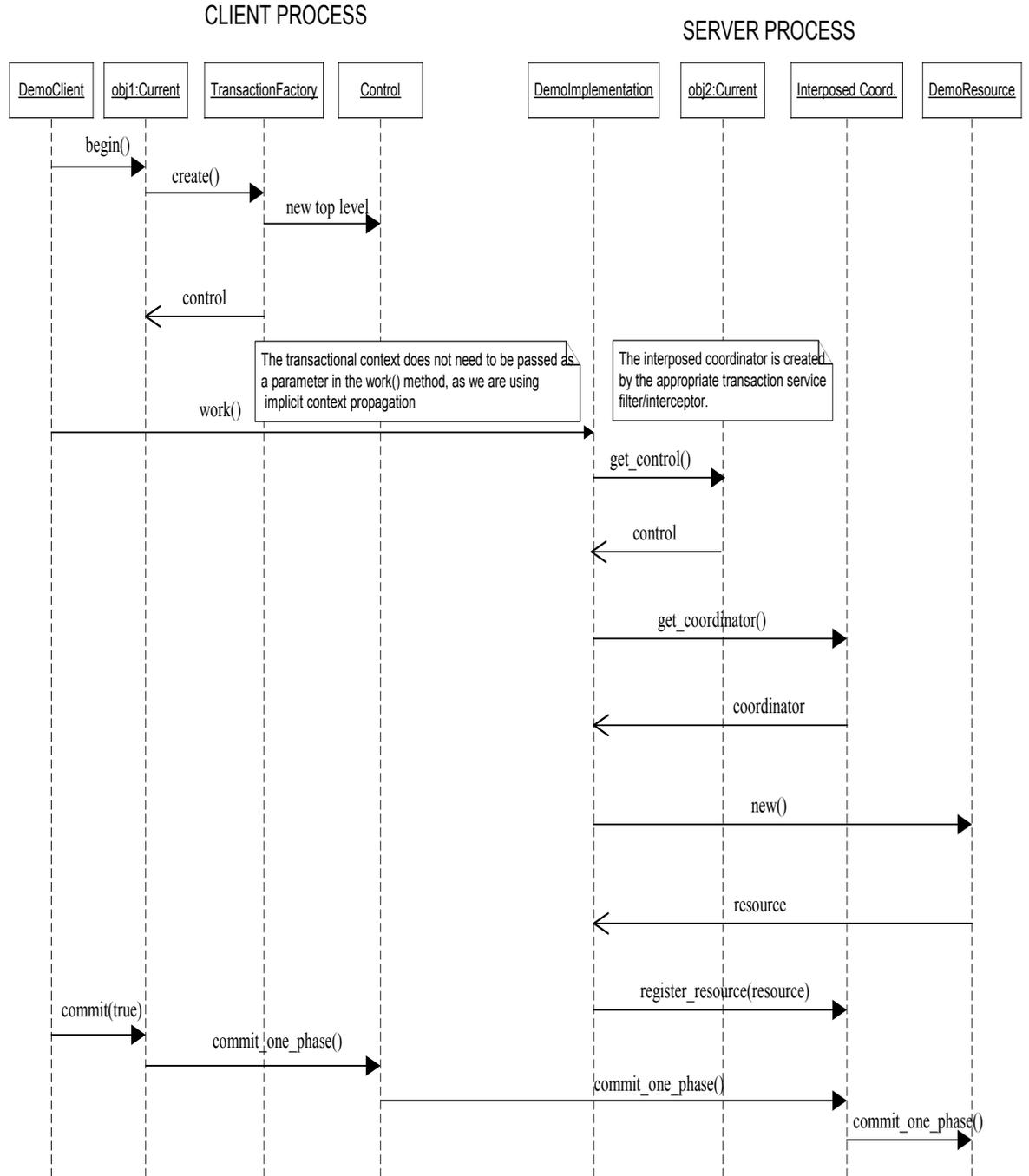
```
1  import Demo.*;
2  import java.io.*;
3  import com.arjuna.orbportability.*;
4  import com.arjuna.ats.jts.*;
5  import org.omg.CosTransactions.*;
6  import org.omg.*;
7
8  public class DemoClient
9  {
10     public static void main(String[] args)
11     {
12         try
13         {
14             ORBInterface.initORB(args, null);
15             OAIInterface.initBOA();
16
17             String ref = new String();
18             BufferedReader File =
19             new BufferedReader(new
20             FileReader("DemoObjReference.tmp"));
21             ref = file.readLine();
22             File.close();
23
24             org.omg.CORBA.Object obj =
25             ORBInterface.orb().string_to_object(ref);
26             DemoInterface d = (DemoInterface)
27             DemoInterfaceHelper.narrow(obj);
```

```
28         OTS.get_current().begin();
29
30         d.work();
31
32         OTS.get_current().commit(true);
33     }
34     catch (Exception e)
35     {
36         System.err.println(e);
37     }
38 }
39 }
```

Sequence diagram

The sequence diagram illustrates the method invocations that occur between the client and server. The following aspects are worthy of further discussion:

- The transactional context does not need to be explicitly passed as a parameter (as we are using implicit context propagation) in the `work()` method.
- Specifying the use of interposition when the client and server processes are started (by using appropriate filters/interceptors) creates an interposed coordinator that the servant process can utilize, negating any requirement for cross-process invocations. The interposed coordinator is automatically registered with the root coordinator at the client.
- The resource that is responsible for committing or rolling back modifications made to the transactional object is associated ('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.



Interpretation of output

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

In this simplistic example, our coordinator object has only a single registered resource. Consequently, it will perform 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 irrespective of whether the implicit context propagation option was used, or interposition. This is because interposition is essentially an aide to improve performance, where ordinarily a lot of marshalling between a client process, and potentially, a server process may be required.

The server output:

```
Object reference written to file

commit_one_phase called
```

Default settings

In this section we shall list some of the settings which *JBossTS* uses by default and how these settings can be overridden at run-time using property variables. These property variables can also be specified in the properties file which typically resides in `etc`.

- context propagation: unless a CORBA object is derived from `CosTransactions::TransactionalObject` then no context need be propagated. By default, to preserve distribution transparency *JBossTS* will *always* propagate a transaction context when calling remote objects, regardless of whether they are marked as transactional objects. This can be overridden 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 then *JBossTS* will transmit a null context and subsequent transactions begun by the object will be top-level. If a context is required then set the `com.arjuna.ats.jts.needTranContext` `YES`, and *JBossTS* will raise the `TransactionRequired` exception.
- *JBossTS* requires a persistent object store to record information about transactions in the event of failures. (If transactions complete successfully then this object store will have no entries). The default location for this must be set using the `com.arjuna.ats.arjuna.objectstore.objectStoreDir` variable in the properties file.
- if using a separate transaction manager for `Current` then its location is obtained from the `CosServices.cfg` file located in the `/etc` directory of the *JBossTS* distribution. If the file is not present then it will be created when the transaction manager is first started. To override the default name and location of the configuration file use the `com.arjuna.orbportability.initialReferencesFile` and `com.arjuna.orbportability.initialReferencesRoot` variables.
- checked transactions are not enabled by default, i.e., 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 `com.arjuna.ats.jts.checkedTransactions` to `YES`.

- if a value of 0 is specified for the timeout of a top-level transaction (or no timeout is specified), then *JBossTS* will not impose any timeout on the transaction, i.e., it will be allowed to run indefinitely. This default timeout can be overridden by setting the `com.arjuna.ats.jts.defaultTimeout` property variable to the required timeout value in *seconds*.

Chapter 7

Failure recovery

Introduction

The failure recovery subsystem of *JBossTS* will 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 machine hosting them crash or lose network connectivity. In the case of machine (system) crash or network failure, the recovery will not take place until the system or network are restored, but the original application does not need to be restarted – recovery responsibility is delegated to the Recovery Manager process (see below). Recovery after failure requires that information about the transaction and the resources involved survives the failure and is 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 that was in progress at the time of the failure may be inaccessible. For database resources, this may be reported as tables or rows held by “in-doubt transactions”. For TransactionalObjects for Java resources, an attempt to activate the Transactional Object (as when trying to get a lock) will fail.

Note: Because of limitations in the ORB which ships with the JDK 1.3, it is not possible to provide crash recovery. We therefore do not recommend using this ORB for mission critical applications.

Configuring the failure recovery subsystem for your ORB

The failure recovery subsystem of *JBossTS* is an area where complete ORB-independence cannot be achieved. However, the basic configuration of *JBossTS* for failure recovery is ORB-independent. The configuration applicable to applications using *JBossTS* is achieved using the *RecoveryManager-properties.xml* file, and the *orbportability-properties.xml*, which should contain, respectively, the entry:

```
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryActivator_1"
value="com.arjuna.ats.internal.jts.orbspecific.recovery.RecoveryEnablement"/>
```

and the entry:

```
<property
    name="com.arjuna.orbportability.orb.PostInit2"
```

```
value="com.arjuna.ats.internal.jts.recovery.RecoveryInit"/>
```

These entries cause the loading of instances of the named classes, which in turn load the ORB-specific classes needed and performs other initialisation. This enables failure recovery for transactions initiated by or involving applications using this property file. The default RecoveryManager-properties.xml file and the orportability-properties.xml with the distribution include these entries.

Note: Failure recovery is NOT supported with the JavaIDL ORB that is part of JDK. Failure recovery is supported in version 3.0 of JBossTS for Orbix 2000 for Java and JacORB.

With Orbix 2000 for Java the recovery subsystem requires the *IONA Location Daemon* to be running on any machine when an *JBossTS*-using application process starts running, or initiates or takes part in a transaction. The Location Daemon must also be running when the RecoveryManager is operating.

If the RecoveryEnablement line in the property file is removed (or commented out), no recovery will be enabled.

The Recovery Manager

The failure recovery subsystem of *JBossTS* requires that the stand-alone Recovery Manager process be running for each ObjectStore (typically one for each node on the network that is running *JBossTS* applications). The RecoveryManager file is located in the *arjuna.jar* file within the package `com.arjuna.ats.arjuna.recovery.RecoveryManager`. To start the Recovery Manager issue the following command:

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager
```

If the `-test` flag is used with the Recovery Manager then it will display a “Ready” message when initialised, i.e.,

```
java com.arjuna.ats.arjuna.recovery.RecoveryManager -test
```

Important Note

To allow successful registration of Resource objects, when the ORB is used is JacORB, the Recovery Manager shall be started.

Configuring the Recovery Manager

The RecoveryManager reads the properties defined in the `jbossjts-properties.xml` file and then also reads the property file `RecoveryManager-properties.xml`, from the same directory as it found the `arjuna` properties file. An entry for a property in the RecoveryManager properties file will override an entry for the same property in the main TransactionService properties file. Most of the entries are specific to the Recovery Manager.

A default version of `RecoveryManager-properties.xml` is supplied with the distribution – this can be used without modification, except possibly the debug tracing fields (see below, Output). The rest of this section discusses the issues relevant in setting the properties to other values (in the order of their appearance in the default version of the file)

Output

It is likely that installations will want to have some form of output from the `RecoveryManager`, to provide a record of what recovery activity has taken place. `RecoveryManager` uses the logging tracing mechanism provided by the Arjuna Common Logging Framework (CLF), which provides a high level interface that hides differences that exist between existing logging APIs such as Jakarta `log4j` or JDK 1.4 logging API.

With the CLF applications make logging calls on `commonLogger` objects. These `commonLogger` objects pass log messages to `Handler` for publication. Both `commonLoggers` and `Handlers` may use logging Levels to decide if they are interested in a particular log message. Each log message has an associated log Level that gives the importance and urgency of a log message. The set of possible Log Levels are `DEBUG`, `INFO`, `WARN`, `ERROR` and `FATAL`. Defined Levels are ordered according to their integer values as follows: `DEBUG < INFO < WARN < ERROR < FATAL`.

The CLF provides an extension so that an application may filter logging messages with a finer granularity. That is, when a log message is provided to the `commonLogger` with the `DEBUG` level, additional conditions can be specified to determine if the log message is enabled or not.

Note: These conditions are applied if and only the `DEBUG` level is enabled and the log request performed by the application specifies debugging granularity.

When enabled, Debugging is filtered conditionally on three variables:

- Debugging level: this is where the log request with the `DEBUG` Level is generated from, e.g., constructors or basic methods.
- Visibility level: the visibility of the constructor, method, etc. that generates the debugging.
- Facility code: for instance the package or sub-module within which debugging is generated, e.g., the object store.

The Common Logging Framework defines three interfaces for these variables. A particular product may implement its own classes defining its own finer granularity. `JBossTS` uses the default Debugging level and the default Visibility level provided by CLF, but it defines its own Facility Code. `ArjunaJTS` uses the default level assigned to its `commonLoggers` objects (`DEBUG`), as described below:

Debugging level – `JBossTS` uses the default values defined in the class `com.hp.mw.common.util.logging.CommonDebugLevel`

- `NO_DEBUGGING = 0x0000`: No diagnostics.
A `CommonLogger` object assigned with this values discard all debug requests
- `FULL_DEBUGGING = 0xffff`: Full diagnostics.
A `CommonLogger` object assigned with this value allows all debug requests if the facility code and the visibility level match those allowed by the `commonLogger`.

Additional Debugging Values are:

- `CONSTRUCTORS = 0x0001`: Diagnostics from constructors.
- `DESTRUCTORS = 0x0002`: Diagnostics from finalizers.
- `CONSTRUCT_AND_DESTRUCT = CONSTRUCTORS | DESTRUCTORS`:
Diagnostics from constructors and finalizers.
- `FUNCTIONS = 0x0010`: Diagnostics from functions.
- `OPERATORS = 0x0020`: Diagnostics from operators, such as equals.
- `FUNCS_AND_OPS = FUNCTIONS | OPERATORS`: Diagnostics from functions and operations.
- `ALL_NON_TRIVIAL: CONSTRUCT_AND_DESTRUCT | FUNCTIONS | OPERATORS` Diagnostics from all non-trivial operations.
- `TRIVIAL_FUNCS = 0x0100`: Diagnostics from trivial functions.
- `TRIVIAL_OPERATORS = 0x0200`: Diagnostics from trivial operations, and operators.
- `ALL_TRIVIAL: TRIVIAL_FUNCS | TRIVIAL_OPERATORS`: Diagnostics from all trivial operations.
- `ERROR_MESSAGES = 0x400`: only output from debugging error/warning messages

Visibility level – JBossTS uses the default values defined in the class `com.hp.mw.common.util.logging.CommonVisibilityLevel`

- `VIS_NONE = 0x0000`: No Diagnostic
- `VIS_PRIVATE = 0x0001`: only from private methods.
- `VIS_PROTECTED = 0x0002` only from protected methods.
- `VIS_PUBLIC = 0x0004` only from public methods.
- `VIS_PACKAGE = 0x0008` only from package methods.
- `VIS_ALL = 0xffff`: Full Diagnostic

Facility Code – JBossTS uses the following values

- `FAC_ATOMIC_ACTION = 0x0000001` (atomic action core module).
- `FAC_BUFFER_MAN = 0x0000004` (state management (buffer) classes).
- `FAC_ABSTRACT_REC = 0x0000008` (abstract records).
- `FAC_OBJECT_STORE = 0x0000010` (object store implementations).
- `FAC_STATE_MAN = 0x0000020` (state management and `StateManager`).
- `FAC_SHMEM = 0x0000040` (shared memory implementation classes).
- `FAC_GENERAL = 0x0000080` (general classes).
- `FAC_CRASH_RECOVERY = 0x0000800` (detailed trace of crash recovery module and classes).

- FAC_THREADING = 0x00002000 (threading classes).
- FAC_JDBC = 0x00008000 (JDBC 1.0 and 2.0 support).
- FAC_RECOVERY_NORMAL = 0x00040000 (normal output for crash recovery manager).

To ensure appropriate output, it is necessary to set some of the finer debug properties explicitly in the properties file, as described below, where default values are given.

```
<properties>

  <!-- CLF 2.0 properties -->

  <property

    name="com.arjuna.common.util.logging.DebugLevel"

    value="0x00000000"/>

  <property

    name="com.arjuna.common.util.logging.FacilityLevel"

    value="0xffffffff"/>

  <property

    name="com.arjuna.common.util.logging.VisibilityLevel"

    value="0xffffffff"/>

  <property

    name="com.arjuna.common.util.logger"

    value="log4j"/>

</properties>
```

If particular debug messages are required, the finer debug properties can be set to the appropriate level described earlier. For instance, to enable all messages related to recovery after a crash or CRASH RECOVERY but only those raised within functions (see Facility level), and public (see Visibility level) the finer debugging could be set as followz:

```
<properties>

  <!-- CLF 2.0 properties -->

  <property

    name="com.arjuna.common.util.logging.DebugLevel"

    value="0x0010"/>
```

```

<property
    name="com.arjuna.common.util.logging.FacilityLevel"
    value="0x00000800"/>

<property
    name="com.arjuna.common.util.logging.VisibilityLevel"
    value="0x0004"/>

<property
    name="com.arjuna.common.util.logger"
    value="log4j"/>

</properties>

```

A user could have an interest in other messages, as well as Crash Recovery. For instance the following combination logs Crash Recovery messages and atomic action messages produced by the TxCore module

```

<properties>

...

<property
    name="com.arjuna.common.util.logging.FacilityLevel"
    value="0x00000801"/>

...

</properties>

```

Note: Finer debug messages are enabled only if the logging level is set to a DEBUG value as defined by the underlying logging configuration (see below example for “log4j”)

The choice of the underlying logging infrastructure is defined by the property “com.arjuna.common.util.logger”, which is set by default to “log4j”. Possible values are described below and more details could be found in the *CLF2.0 Programmer’s Guide*.

Property Value	Description
log4j	Log4j logging (log4j classes must be available in the classpath); configuration through the log4j.properties file, which is picked up from the CLASSPATH or given through a System property: log4j.configuration
jdk14	JDK 1.4 logging API (only supported on JVMs of version 1.4 or higher). Configuration is done through a file logging.properties in the jre/lib directory.
simple	Selects the simple JDK 1.1 compatible console-based logger provided by Jakarta Commons Logging
Csf	Selects CSF-based logging (CSF embeddor must be available)
dotnet	Selects a .net logging implementation Since a dotnet logger is not currently implemented, this is currently identical to simple. Simple is a purely JDK1.1 console-based log implementation.
avalon	Uses the Avalon Logkit implementation
noop	Disables all logging
jakarta	Uses the default log system discovery mechanism of the Jakarta Commons Logging framework

The properties of the underlying log system are configured in a manner specific to that log system, e.g., a log4j.properties file in the case that log4j logging is used. The default configuration file for log4j used by JBossTS is:

```
# Default LOG4J Configuration

# Arjuna Technologies Ltd.

# $Id: log4j.properties,v 1.1 2003/09/28 11:38:24 rbegg Exp $

log4j.rootLogger=WARN, stdout

log4j.appender.stdout=org.apache.log4j.ConsoleAppender

log4j.appender.stdout.layout=org.apache.log4j.PatternLayout

log4j.appender.stdout.layout.ConversionPattern=%d [%t] %-5p %c - %m%n
```

To set the logging level to the DEBUG level in order to log finer debug messages, the log4j.rootLogger should have a value as follows:

```
log4j.rootLogger=DEBUG, ...
```

Messages describing the start and the periodical behavior made by the RecoveryManager are set the INFO level. To see them, the logging level should be set to the INFO level, as described above for DEBUG. Setting the normal recovery messages to the INFO level allows the RecoveryManager to produce a moderate level of reporting. If there is no work to be done, it just reports the entry into each module for each periodic pass. To disable the INFO

messages produced by the Recovery Manager, the logging level can be set to the value above (ERROR).

Note: The Logging level should be set at least to the WARN value to enable the report of warning messages.

Periodic Recovery

The RecoveryManager scans the ObjectStore and other locations of information, looking for transactions and resources that require, or may require recovery. The scans and recovery processing are performed by recovery modules, (instances of classes that implement the `com.arjuna.ats.arjuna.recovery.RecoveryModule` interface), each with responsibility for a particular category of transaction or resource. The set of recovery modules used are dynamically loaded, using properties found in the RecoveryManager property file.

The interface has two methods: `periodicWorkFirstPass` and `periodicWorkSecondPass`. At an interval (defined by property `PERIODIC_RECOVERY_PERIOD`), the RecoveryManager will call the first pass method on each property, then wait for a brief period (defined by `RECOVERY_BACKOFF_PERIOD`), then call the second pass of each module. Typically, in the first pass, the module scans (e.g. the relevant part of the ObjectStore) to find transactions or resources that are in-doubt (i.e. are part way through the commitment process). On the second pass, if any of the same items are still in-doubt, it is possible the original application process has crashed and the item is a candidate for recovery.

An attempt, by the RecoveryManager, to recover a transaction that is still progressing in the original process(es) is likely to break the consistency. Accordingly, the recovery modules use a mechanism (implemented in the `com.arjuna.ats.internal.jts.recovery.contact` package) to check to see if the original process is still alive, and if the transaction is still in progress. The RecoveryManager only proceeds with recovery if the original process has gone, or, if still alive, the transaction is completed. (If a server process or machine crashes, but the transaction-initiating process survives, the transaction will complete, usually generating a warning. Recovery of such a transaction is the RecoveryManager's responsibility).

It is clearly important to set the interval periods appropriately. The total iteration time will be the sum of `PERIODIC_RECOVERY_PERIOD`, `RECOVERY_BACKOFF_PERIOD` and the length of time it takes to scan the stores and to attempt recovery of any in-doubt transactions found, for all the recovery modules. The recovery attempt time may include connection timeouts while trying to communicate (via the ORB) with processes or machines that have crashed or are inaccessible (which is why there are mechanisms in the recovery system to avoid trying to recover the same transaction for ever). The total iteration time will affect how long a resource will remain inaccessible after a failure – `PERIODIC_RECOVERY_PERIOD` should be set accordingly (default is 120 seconds). The `RECOVERY_BACKOFF_PERIOD` can be comparatively short (default is 10 seconds) – its purpose is mainly to reduce the number of transactions that are candidates for recovery and which thus require a “contact” call to the original process to see if they are still in progress (note: in JBossTS 2.0, there was no contact mechanism, and the backoff period had to be long enough to avoid catching transactions in flight at all. From 2.1, there is no such risk)

Several recovery modules (implementations of the `com.arjuna.ats.arjuna.recovery.RecoveryModule` interface) are supplied with *JBossTS*, supporting various aspects of transaction recovery including JDBC recovery. It is possible for advanced users to create their own recovery modules and register them with the Recovery Manager. The recovery modules are registered with the `RecoveryManager` using properties (in `RecoveryManager-properties.xml`) that begin with “`RecoveryExtension`”. These will be invoked on each pass of the periodic recovery in the sort-order of the property names – it is thus possible to predict the ordering (but note that a failure in an application process might occur while a periodic recovery pass is in progress). The default `RecoveryExtension` settings are:

```
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryExtension1"
    value="com.arjuna.ats.internal.arjuna.
        recovery.AtomicActionRecoveryModule"/>
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryExtension2"
    value="com.arjuna.ats.internal.txoj.recovery.TORecoveryModule"/>
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryExtension3"
    value="com.arjuna.ats.internal.jts.recovery.transactions.
        TopLevelTransactionRecoveryModule"/>
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryExtension4"
    value="com.arjuna.ats.internal.jts.recovery.transactions.
        ServerTransactionRecoveryModule"/>
<property
    name="com.arjuna.ats.arjuna.recovery.recoveryExtension5"
    value="com.arjuna.ats.internal.jta.recovery.arjunacore.
        XARRecoveryModule"/>
```

XA resource recovery

Recovery of XA resources (databases etc.) accessed via JDBC is handled by the XARecoveryModule. This has two aspects: “transaction-initiated” and “resource-initiated” recovery. Transaction-initiated recovery is possible where the particular transaction branch had progressed far enough for a JTA_ResourceRecord to be written in the ObjectStore. The record contains the information needed to link the transaction, as known to the rest of JBossTS to the database. Resource-initiated recovery is necessary for branches where a failure occurred after the database had made a persistent record of the transaction, but before the JTA_ResourceRecord was persisted. Resource-initiated recovery is also necessary for datasources for which it is not possible to hold information in the JTA_ResourceRecord that allows the recreation in the RecoveryManager of the XAConnection/XAResource that was used in the original application.

Transaction-initiated recovery is automatic. The XARecoveryModule finds the JTA_ResourceRecord that need recovery (using the two-pass mechanism described above), then uses the normal recovery mechanisms to find the status of the transaction it was involved in (i.e. it calls `replay_completion` on the RecoveryCoordinator for the transaction branch), (re)creates the appropriate XAResource and issues commit or rollback on it as appropriate. The XAResource creation will use the same information, database name, username, password etc., as the original application.

Resource-initiated recovery has to be specifically configured, by supplying the RecoveryManager with the appropriate information for it to interrogate all the databases (XADatasources) that have been accessed by any JBossTS application. The access to each XADatasource is handled by a class that implements the `com.arjuna.ats.jta.recovery.XAConnectionRecovery` interface. Instances of this are dynamically loaded, as controlled by properties with names beginning “XAConnectionRecovery”.

The XARecoveryModule will use the XAConnectionRecovery implementation to get an XAResource to the target datasource. On each invocation of `periodicWorkSecondPass`, the recovery module will issue an `XAResource.recover` request – this will (as described in the XA specification) return a list of the transaction identifiers (Xid’s) that are known to the datasource and are in an indeterminate (in-doubt) state. The list of these in-doubt Xid’s received on successive passes (i.e. `periodicWorkSecondPass`-es) is compared. Any Xid that appears in both lists, and for which no JTA_ResourceRecord was found by the intervening transaction-initiated recovery is assumed to belong to a transaction that was 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 XAConnectionRecovery implementation class can be written to contain all the information needed to perform recovery to some datasource. Alternatively, a single class can handle multiple datasources (with some similar features, presumably). The constructor of the implementation class must have an empty parameter list (because it is loaded dynamically),

but 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 use made of this string is determined by the `XAConnectionRecovery` implementation class.

An `XAConnectionRecovery` implementation class, `com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery` is provided to support resource-initiated recovery for any `XADataSource` (see *JBossTS JTA* manual). For this class, the string received in `initialise` is treated as containing 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 url needed to access the database (sometimes called the database name, though this may be only a component of the 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;OraRecoveryInfo
```

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

```
/*
 * 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(breakPosition +1));
        }
        catch (NumberFormatException e)
        {
            //Produce a Warning Message
            return false;
        }
    }

    PropertyManager.addPropertyFile(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
}

```

Caution: Oracle usernames for Oracle 8.0 to 8.1.4: it is necessary for *any* database user that will use distributed transactions (e.g., *JBossTS* and JDBC) to have select privilege on the SYS via DBA_PENDING_TRANSACTIONS. For 8.1.5 and higher, this is not (apparently) necessary for normal transaction access. However, this privilege is needed for the database user given when creating an XAConnection that provides an XAResource that is then used for XAResource.recover. (XAResource.commit, rollback etc. do not require the privilege). Accordingly, administrators may wish to create a special database username for the *JBossTS* RecoveryManager, which has this privilege, which need not be granted to users in general. An implication of this is that access to the RecoveryManager_2_2.properties file needs to be appropriately controlled, if the password for the RecoveryManager user is contained in it.

Note: *Note:* 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 (see below) are used with a single datasource, resource-initiated recovery will “see” transactions from other domains. Since it will not have a JTA_ResourceRecord available, it will rollback the transaction in the database, if the Xid appears in successive recover calls. Administrators may wish to suppress resource-initiated recovery (by not supplying an XAConnectionRecovery property) in such cases, or confine it to one (current) recovery domain.

Recovery behaviour

A property, `OTS_ISSUE_RECOVERY_ROLLBACK`, is provided to control whether the `RecoveryManager` explicitly issues a rollback request when asked (by `replay_completion`) for the status of a transaction that is unknown. According to the presume-abort mechanism used by `OTS` and `JTS`, it can be inferred that the transaction has rolledback, and this is the response that will be returned to the Resource (including a subordinate coordinator) in this case. The Resource can (and should) then apply that result to the underlying resources. However, it is also legitimate for the superior to issue a rollback: this is done if `OTS_ISSUE_RECOVERY_ROLLBACK=YES`.

A property of the `OTS` transaction identification mechanism is that it is 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 has crashed before receiving the commit or rollback request from the coordinator
- the Resource received the commit or rollback, and responded but the message was lost (or the coordinator process crashed).

In the first case, the `RecoveryManager` for the Resource `ObjectStore` will (eventually) reconstruct a new Resource (with a different IOR (`CORBA` object reference)) and issue a `replay_completion` request containing the new Resource IOR – the `RecoveryManager` for the coordinator will swap this in place of the original, useless one, and issue commit to the new (reconstructed) Resource (it must be commit, or there would be no transaction intention list to worry about). Until the `replay_completion` is received, the `RecoveryManager` will try to send commit to the Resource reference it had – this will fail (with a `CORBA System Exception` – exactly which one depends on the orb and other details).

In the second case, the Resource will never exist again. The `RecoveryManager` at the coordinator will never get through, and will receive `System Exceptions` forever.

It is important to realise that the `RecoveryManager` cannot distinguish these two cases by any protocol mechanism.⁴ 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 (real) resources inaccessible for longer.

To avoid this, the `RecoveryManager` will only attempt to send commit to a Resource a limited number of times. After that, it will consider the transaction “assumed complete”. It will retain 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

⁴ With some ORB environments, it is possible to avoid this situation – the reconstructed Resource can re-appear in response to a request targeted on the original IOR, and, as importantly, the ORB will not generate `OBJECT_NOT_EXIST` system exception if the Resource ever will re-appear. This behaviour cannot be guaranteed for all ORBs however.

RecoveryManager will activate the transaction and issue the commit request (to the new Resource IOR). The number of times the RecoveryManager will attempt to issue commit (at its own initiative, as part of the periodic recovery) is controlled by the property `COMMITTED_TRANSACTION_RETRY_LIMIT` (default is 3 times).

Expired entry removal

The operation of the recovery subsystem will cause some entries to be made in the ObjectStore that will not be 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 name as the value of a property whose name begins with “ExpiryScanner”. The RecoveryManager calls the `scan()` method on each loaded ExpiryScanner implementation at an interval determined by the property “`EXPIRY_SCAN_INTERVAL`”. This value is given in *hours* – default is 12. An `EXPIRY_SCAN_INTERVAL` value of zero will suppress 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 of these is created by every application process that uses JBossTS – they contain the information that allows the RecoveryManager 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 `FACTORY_CONTACT_EXPIRY_TIME` (in hours – default is 12, zero means never expire). The expiry time should be greater than the lifetime of any single *JBossTS*-using process.
- Assumed complete transactions : see above for detailed explanation. The expiry time is counted from when they were assumed to be complete (but a received `replay_completion` request will reset the clock). The risk with removing assumed complete transactions is that a prolonged communication outage will mean 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 rolledback, 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` (in hours, default is 240, zero means never expire).

The ExpiryScanner properties for these are:

```
<property
    name="com.arjuna.ats.arjuna.recovery.
        expiryScannerTransactionStatusManager"
    value="com.arjuna.ats.internal.arjuna.recovery.
```

```

        ExpiredTransactionStatusManagerScanner"/>
<property
    name="com.arjuna.ats.arjuna.recovery.
        expiryScannerContact"
    value="com.arjuna.ats.internal.jts.recovery.contact.
        ExpiredContactScanner"/>
<property
    name="com.arjuna.ats.arjuna.recovery.
        expiryScannerTopLevelTran"
    value="com.arjuna.ats.internal.jts.recovery.transactions.
        ExpiredTopLevelScanner"/>
<property
    name="com.arjuna.ats.arjuna.recovery.
        expiryScannerServerTran"
    value="com.arjuna.ats.internal.jts.recovery.transactions.
        ExpiredServerScanner"/>

```

There are two ExpiryScanners for the assumed complete transactions as there are different types in the ActionStore.

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. There are three paths by which information passes between the application process and the RecoveryManager:

- RecoveryCoordinator object references (IORs) are created in the application process, containing information identifying the transaction in the object key, passed to the Resource objects and then the object key is received by RecoveryManager
- the application process and the RecoveryManager access the same jbossts-properties.xml file and thus the same ObjectStore
- direct CORBA invocations from the RecoveryCoordinator to the application process, using information in the contact items (which are kept in the ObjectStore).

Making the RecoveryManager deal with `replay_completion` requests using an IOR created in the application process requires the assistance of the LocationDaemon (for Orbix 2000). The mechanism uses a string (POA name for Orbix 2000) to locate the process (the RecoveryManager) that is to receive the request.

Multiple recovery domains may be of use in migration scenarios, where separate ObjectStores are useful. However, multiple RecoveryManagers can cause problems with XA datasources if resource-initiated recovery is active on any of them.

Transaction statuses and replay_completion

When a transaction successfully commits (i.e., informs all registered Resources that it has committed) the transaction log is removed from the system. This is because the log is no longer required: all registered Resources have responded successfully to the prepare/commit invocation sequence. However, as a result if a Resource calls `replay_completion` on the RecoveryCoordinator after the transaction it represents has committed, the status returned will be `StatusRolledback`. The transaction system does not keep a record of committed transactions, and as such it assumes that the absence of a transaction log means that the transaction must have rolled back (in line with the presumed abort protocol used by the OTS).

JTA and the JTS

Distributed JTA

The JBossTS manuals describe how to use the JTA interfaces for purely local transactions. This is a high-performance implementation, but can only be used to execute transactions within the same process. If distributed transaction support is required, then it is necessary for the JTA to use the JTS. This also has the added advantage of providing interoperability with other JTS compliant transaction systems.

Note: if using the JTS and JTA interfaces to manage the same transactions, it is important that the JTA is configured to be aware of the JTS. Otherwise, local transactions will be created which are unaware of their JTS counterparts.

This configuration of the JTA to be aware of the JTS is explicit and at the user's control because it is possible that some applications may be using JBossTS in a purely local manner or may want to differentiate between JTS and JTA managed transactions.

To make the JTA interfaces JTS-aware, it is necessary to set the following property values:

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

Chapter 9

Tools

Introduction

This chapter explains how to start and use the tools framework and what tools are available.

Starting the Transaction Service tools

The way to start the transaction service tools differs on the operating system being used:

Windows:

Double click on the 'Start Tools' link in the JBoss Transaction Service program group in the start menu.

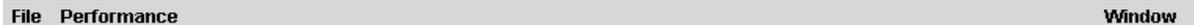
UNIX:

Start a bash shell and type:

```
cd < JBossTS INSTALL DIRECTORY >

./run-tools.sh
```

Once you have done this the tools window will appear. This is the launch area for all of the tools shipped with the JBoss Transaction Service. At the top of the window you will notice a menu bar (see Figure 20 - Menu bar).



The image shows a horizontal menu bar with a light gray background. On the left side, the text 'File Performance' is displayed in a bold, black font. On the right side, the text 'Window' is displayed in a bold, black font.

Figure 20 - Menu bar

This menu bar has four menus:

The **F**ile menu:



Open JMX Browser – this displays the JMX browser window (see **Using the JMX Browser** for more information on how to use the JMX browser).

Open Object Store Browser – this displays the JBossTS Object Store browser window (see **Using the Object Store Browser** for more information on how to use the Object Store browser).

Settings – this option opens the settings dialog which lets you configure the different tools available.

Exit – this closes the tools window and exits the application, any unsaved/unconfirmed changes will be lost.

The **Performance** menu:



Open – this opens a performance window – see the section named ‘Using the Performance Tool’ for more information on the performance tool.

Close All – this closes all of the currently open performance windows – see the section named ‘Using the Performance Tool’ for more information on the performance tool.

The **Window** menu:



Cascade Windows – this arranges the windows in a diagonal line to you find a specific window.

1. xxxxxx – For each window currently visible an extra menu option will be available here. Selecting this menu option will bring the associated window to the front of the desktop.

The **Help** menu:



About – this displays the about window which displays the product information.

Using the Performance Tool

The performance tool can be used to display performance information about the transaction service. This information is gathered using the Performance JMX bean which means that the transaction service needs to be integrated into an Application Server to give any performance information.

The performance information is displayed via a multi-series graph. To view this graph simply open a performance window by selecting Performance > Open (see Figure 21 - Performance window).

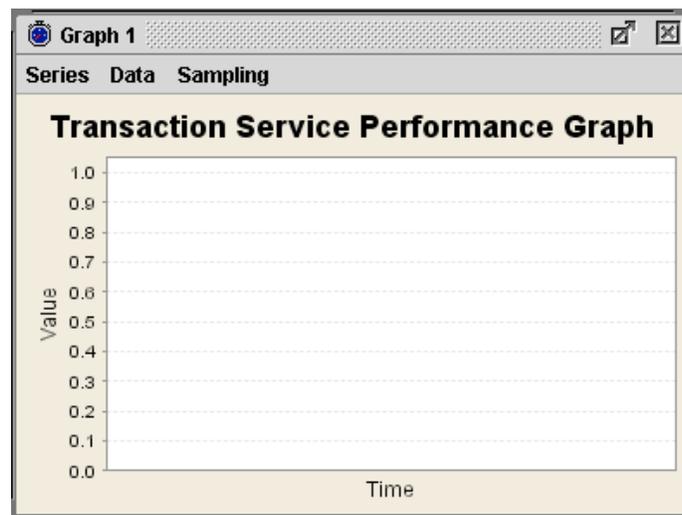


Figure 21 - Performance window

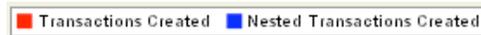
This window contains a multi-series graph which can display the following information:

- Number of transactions.
- Number of committed transactions.
- Number of aborted transactions.
- Number of nested transactions.
- Number of heuristics raised.

To turn these series on and off simply select the menu option from the series menu:



When series are turned on they appear in the legend at the bottom of the graph. The colour next to the series name (e.g. Transactions Created) is the colour of the line representing that data.



The data shown is graphed against time. The Y-axis represents the number of transactions and the X-axis represents time.

At any point the sampling of data can be stopped and restarted using the ‘Sampling’ menu and the data currently visible in the graph can be saved to a Comma Separate Values (CSV) file for importing the data into a spreadsheet application using the ‘Save to .csv’ menu option from the ‘Data’ menu.

Using the JMX Browser

To open the JMX browser window click on the **File** menu and then click the **Open JMX Browser** option. The JMX browser window will then be displayed (see **Figure 22 - JMX Browser window**).

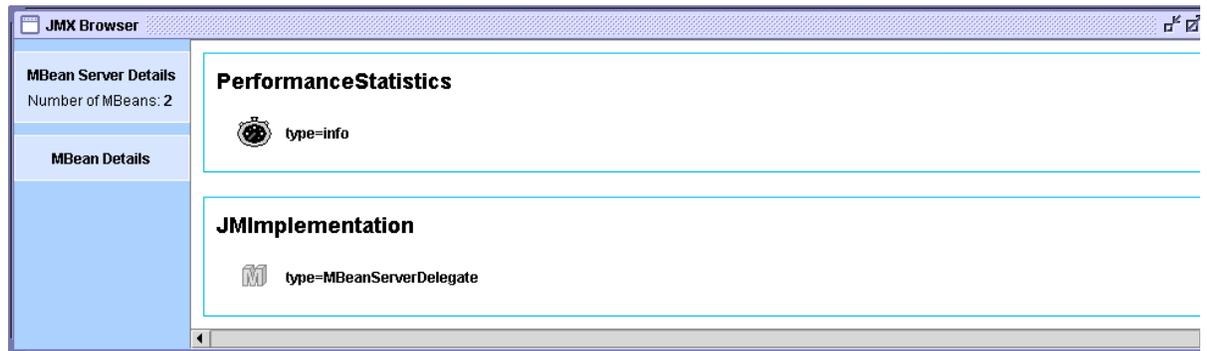


Figure 22 - JMX Browser window.

The window is made up of two main sections: the details panel and the MBean panel. The MBean panel displays the MBeans exposed by the MBean server. These are grouped by domain name. The details panel displays information about the currently selected MBean. To select an MBean just left-click it with the mouse and it will become highlighted. The information displayed in the details panel is as follows (see **Figure 23 - An example of what the details panel displays** for an example):

- The total number of MBeans registered on this server,

- The number of constructors exposed by this MBean,
- The number of attributes exposed by this MBean,
- The number of operations exposed by this MBean,
- The number of notifications exposed by this MBean,
- A brief description of the MBean.

There is also a **View** link which when clicked displays the attributes and operations exposed by this MBean. From there you can view readable attributes, alter writeable attributes and invoke operations.

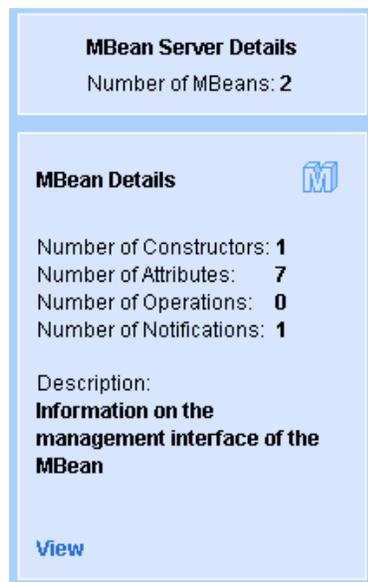


Figure 23 - An example of what the details panel displays.

Using Attributes and Operations

When the **View** link is clicked the View JMX Attributes and Operations window is displayed (see **Figure 24 - View JMX Attributes and Operations window**). From here you can view all readable attributes exposed by the selected MBean. You can also alter writeable attributes. If an attribute is read-only then you will not be able to alter an attributes value. To alter an attributes value just double click on the current value and enter the new value. If the button is enabled then you can click this to view a more suitable editing method. If the attribute type is a JMX object name then clicking this button will display the JMX attributes and operations for that object.

At any point you can click the button to refresh the attribute values. If an exception occurs while retrieving the value of an attribute the exception will be displayed in place of the attributes value.

You can also invoke operations upon an MBean. A list of operations exposed by an MBean is displayed below the attributes list. To invoke an operation simply select it from the list and click the  button. If the operation requires parameters a further window will be displayed, from this window you must specify values for each of the parameters required (see **Figure 25 - Invoke Operation Parameters**). You specify parameter values in the same way as you specify JMX attribute values. Once you have specified a value for each of the parameters click the **Invoke** button to perform the invocation.

Once the method invocation has completed its return value will be displayed.

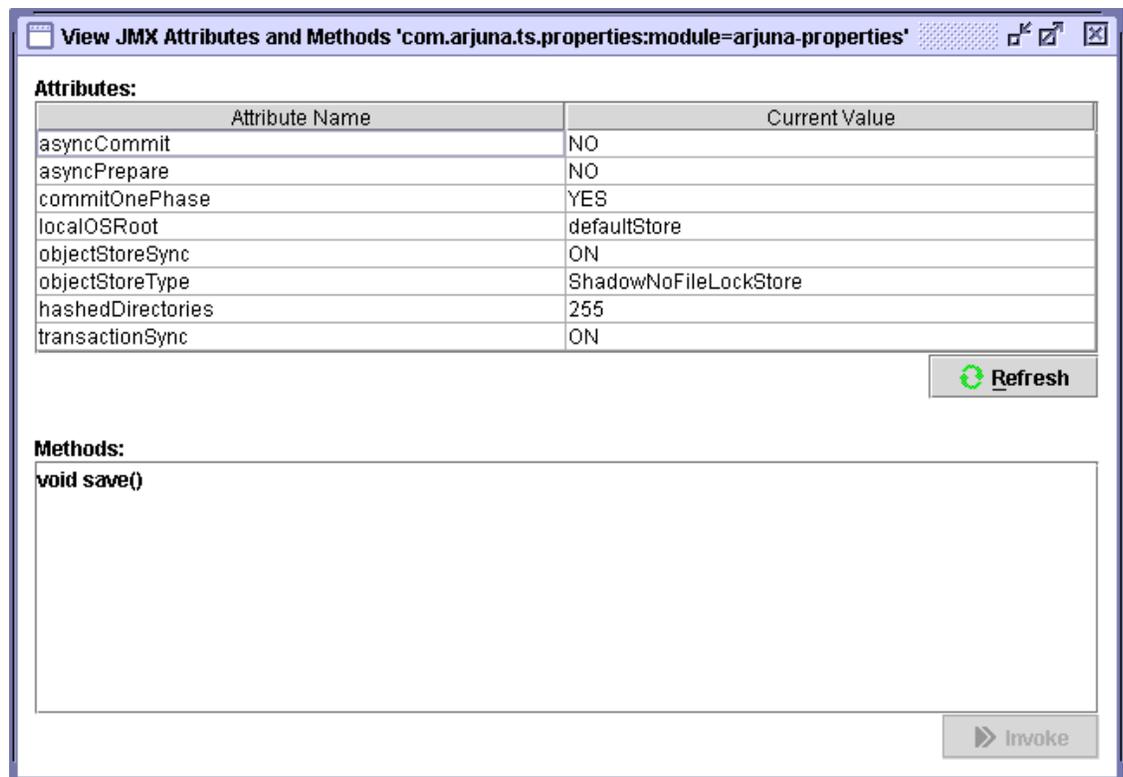


Figure 24 - View JMX Attributes and Operations window.



Figure 25 - Invoke Operation Parameters.

Using the Object Store Browser

To open the Object Store browser window click on the **File** menu and then click the **Open Object Store Browser** option. The Object Store browser window will then be displayed (see **Figure 26 - Object Store Browser window**).

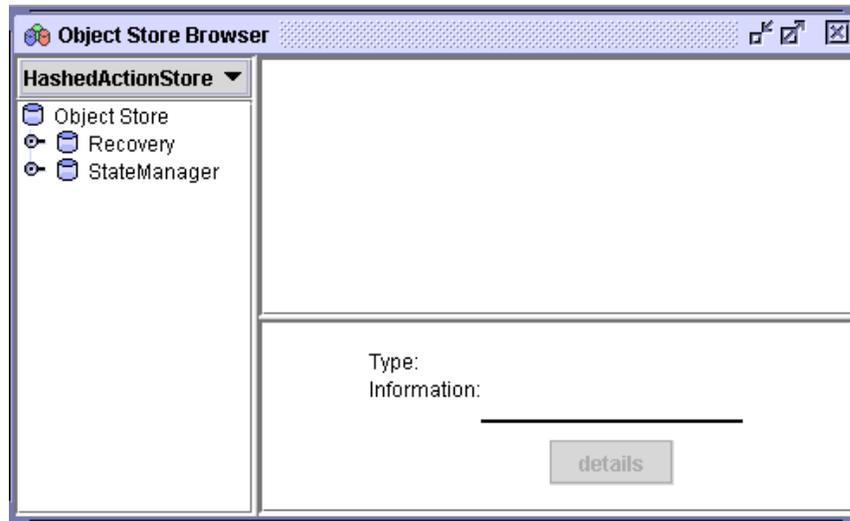


Figure 26 - Object Store Browser window

The object store browser window is split into four sections:

Object Store Roots	Objects
Object Store Hierarchy	Object Details

- Object Store Roots – this is a pull down of the currently available object store roots. Selecting an option from the list will repopulate the hierarchy view with the contents of the selected root.
- Object Store Hierarchy – this is a tree which shows the current object store hierarchy. Selecting a node from this tree will display the objects stored in that location.

- Objects – this is a list of icons which represent the objects stored in the selected location.
- Object Details – this shows information about the currently selected object (only if the object’s type is known to the state viewer repository see **Writing an OSV** for information on how to write a object state viewers).

Object State Viewers (OSV)

When an object is selected in the objects pane of the main window the registered Object State Viewer (or OSV) for that object type is invoked. An OSV’s job is to make information available via the user interface to the user to show information about the selected object. Distributed with the standard tools is an OSV for Atomic Actions, the OSV displays information on the Abstract Records in it’s various lists (e.g. heuristic, failed, read-only, etc). It is also possible to write your own OSVs which can be used to display information about object types you have defined. This subject is covered next.

Writing an OSV

Writing an OSV plugin allows you to extend the capabilities of the Object Store browser to show the state of user defined abstract records. An OSV plug-in is simply a class which implements the interface:

```
com.arjuna.ats.tools.objectstorebrowser.stateviewers.StateViewerInterface
```

It must be packaged in a JAR within the plugins directory. This example shows how to create an OSV plugin for an abstract record subclass which looks as follows:

```
public class SimpleRecord extends AbstractRecord
{
    private int _value = 0;

    .....

    public void increase()
    {
        _value++;
    }

    public int get ()
```

```

{
    return _value;
}

public String type()
{
    return "/StateManager/AbstractRecord/SimpleRecord";
}

public boolean restore_state(InputObjectState os, int i)
{
    boolean returnValue = true;

    try
    {
        _value = os.unpackInt();
    }

    catch (java.io.IOException e)
    {
        returnValue = false;
    }

    return returnValue;
}

public boolean save_state(OutputObjectState os, int i)
{

```

```
        boolean returnValue = true;

        try

        {

            os.packInt(_value);

        }

        catch (java.io.IOException e)

        {

            returnValue = false;

        }

        return returnValue;

    }

}
```

When this abstract record is viewed in the object store browser it would be nice to see the current value. This is easy to do as we can read the state into an instance of our abstract record and call `getValue()`. The following is the object store browser plug-in source code:

```
public class SimpleRecordOSVPlugin implements StateViewerInterface

{

    /**

     * A uid node of the type this viewer is registered against has been

     expanded.

     * @param os

     * @param type

     * @param manipulator

     * @param node

     * @throws ObjectStoreException

     */

}
```

```

public void uidNodeExpanded(ObjectStore os,

                               String type,

                               ObjectStoreBrowserTreeManipulationInterface

                                   manipulator,

                               UidNode node,

                               StatePanel infoPanel)

                               throws ObjectStoreException

{

    // Do nothing

}

/**

 * An entry has been selected of the type this viewer is registered

 against.

 *

 * @param os

 * @param type

 * @param uid

 * @param entry

 * @param statePanel

 * @throws ObjectStoreException

 */

public void entrySelected(ObjectStore os,

                               String type,

                               Uid uid,

                               ObjectStoreViewEntry entry,

                               StatePanel statePanel)

```

```

        throws ObjectStoreException

    {

        SimpleRecord rec = new SimpleRecord();

        if ( rec.restore_state( os.read_committed(uid, type),
        ObjectType.ANDPERSISTENT ) )

        {

            statePanel.setData( "Value", rec.getValue() );

        }

    }

    /**
     * Get the type this state viewer is intended to be registered against.
     * @return
     */
    public String getType()

    {

        return "/StateManager/AbstractRecord/SimpleRecord";

    }

}

```

The method `uidNodeExpanded` is invoked when a UID (Unique Identification) representing the given type is expanded in the object store hierarchy tree. This is not required by this plugin as this abstract record is not visible in the object store directly it is only viewable via one of the lists in an atomic action. The method `entrySelected` is invoked when an entry is selected from the object view which represents an object with the given type. In both methods the `StatePanel` is used to display information regarding the state of the object. The state panel has the following methods that assist in display this information:

- `setInfo(String info)` – This method can be used to show general information.
- `setData(String name, String value)` – This method is used to put information into the table which is displayed by the object store browser tool.
- `enableDetailsButton(DetailsButtonListener listener)` – This method is used to enable the details button. The listener interface allows a plug-in to be informed

when the button is pressed. It is up to the plug-in developer to decide how to display this further information.

In this example we read the state from the object store and use the value returned by `getValue()` to put an entry into the state panel table. The `getType()` method returns the type this plug-in is to be registered against.

To add this plug-in to the object store browser it is necessary to package it into a JAR (Java Archive) file with a name that is prefixed with 'osbv-'. The JAR file must contain certain information within the manifest file so that the object store browser knows which classes are plug-ins. All of this can be performed using an Apache ANT (<http://ant.apache.org>) script, as follows:

```
<jar jarfile="osbv-simplerecord.jar">
    <fileset dir="build" includes="*.class"/>
    <manifest>
        <section name="arjuna-tools-objectstorebrowser">
            <attribute name="plugin-classname-1" value="
SimpleRecordOSVPlugin "/>
        </section>
    </manifest>
</jar>
```

Once the JAR has been created with the correct information in the manifest file it just needs to be placed in the 'bin/tools/plugins' directory.

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.

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 its 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>
```

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.

```
<rmic compiler="com.arjuna.common.tools.rmictool.RMICTool"
      classname="RMIOBJECTImpl"
      base="build-dir" 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 configurations

Orbix 2000

It is necessary to register all idl files with the Orbix 2000 interface repository.

The following configuration modifications are necessary to support transaction context propagation and interposition; it may be necessary to consult the Orbix 2000 documentation to determine how to accomplish this. A new orb name domain called *arjuna* should be created within the main Orbix 2000 domain being used by the application. It requires the following format:

```
arjuna
{
  portable_interceptor
  {
    orb_plugins = ["local_log_stream", "iiop_profile", "giop", "iiop",
                  "portable_interceptor"];

    ots_recovery_coordinator
    {
      recovery_coordinator:iiop:addr_list = ["<name>:<port>"];
    };

    ots_transaction
    {
      transaction:iiop:addr_list = ["+<name>:<port>"];
    };

    ots_context
    {
      binding:client_binding_list = ["OTS_Context",
"OTS_Context+GIOP+SIOP", "GIOP+SIOP", "OTS_Context+GIOP+IIOP",
"GIOP+IIOP"];
      binding:server_binding_list = ["OTS_Context", ""];
    };

    ots_interposition
    {
      binding:client_binding_list = ["OTS_Interposition",
"OTS_Interposition+GIOP+SIOP", "GIOP+SIOP", "OTS_Interposition+GIOP+IIOP",
"GIOP+IIOP"];
      binding:server_binding_list = ["OTS_Interposition", ""];
    };
  };
};
```

The <name> field should be substituted by the name of the machine on which *JBossTS* is being run. The <port> field should be an unused port on which the *JBossTS* recovery manager may listen for recovery requests.

When using transaction context propagation only, the `-ORBname arjuna.portable_interceptor.ots_context` parameter should be passed to the client and server. When using context propagation and interposition, the `-ORBname.arjuna.portable_interceptor.ots_interposition` parameter should be used. For example:

```
java mytest -ORBname arjuna.portable_interceptor.ots_context
```

Orbix2000 comes with its own implementation of the classes defined in the `CosTransactions.idl` file. Unfortunately these are incompatible with the version shipped with *JBossTS*. Therefore, it is important that the *JBossTS* jar files appear in the `CLASSPATH` before any Orbix2000 jars.

Note: Because of the way in which Orbix works with persistent POAs, if you want crash recovery support for your applications you must use one of the Arjuna ORB names provided (context or interposition) when running your clients and services.

Configuring JBossTS

Options

The following table shows the configuration features, with default values shown in italics. For more detailed information, the relevant section numbers are provided.

Configuration Name	Possible Values	Relevant Section
com.arjuna.ats.jta.support Subtransactions	<i>YES/NO</i>	
com.arjuna.ats.jta.jtaTMI Implementation	com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImpl com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImpl	
com.arjuna.ats.jta.jtaUTI Implementation	com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImpl com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImpl	
com.arjuna.ats.jta.xaBack offPeriod		

Table 4: JBossTS configuration options.

Appendix A

IDL Definitions

Introduction

The following sections detail the idl files which form the core of *JBossTS*.

Note: 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.

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 CostTSInteroperation

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, HeuristicHazard, TransactionRolledBack);
    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, TransactionRolledBack);
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, SynchronizationUnavailable);
void register_subtran_aware (in SubtransactionAwareResource r) raises
(Inactive, NotSubtransaction);

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

```

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

References

[OMG95] “CORBAservices: Common Object Services Specification”, OMG Document Number 95-3-31, March 1995.

[JTA99] “Java Transaction API”, Sun Microsystems, 1999.

Index

- ACID properties, **9**
- ArjunaOTS.idl, 120
- ArjunaSubtranAwareResource, 65
- Asynchronous commit, **51**
- Asynchronous prepare, **51**
- AtomicTransaction, 67
- BOA_init/create_POA, **56**
- Checked transactions, 52
 - CheckedAction, 54
 - default, 77
 - JBoss Transactions specifics, 54
- CheckedAction, **54**
- Co-located transaction server, **22**
- commit, **9, 15, 30, 31**
 - report_heuristics, 58
- Compliance with OTS specification, **12**
 - implementation choices, 16
- Configurable options, **116**
- Context management, **57**
 - direct and explicit, 58
 - indirect and implicit, 57
- Context propagation, **27**
- Control, 29
 - destroyControl, 30
 - JBoss Transactions specifics, 29
- Coordinator, 31
 - JBoss Transactions specifics, 33
- CosServices.cfg
 - overriding default name and location, 77
 - overview, 77
- CosTransactions.idl, 117
- Crash recovery
 - daemons, 80
 - RecoveryManager, 80
 - setup, 80
- Current, 33
 - JBoss Transactions specifics, 36
- Explicit interposition, **68**
 - Transactional Objects for Java, 68
- Heuristics, 33
 - report_heuristics, 33
- Interposition, **50, 68**
 - explicit interposition, 68
 - Transactional Objects for Java, 68
- JBoss Transactions
 - AtomicTransaction class, 67
 - basic interfaces, 64
 - Basic OTS interfaces, 13
 - compliance, 13
 - default settings, 77
 - enhanced OTS API, 14
 - extended resources, 65
 - extended subtransactions, 65
 - OTS compliance, 12
 - Transactional Objects for Java, 64
 - Transactional Objects for Java class
 - hierarchy, 15
- ORB portability
 - overview, 17
- ORB_init, **56**
- OTS
 - architecture, 18
 - heuristics, 33
 - introduction, 11
 - summary of JBoss Transactions specifics, 55
- OTS_ExplicitInterposition, **68**
- OTSAbstractRecord, 65
- Property variables
 - ASSUMED_COMPLETE_EXPIRY_TIME, 95
 - ASYNC_COMMIT, 51
 - ASYNC_PREPARE, 51
 - COMMIT_ONE_PHASE, 31
 - COMMITTED_TRANSACTION_RETRY_LIMIT, 95
 - ENABLE_STATISTICS, 36
 - EXPIRY_SCAN_INTERVAL, 95

FACTORY Contac_EXPIRY_TIME, 95
 filters, 27, 56
 initial references, 22
 INITIAL_REFERENCES_FILE, 77
 INITIAL_REFERENCES_ROOT, 77
 OBJECTSTORE_DIR, 77
 OTS_ALWAYS_PROPAGATE_CONTEXT, 49, 50, 77
 OTS_CHECKED_TRANSACTIONS, 54, 55, 77
 OTS_CONTEXT_PROP_MODE, 27, 56
 OTS_DEFAULT_TIMEOUT, 78
 OTS_ISSUE_RECOVERY_ROLLBACK, 94
 OTS_NEED_TRAN_CONTEXT, 50, 77
 OTS_SUPPORT_INTERPOSED_SYNCHRONIZATION, 45
 OTS_SUPPORT_ROLLBACK_SYNC, 44
 OTS_SUPPORT_SUBTRANSACTIONS, 33, 36, 55
 OTS_TRANSACTION_MANAGER, 22, 36, 55
 RecoveryEnablement, 79
 RESOLVE_SERVICE, 23
 TRANSACTION_SYNC, 31
 Recovery modules, 86
 RecoveryEnablement, 79
 RecoveryManager, 80
 initialisation, 80
 properties file, 80
 report_heuristics, 33, 58
 Resource, 37
 example, 71
 Separate transaction server, 22
 SubtransactionAwareResource, 39
 JBoss Transactions specifics, 42
 SubtransactionAwareResources, 39
 Subtransactions, 10, 26
 fault-isolation, 11
 modularity, 11
 one-phase property, 26
 Synchronizations, 43
 Terminator, 30
 JBoss Transactions specifics, 30
 Threading, 17
 optimisations, 51
 OTS_Thread class, 17
 Transaction
 default timeout value, 36
 Transaction context management, 19
 Transaction context propagation, 19
 Transaction contexts, 23
 Transaction processing overview, 9
 two-phase commit protocol, 9
 Transaction synchronisation, 30
 Transactional Objects for Java, 14
 configuration, 116
 interposition, 68
 TransactionalObject, 49
 JBoss Transactions specifics, 49, 50
 overview, 12
 TransactionFactory, 21
 Transactions
 nested transactions, 26
 propagation, 27
 Writing an OTS application
 initialising JBoss Transactions, 56
 overview, 56
 X/Open
 checked transactions, 53
 XA recovery, 88
 XAConnectionrecovery, 88
 BasicXARecovery, 89
 example, 89