JBoss Transactions 4.3.0

Quick Start Guide

JBTS-QSG-10/3/07

Legal Notices

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

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

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

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

Copyright

Software Version

JBoss Transactions 4.3.0

Restricted Rights Legend

Contents

About This Guide

What This Guide Contains

The Quick Start Guide contains information on how to use JBoss Transactions 4.3.0.

Audience

This guide is most relevant to engineers who are responsible for administering JBoss Transactions 4.3.0 installations. It is intended for those who are familiar with transactions in general and the OTS/JTS and JTA in particular.

Prerequisites

Familiarity with the JTA, OTS, transactions and ORBs.

Organization

This guide contains the following chapters:

- **Chapter 1, Quick Start to JTA:** This chapter covers the key features required to construct a basic JTA application with JBossTS.
- **Chapter 2, Quick Start to JTS/OTS:** This chapter covers the key features required to construct a basic OTS application using the raw OMG/OTS interfaces, and to configure JBossTS.

Documentation Conventions

The following conventions are used in this guide:

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

Table 1 Formatting Conventions

Additional Documentation

In addition to this guide, the following guides are available in the JBoss Transactions 4.3.0 documentation set:

- JBoss Transactions 4.3.0 *Release Notes*: Provides late-breaking information about JBoss Transactions 4.3.0.
- JBoss Transactions 4.3.0 *Installation Guide*: This guide provides instructions for installing JBoss Transactions 4.3.0.
- JBoss Transactions 4.3.0 *Programmer's Guide*: Provides guidance for writing applications.
- 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 4.3.0 should be directed to our support team.

Chapter 1

Quick Start to JTA

Introduction

This chapter will briefly cover the key features required to construct a JTA application. *It is assumed that the reader is familiar with the concepts of the JTA.*

Package layout

The key Java packages (and corresponding jar files) for writing basic JTA applications are:

- `com.arjuna.ats.jts`: this package contains the *JBossTS* implementations of the JTS and JTA.
- `com.arjuna.ats.jta`: this package contains local and remote JTA implementation support.
- `com.arjuna.ats.jdbc`: this package contains transactional JDBC 2.0 support.

All of these packages appear in the `lib` directory of the *JBossTS* installation, and should be added to the programmer's `CLASSPATH`.

In order to fully utilize all of the facilities available within *JBossTS*, it will be necessary to add the following jar files to your `CLASSPATH`: jta-spec1_0_1.jar, jdbc2_0-stdext.jar and jndi.jar.

Setting properties

JBossJTA has also been designed to be configurable at runtime through the use of various property attributes. These attributes can be provided at runtime on command line or specified through a properties file.

Specifying the object store location

JBossJTA requires an object store in order to persistently record the outcomes of transactions in the event of failures. In order to specify the location of the object store it is necessary to specify the location when the application is executed; for example:

```
java -
Dcom.arjuna.ats.arjuna.objectstore.objectStoreDir=/var/tmp/ObjectStore
myprogram
```

The default location is a directory under the current execution directory.

By default, all object states will be stored within the `defaultStore` subdirectory of the object store root, e.g., `/usr/local/Arjuna/TransactionService/ObjectStore/defaultStore`. However, this subdirectory can be changed by setting the `com.arjuna.ats.arjuna.objectstore.localOSRoot` property variable accordingly.

Demarcating Transactions

The Java Transaction API consists of three elements: a high-level application transaction demarcation interface, a high-level transaction manager interface intended for application server, and a standard Java mapping of the X/Open XA protocol intended for transactional resource manager. All of the JTA classes and interfaces occur within the `javax.transaction` package, and the corresponding *JBossJTA* implementations within the `com.arjuna.ats.jta` package.

UserTransaction

The `UserTransaction` interface provides applications with the ability to control transaction boundaries.

In JBossJTA, `UserTransaction` can be obtained from the static `com.arjuna.ats.jta.UserTransaction.userTransaction()` method. When obtained the UserTransaction object can be used to control transactions

```
//get UserTransaction
UserTransaction utx = com.arjuna.ats.jta.UserTransaction.userTransaction();
// start transaction work..
utx.begin();
.. do work
utx.commit();
```

TransactionManager

The `TransactionManager` interface allows the application server to control transaction boundaries on behalf of the application being managed.

In JBossJTA, transaction manager implementations can be obtained from the static `com.arjuna.ats.jta.TransactionManager.transactionManager()` method

The Transaction interface

The `Transaction` interface allows operations to be performed on the transaction associated with the target object. Every top-level transaction is associated with one `Transaction` object when the transaction is created. The `Transaction` object can be used to:

- enlist the transactional resources in use by the application.
- register for transaction synchronization call backs.

- commit or rollback the transaction.
- obtain the status of the transaction.

A `Transaction` object can be obtained using the `TransactionManager` by invoking the method getTransaction() method.

```
Transaction txObj = TransactionManager.getTransaction();
```

Local vs Distributed JTA implementations

In order to ensure interoperability between JTA applications, it is recommended to rely on the JTS/OTS specification to ensure transaction propagation among transaction managers.

In order to select the local JTA implementation it is necessary to perform the following steps:

1. make sure the property com.arjuna.ats.jta.jtaTMImplementation is set to com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple.
2. make sure the property com.arjuna.ats.jta.jtaUTImplementation is set to com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple.

In order to select the distributed JTA implementation it is necessary to perform the following steps:

3. make sure the property com.arjuna.ats.jta.jtaTMImplementation is set to com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple.
4. make sure the property com.arjuna.ats.jta.jtaUTImplementation is set to com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple.

JDBC and Transactions

ArjunaJTS supports the construction of both local and distributed transactional applications which access databases using the JDBC 2.0 APIs. JDBC 2.0 supports two-phase commit of transactions, and is similar to the XA X/Open standard. The JDBC 2.0 support is found in the com.arjuna.ats.jdbc package.

The ArjunaJTS approach to incorporating JDBC connections within transactions is to provide transactional JDBC drivers through which all interactions occur. These drivers intercept all invocations and ensure that they are registered with, and driven by, appropriate transactions. (There is a single type of transactional driver through which any JDBC driver can be driven. This driver is com.arjuna.ats.jdbc.TransactionalDriver, which implements the java.sql.Driver interface.)

Once the connection has been established (for example, using the java.sql.DriverManager.getConnection method), all operations on the connection will be monitored by JBossJTA. Once created, the driver and any connection can be used in the same way as any other JDBC driver or connection.

JBossJTA connections can be used within multiple different transactions simultaneously, i.e., different threads, with different notions of the current transaction, may use the same JDBC connection. JBossJTA does connection pooling for each transaction within the JDBC connection. So, although multiple threads may use the same instance of the JDBC connection, internally this may be using a different connection instance per transaction. With the exception of close, all operations performed on the connection at the application level will only be performed on this transaction-specific connection.

JBossJTA will automatically register the JDBC driver connection with the transaction via an appropriate resource. When the transaction terminates, this resource will be responsible for either committing or rolling back any changes made to the underlying database via appropriate calls on the JDBC driver.

Configurable options

The following table shows some of the configuration features, with default values shown in italics. For more detailed information, the relevant section numbers are provided. You should look at the various Programmers Guides for more options.

Configuration Name	Possible Values
com.arjuna.ats.jta.supportSubtransactions	*YES*/NO
com.arjuna.ats.jta.jtaTMImplementation	com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple/com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple
com.arjuna.ats.jta.jtaUTImplementation	com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple/com.arjuna.ats.internal.jta.transaction.jts.UserTransactionImple
com.arjuna.ats.jta.xaBackoffPeriod	Time in seconds.
com.arjuna.ats.jdbc.isolationLevel	Any supported JDBC isolation level.

Table 3: JBossJTA configuration options.

Chapter 2

Quick Start to JTS/OTS

Introduction

This chapter will briefly cover the key features required to construct a basic OTS application using the raw OTS interfaces defined by the OMG specification. *It is assumed that the reader is familiar with the concepts of the JTS/OTS and has read the relevant ORB specific portions of the JBossTS Programmer's Guide.* Further topics and the advanced facilities provided by *JBossTS* will be described in subsequent sections of this manual; references to chapters in the other manuals of the document set will be given in the relevant sections.

Package layout

The key Java packages (and corresponding jar files) for writing basic OTS applications are:

- `com.arjuna.orbportability`: this package contains the classes which constitute the ORB portability library and other useful utility classes.
- `org.omg.CosTransactions`: this package contains the classes which make up the `CosTransactions.idl` module.
- `com.arjuna.ats.jts`: this package contains the *JBossTS* implementations of the JTS and JTA.
- `com.arjuna.ats.arjuna`: this package contains further classes necessary for the *JBossTS* implementation of the JTS.
- `com.arjuna.ats.jta`: this package contains local and remote JTA implementation support.
- `com.arjuna.ats.jdbc`: this package contains transactional JDBC 2.0 support.

All of these packages appear in the `lib` directory of the *JBossTS* installation, and should be added to the programmer's `CLASSPATH`.

In order to fully utilize all of the facilities available within *JBossTS*, it will be necessary to add the following jar files to your `CLASSPATH`: jta-spec1_0_1.jar, jdbc2_0-stdext.jar and jndi.jar.

Setting properties

JBossTS has been designed to be highly configurable at runtime through the use of various property attributes, which will be described in subsequent sections. Although these attributes can be provided at runtime on the command line, it is possible (and may be more convenient)

to specify them through the single properties file `jbossjts-properties.xml`. At runtime *JBossTS* looks for its property file in the following order:

- the current working directory, i.e., where the application was executed from.
- the user's home directory.
- the CLASSPATH (via the getResource() method).

If found, all entries within this file will be added to the system properties. Obviously non-*JBossTS* specific properties can also be specified in this file.

Starting and terminating the ORB and BOA/POA

It is important that *JBossTS* is correctly initialized prior to any application object being created. In order to guarantee this, the programmer must use the `initORB` and `initBOA/initPOA` methods of the `ORBInterface` class described in the *ORB Portability Manual*. Using the `ORB_init` and `BOA_init/create_POA` methods provided by the underlying ORB will not be sufficient, and may lead to incorrectly operating applications. For example:

```
public static void main (String[] args)
{
ORBInterface.initORB(args, null);
ORBInterface.initOA();
. . .
};
```

The `ORBInterface` class has operations `orb()` and `boa()/poa()/rootPoa()` for returning references to the orb and boa/child POA/root POA respectively after initialization has been performed. If the ORB being used does not support the BOA (e.g., Sun's JDK 1.2) then `boa()` does not appear in the class definition, and `initBOA` will do nothing.

In addition, it is necessary to use `shutdownOA` and `shutdownORB` (in that order) prior to terminating an application to allow *JBossTS* to perform necessary cleanup routines. `shutdownOA` routine will either shutdown the BOA or the POA depending upon the ORB being used.

```
public static void main (String[] args)
{
. . .
ORBInterface.shutdownOA();
ORBInterface.shutdownORB();
};
```

No further CORBA objects should be used once shutdown has been called. It will be necessary to re-initialise the BOA/POA and ORB in such an event.

Note: In the rest of this document we shall use the term Object Adapter to mean either the Basic Object Adapter (BOA) or the Portable Object Adapter (POA). In addition, where possible we shall use the ORB Portability classes which attempt to mask the differences between POA and BOA.

Specifying the object store location

JBossTS requires an object store in order to persistently record the outcomes of transactions in the event of failures. In order to specify the location of the object store it is necessary to specify the location when the application is executed; for example:

```
java –
Dcom.arjuna.ats.arjuna.objectstore.objectStoreDir=/var/tmp/ObjectStore
myprogram
```

The default location is a directory under the current execution directory.

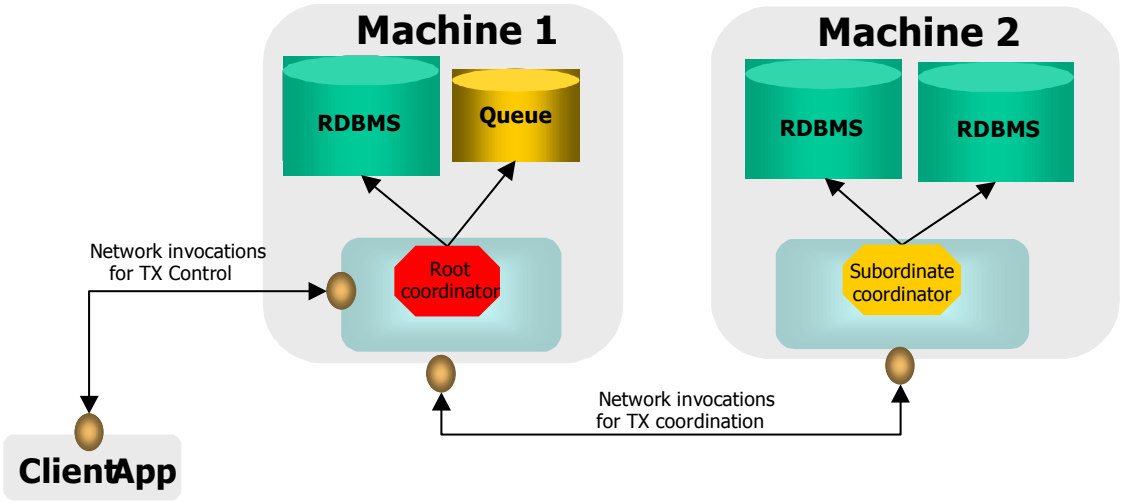
By default, all object states will be stored within the `defaultStore` subdirectory of the object store root, e.g., `/usr/local/Arjuna/TransactionService/ObjectStore/defaultStore`. However, this subdirectory can be changed by setting the `com.arjuna.ats.arjuna.objectstore.localOSRoot` property variable accordingly.

Implicit transaction propagation and interposition

Transactions can be created within one domain (e.g., process) and used within another. Therefore, information about a transaction (the transaction context) needs to be propagated between these domains. This can be accomplished in two ways:

- *Explicit propagation* means that an application propagates a transaction context by passing context objects (instances of the `Control` interface or the `PropagationContext` structure) defined by the Transaction Service as explicit parameters. Note, for efficiency reasons it is recommended that the `PropagationContext` be passed, rather than the `Control`.
- *Implicit propagation* means that requests on objects are implicitly associated with the client's transaction; they share the client's transaction context. The context is transmitted implicitly to the objects, without direct client intervention.

OTS objects supporting the `Control` interface are standard CORBA objects. When the 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 object performs on the interface (such as registering resources as participants within the transaction) are performed on the real object. However, this can have substantial penalties for the application if it frequently uses these interfaces due to the overheads of remote invocation. To avoid this overhead *JBossTS* supports *interposition*, whereby the server creates a local object which acts as a proxy for the remote transaction and fields all requests that would normally have been passed back to the originator. This surrogate registers itself with the original transaction coordinator to enable it to correctly participate in the termination of the transaction. Interposed coordinators effectively form a tree structure with their parent coordinators. This is shown in the figure below.



Note: implicit transaction propagation does not imply interposition will be used at the server, but (typically) interposition requires implicit propagation.

If implicit context propagation and interposition are 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 on which, if any, protocol (implicit or interposition) is being used. Implicit context propagation is only possible on those ORBs which either support filters/interceptors, or the `CosTSPortability` interface. Currently this is HP-ORB, JacORB, the JDK 1.4 ORB and Orbix 2000. Depending upon which type of functionality is required, the programmer *must* perform the following:

- Implicit context propagation:
  - set the `com.arjuna.ats.jts.contextPropMode` property variable to `CONTEXT`. If using Orbix 2000, see the Orbix 2000 configuration section in the Administrator's Guide.
- Interposition:
  - set the `com.arjuna.ats.jts.contextPropMode` 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*.

Obtaining Current

The `Current` pseudo object can be obtained from the `com.arjuna.ats.jts.OTSManager` class using its `get_current()` method.

Transaction termination

It is implementation dependant as to how long a `Control` remains able to access a transaction after it terminates. 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 explicitly using the `Terminator` interface then information about the transaction will be removed when all outstanding references to it are destroyed. However, the programmer can signal that the transaction information is no longer required 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 to using `Control` references apply as described above.

Transaction factory

By default, *JBossTS* does not use a separate transaction manager when creating transactions through the `Current` interface. Each transactional client essentially has its own transaction manager (`TransactionFactory`) which is co-located with it. By setting the `com.arjuna.ats.jts.transactionManager property` variable to `YES` this can be overridden at runtime. The transaction factory is located in the `bin` directory of the *JBossTS* distribution, and should be started by executing the `start-transaction-service` script located in `<ats_root>/bin`.

Typically `Current` locates the factory using 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 (or created) 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.

Note: this is the default name and location of the configuration file. The name of the file can be specified at runtime using the com.arjuna.orbportability.initialReferencesFile variable, and its location can be changed using the com.arjuna.orbportability.initialReferencesRoot variable:

java –Dcom.arjuna.orbportability.initialReferencesFile=ref –Dcom.arjuna.orbportability.initialReferencesRoot=c:\\temp prog

It is possible to override the default location mechanism by using the `com.arjuna.orbportability.resolveService` 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 com.arjuna.orbportability.resolveService is specified when the transaction factory is run, then the factory will register itself with the specified resolution mechanism.

Recovery manager

You will need to start the recovery manager subsystem to ensure that transactions are recovered despite failures. In order to do this, you should run the start-recovery-manager script in <ats_root>/bin.

Chapter 3

Quick Start to Web Services Transactions

Configuring the Web Services component

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

Configuration Name	Possible Values	Relevant Section
com.arjuna.orbportability.initialReferencesFile	CosServices.cfg	
com.arjuna.orbportability.initialReferencesRoot	The directory containing the file arjuna.properties.	
ArjunaJTS_LicenceKey	System specific licence.	
com.arjuna.orbportability.resolveService	*CONFIGURATION_FILE*/NAME_SERVICE/BIND_CONNECT	
com.arjuna.ats.arjuna.objectstore.objectStoreDir	Any location that the application can write to.	
com.arjuna.ats.arjuna.objectstore.localOSRoot	defaultStore	
PROPERTIES_FILE	arjuna.properties	
com.arjuna.ats.arjuna.coordinator.asyncPrepare	YES/*NO*	
com.arjuna.ats.arjuna.coordinator.asyncCommit	YES/*NO*	
com.arjuna.ats.arjuna.coordinator.commitOnePhase	*YES*/NO	
com.arjuna.ats.arjuna.coordinator.transactionSync	*ON*/OFF	
com.arjuna.ats.arjuna.coordinator.enableStatistics	*ON*/OFF	
com.arjuna.ats.jts.alwaysPropagateContext	*YES*/NO	
com.arjuna.ats.jts.defaultTimeout	No timeout	
com.arjuna.ats.jts.supportRollbackSync	YES/*NO*	
com.arjuna.ats.jts.supportInterposedSynchronization	YES/*NO*	
com.arjuna.ats.jts.supportSubtransactions	*YES*/NO	
com.arjuna.ats.jts.checkedTransactions	YES/*NO*	
com.arjuna.ats.jts.transactionManager	YES/*NO*	

com.arjuna.ats.jts.needTranContext	YES/*NO*	
com.arjuna.ats.arjuna.coordinator.txReaperTimeout	120000000 microseconds	
com.arjuna.ats.arjuna.coordinator.txReaperMode	*NORMAL*/DYNAMIC	
com.arjuna.ats.jts.contextPropMode	*NONE*/CONTEXT/INTERPOSITION	

Table 2: JBossTS configuration options.

Index