

**JBossJTS Administration Guide**

# **Administration of the JBossJTS toolkit, used as part of the Transaction Service**

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

edited by Misty Red Hat Stanley-Jones

---

---

---

Preface .....	v
1. Prerequisites .....	v
2. Document Conventions .....	v
2.1. Typographic Conventions .....	v
2.2. Pull-quote Conventions .....	vii
2.3. Notes and Warnings .....	vii
3. We Need Feedback! .....	viii
<b>1. Introduction .....</b>	<b>1</b>
<b>2. Starting and Stopping the Transaction Manager .....</b>	<b>3</b>
<b>3. OTS and Java EE Transaction Service Management .....</b>	<b>5</b>
3.1. Starting the run-time system .....	5
3.1.1. OTS configuration file .....	6
3.1.2. Name service .....	6
3.1.3. resolve_initial_references .....	6
3.1.4. Resolution services supported per ORB .....	6
3.2. XA Specific management .....	7
3.3. Selecting the JTA implementation .....	7
<b>4. Failure Recovery Administration .....</b>	<b>9</b>
4.1. The Recovery Manager .....	9
4.2. Configuring the Recovery Manager .....	9
4.3. Output .....	10
4.4. Periodic Recovery .....	10
4.5. Expired Entry Removal .....	12
<b>5. ORB-specific Configurations .....</b>	<b>13</b>
5.1. JacORB .....	13
<b>6. Initializing JBossTS Applications .....</b>	<b>15</b>
A. Revision History .....	17

---

---

## Preface

# 1. Prerequisites

JBossJTS is the implementation of the JTS API provided for use with JBoss Transaction Manager. For more details about the Transaction Manager, refer to the ArjunaCore Installation and Administration Guide, which is also available from the JBoss Transactions website, <http://www.jboss.org/jbosstm>.

## 2. Document Conventions

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

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

### 2.1. Typographic Conventions

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

Mono-spaced Bold

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

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

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

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

Press **Enter** to execute the command.

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

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

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

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

### Proportional Bold

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

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

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

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

*Mono-spaced Bold Italic Of Proportional Bold Italic*

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

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

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

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

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

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

Publican is a *DocBook* publishing system.

## 2.2. Pull-quote Conventions

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

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

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

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

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

import javax.naming.InitialContext;

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

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

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

## 2.3. Notes and Warnings

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



### Note

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



### Important

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



### Warning

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

## 3. We Need Feedback!

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



# Introduction

Since the release of JBossTS 4.1, the Web Services Transaction product has been merged into JBoss Transactions. JBoss Transactions is thus a single product that is compliant with all of the major distributed transaction standards and specifications.

Knowledge of Web Services is not required to administer a JBoss Transactions installation that only uses the CORBA/J2EE component, nor is knowledge of CORBA required to use the Web Services component. This, administrative tasks are separated when they touch only one component or the other.

Apart from ensuring that the run-time system is executing normally, there is little continuous administration needed for the JBossJTS software. Refer to [Important Points for Administrators](#) for some specific concerns.

## Important Points for Administrators

- The present implementation of the JBossJTS system provides no security or protection for data. The objects stored in the JBossJTS object store are (typically) owned by the user who ran the application that created them. The Object Store and Object Manager facilities make no attempt to enforce even the limited form of protection that Unix/Windows provides. There is no checking of user or group IDs on access to objects for either reading or writing.
- Persistent objects created in the Object Store never go away unless the `StateManager.destroy` method is invoked on the object or some application program explicitly deletes them. This means that the Object Store gradually accumulates garbage (especially during application development and testing phases). At present we have no automated garbage collection facility. Further, we have not addressed the problem of dangling references. That is, a persistent object, A, may have stored a Uid for another persistent object, B, in its passive representation on disk. There is nothing to prevent an application from deleting B even though A still contains a reference to it. When A is next activated and attempts to access B, a run-time error will occur.
- There is presently no support for version control of objects or database reconfiguration in the event of class structure changes. This is a complex research area that we have not addressed. At present, if you change the definition of a class of persistent objects, you are entirely responsible for ensuring that existing instances of the object in the Object Store are converted to the new representation. The JBossJTS software can neither detect nor correct references to old object state by new operation versions or vice versa.
- Object store management is critically important to the transaction service.



# Starting and Stopping the Transaction Manager

By default the transaction manager starts up in an active state such that new transactions can be created immediately. If you wish to have more control over this it is possible to set the `CoordinatorEnvironmentBean.startDisabled` configuration option to `YES` and in which case no transactions can be created until the transaction manager is enabled via a call to method `TxControl.enable()`.

It is possible to stop the creation of new transactions at any time by calling method `TxControl.disable`. Transactions that are currently executing will not be affected. By default recovery will be allowed to continue and the transaction system will still be available to manage recovery requests from other instances in a distributed environment. (See the Failure Recovery Guide for further details). However, if you wish to disable recovery as well as remove any resources it maintains, then you can pass `true` to method `TxControl.disable`; the default is to use `false`.

If you wish to shut the system down completely then it may also be necessary to terminate the background transaction reaper (see the Programmers Guide for information about what the reaper does.) In order to do this you may want to first prevent the creation of new transactions (if you are not creating transactions with timeouts then this step is not necessary) using method `TxControl.disable`. Then you should call method `TransactionReaper.terminate`. This method takes a Boolean parameter: if `true` then the method will wait for the normal timeout periods associated with any transactions to expire before terminating the transactions; if `false` then transactions will be forced to terminate (rollback or have their outcome set such that they can only ever rollback) immediately.



## Note

if you intent to restart the recovery manager later after having terminated it then you **MUST** use the `TransactionReaper.terminate` method with asynchronous behavior set to `false`.



# OTS and Java EE Transaction Service Management

## 3.1. Starting the run-time system

The JBossTS run-time support consists of run-time packages and the OTS transaction manager server. By default, JBossTS does not use a separate transaction manager server. Instead, transaction managers are co-located with each application process to improve performance and improve application fault-tolerance by reducing application dependency on other services.

When running applications which require a separate transaction manager, set the `JTSEnvironmentBean.transactionManager` environment variable to value `YES`. The system locates the transaction manager server in a manner specific to the ORB being used. This method may be any of:

- Being registered with a name server.
- Being added to the ORB's initial references.
- Via a JBossTS specific references file.
- By the ORB's specific location mechanism (if applicable).

You override the default registration mechanism by using the `OrbPortabilityEnvironmentBean.resolveService` environment variable, which takes the following values:

**Table 3.1. Possible values of `OrbPortabilityEnvironmentBean.resolveService`**

CONFIGURATION_FILE	This is the default, and causes the system to use the <code>CosServices.cfg</code> file.
NAME_SERVICE	JBossTS attempts to use a name service to register the transaction factory. If this is not supported, an exception is thrown.
BIND_CONNECT	JBossTS uses the ORB-specific bind mechanism. If this is not supported, an exception is thrown.
RESOLVE_INITIAL_REFERENCES	JBossTS attempts to register the transaction service with the ORB's initial service references. If the ORB does not support this, an exception is thrown, and another option must be used.

### 3.1.1. 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` is located at runtime by the following `OrbPortabilityEnvironmentBean` properties:

<code>initialReferencesRoot</code>	The directory where the file is located, defaulting to the current working directory.
<code>initialReferencesFile</code>	The name of the configuration file itself, <code>CosServices.cfg</code> by default.

The OTS server automatically registers itself in the `CosServices.cfg` file if the `OrbPortabilityEnvironmentBean` option is used, creating the file if necessary. Stale information is also automatically removed. Machines sharing the same transaction server should have access to this file, or a copy of it locally.

#### Example 3.1. Example ORB reference file settings

```
OrbPortabilityEnvironmentBean.initialReferencesFile=myFile
OrbPortabilityEnvironmentBean.initialReferencesRoot=/tmp
```

### 3.1.2. Name service

If your ORB supports a name service, and JBossTS is configured to use it, the transaction manager is registered with it automatically. There is no further work required.



#### Note

This option is not used for JacORB

### 3.1.3. `resolve_initial_references`

Currently this option is only supported for JacORB.

### 3.1.4. Resolution services supported per ORB

Resolution Mechanism	ORB
OTS configuration file	All available ORBs
Name Service	JacORB

Resolution Mechanism	ORB
resolve_initial_references	JacORB

## 3.2. XA Specific management

Each XA Xid that JBossTS creates must have a unique node identifier encoded within it. JBossTS only recovers transactions and states that match a specified node identifier. Provide the node identifier with the `CoreEnvironmentBean.nodeIdentifier` property. This value must be unique across your JBossTS instances. If you do not provide a value, JBossTS generates one and reports the value via the logging infrastructure.

When running XA recovery, you need to specify which types of Xid JBossTS can recover. Use the `JTAEnvironmentBean.xaRecoveryNodes` property to provide one or more values, in a space-separated list.



### Important

A value of `**` forces JBossTS to recover, and possibly rollback, all transactions, regardless of their node identifier. Use this value with extreme caution.

## 3.3. Selecting the JTA implementation

Two variants of the JTA implementation are now provided and accessible through the same interface. These are:

### Purely local JTA

Only non-distributed JTA transactions can be executed. This is the only version available with the JBossJTA product.

### Remote, CORBA-based JTA

Distributed JTA transactions can be executed. This version is only available with the JBossTS product and requires a supported CORBA ORB.

Both of these implementations are fully compatible with the transactional JDBC driver provided with JBossTS.

### Procedure 3.1. Selecting the local JTA implementation

1. Set the property `JTAEnvironmentBean.jtaTMImplementation` to value `com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerImple`.
2. Set the property `JTAEnvironmentBean.jtaUTImplementation` to value `com.arjuna.ats.internal.jta.transaction.arjunacore.UserTransactionImple`.



### Note

These settings are the default values for the properties and do not need to be specified if the local implementation is required.

### Procedure 3.2. Selecting the remote JTA implementation

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



# Failure Recovery Administration

The failure recovery subsystem of JBossJTS 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 [Section 4.1, “The Recovery Manager”](#). 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`.



## Warning

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.

## 4.1. The Recovery Manager

The failure recovery subsystem of JBossJTS requires that the stand-alone Recovery Manager process be running for each `ObjectStore` (typically one for each node on the network that is running JBossJTS applications). The `RecoveryManager` file is located in the `arjunaJTS` 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 initialized, i.e.,

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

## 4.2. Configuring the Recovery Manager

The `RecoveryManager` reads the properties defined in the `jbossts-properties.xml` file.

A default version of `jbossts-properties.xml` is supplied with the distribution. This can be used without modification, except possibly the debug tracing fields, as shown in [Section 4.3, “Output”](#).

### 4.3. 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 mechanism provided by **jboss logging**, which provides a high level interface that hides differences that exist between existing logging APIs such as Jakarta log4j or JDK logging API.

The configuration of **jboss logging** depends on the underlying logging framework that is used, which is determined by the availability and ordering of alternatives on the classpath. Please consult the jboss logging documentation for details. Each log message has an associated log Level, that gives the importance and urgency of a log message. The set of possible Log Levels, in order of least severity, and highest verbosity, is:

1. TRACE
2. DEBUG
3. INFO
4. WARN
5. ERROR
6. FATAL

Messages describing the start and the periodical behavior made by the `RecoveryManager` are output using the `INFO` level. If other debug tracing is wanted, the finer debug or trace levels should be set appropriately.

Setting the normal recovery messages to the `INFO` level allows the `RecoveryManager` to produce a moderate level of reporting. If nothing is going on, it just reports the entry into each module for each periodic pass. To disable `INFO` messages produced by the Recovery Manager, the logging level could be set to the higher level of `ERROR`, which means that the `RecoveryManager` will only produce `ERROR`, `WARNING`, or `FATAL` messages.

### 4.4. 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. These recovery modules are instances of classes that implement the `com.arjuna.ats.arjuna.recovery.RecoveryModule` interface. Each module has responsibility for a particular category of transaction or resource. The set of recovery modules used is dynamically loaded, using properties found in the `RecoveryManager` property file.

The interface has two methods: `periodicWorkFirstPass` and `periodicWorkSecondPass`. At an interval defined by property `com.arjuna.ats.arjuna.recovery.periodicRecoveryPeriod`, the `RecoveryManager` calls the first pass method on each property, then waits for a brief period, defined by property `com.arjuna.ats.arjuna.recovery.recoveryBackoffPeriod`. Next, it calls the second pass of each module. Typically, in the first pass, the module scans the relevant part of the `ObjectStore` to find transactions or resources that are in-doubt. An in-doubt transaction may be part of the way through the commitment process, for instance. On the second pass, if any of the same items are still in-doubt, the original application process may have 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 is likely to break the consistency. Accordingly, the recovery modules use a mechanism, implemented in the `com.arjuna.ats.arjuna.recovery.TransactionStatusManager` 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 completes, usually generating a warning. Recovery of such a transaction is the responsibility of the `RecoveryManager`.

It is clearly important to set the interval periods appropriately. The total iteration time will be the sum of the `periodicRecoveryPeriod` and `recoveryBackoffPeriod` properties, 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 with processes or machines that have crashed or are inaccessible. There are mechanisms in the recovery system to avoid trying to recover the same transaction indefinitely. The total iteration time affects how long a resource will remain inaccessible after a failure. – `periodicRecoveryPeriod` should be set accordingly. Its default is 120 seconds. The `recoveryBackoffPeriod` can be comparatively short, and defaults to 10 seconds. – Its purpose is mainly to reduce the number of transactions that are candidates for recovery and which thus require a call to the original process to see if they are still in progress.



### Note

In previous versions of **JBossJTS**, there was no contact mechanism, and the back-off period needed to be long enough to avoid catching transactions in flight at all. From 3.0, there is no such risk.

Two recovery modules, implementations of the `com.arjuna.ats.arjuna.recovery.RecoveryModule` interface, are supplied with **JBossJTS**. These modules support 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 `RecoveryManager`. The recovery modules are registered with the `RecoveryManager` using `RecoveryEnvironmentBean.recoveryModuleClassNames`. 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 a failure in an application process might occur while a periodic recovery pass is in progress. The default Recovery Extension settings are:

```
<entry key="RecoveryEnvironmentBean.recoveryModuleClassNames">
    com.arjuna.ats.internal.arjuna.recovery.AtomicActionRecoveryModule
    com.arjuna.ats.internal.txoj.recovery.TORecoveryModule
    com.arjuna.ats.internal.jts.recovery.transactions.TopLevelTransactionRecoveryModule
    com.arjuna.ats.internal.jts.recovery.transactions.ServerTransactionRecoveryModule
    com.arjuna.ats.internal.jta.recovery.jts.XARecoveryModule
</entry>
```

### 4.5. Expired Entry Removal

The operation of the recovery subsystem cause some entries to be made in the ObjectStore that are not removed in normal progress. The RecoveryManager has a facility for scanning for these and removing items that are very old. Scans and removals are performed by implementations of the `com.arjuna.ats.arjuna.recovery.ExpiryScanner` interface. These implementations are loaded by giving the class names as the value of a property `RecoveryEnvironmentBean.expiryScannerClassNames`. The RecoveryManager calls the `scan()` method on each loaded Expiry Scanner implementation at an interval determined by the property `RecoveryEnvironmentBean.expiryScanInterval`. This value is given in hours, and defaults to 12hours. An `expiryScanInterval` value of zero suppresses any expiry scanning. If the value 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.

The kinds of item that are scanned for expiry are:

#### TransactionStatusManager items

One TransactionStatusManager item is created by every application process that uses **JBossJTS**. It contains the information that allows the RecoveryManager to determine if the process that initiated the transaction is still alive, and its status. The expiry time for these items is set by the property `com.arjuna.ats.arjuna.recovery.transactionStatusManagerExpiryTime`, expressed in hours. The default is 12, and 0 (zero) means never to expire. The expiry time should be greater than the lifetime of any single processes using **JBossJTS**.

The Expiry Scanner properties for these are:

```
<entry key="RecoveryEnvironmentBean.expiryScannerClassNames">

    com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner
</entry>
```

# ORB-specific Configurations

## 5.1. JacORB

For JacORB to function correctly it needs a valid `jacorb.properties` or `.jacorb.properties` file in one of the following places, in searched order:

1. The classpath
2. The home directory of the user running the JBoss Transaction Service. The home directory is retrieved using `System.getProperty( "user.home" );`
3. The current directory
4. The `lib/` directory of the JDK used to run your application. This is retrieved using `System.getProperty( "java.home" );`



### Note

A template `jacorb.properties` file is located in the JacORB installation directory.

Within the JacORB properties file there are two important properties which must be tailored to suit your application.

- `jacorb.poa.thread_pool_max`
- `jacorb.poa.thread_pool_min`

These properties specify the minimum and maximum number of request processing threads that JacORB uses in its thread pool. If no threads are available, may block until a thread becomes available.. For more information on configuring JacORB, refer to the JacORB documentation.



### Important

JacORB includes its own implementation of the classes defined in the `CosTransactions.idl` file. Unfortunately these are incompatible with the version shipped with JBossTS. Therefore, the JBossTS jar files absolutely must appear in the CLASSPATH before any JacORB jars.

When running the recovery manager, it should always uses the same well-known port for each machine on which it runs. Do not use the `OAPort` property provided by JacORB unless the recovery manager has its own `jacorb.properties` file or the property is provided on the command line when starting the recovery manager. If the recovery manager and other components of JBossTS

share the same `jacorb.properties` file, use the `JTSEnvironmentBean.recoveryManagerPort` and `JTSEnvironmentBean.recoveryManagerAddress` properties.

# Initializing JBossTS Applications

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





---

# Appendix A. Revision History

Revision History

Revision 0

Wed Sep 1 2010

MistyStanley-

Jones<misty@redhat.com>

Conversion to Docbook

