JBoss Transactions 4.3.0

Administration Guide

JBTS-AG-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 Administration Guide contains information on how to use JBoss Transactions 4.3.0.

Audience

This guide is most relevant to engineers who are responsible for administration of JBoss Transactions 4.3.0 installations.

Prerequisites

In order to administer *JBossTS* it is first necessary to understand that it relies on TxCore for a lot of the transaction functionality. As such, it is important to read the TxCore Administration Guide before attempting to administer *JBossTS*.

Organization

This guide contains the following chapters:

- **Chapter 1, Administration of JBossTS:** This chapter describes how to administer and 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 | **	A path to a function or dialog box within an interface. For example, "**Select File** | **Open**." indicates that you should select the Open function

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 4.3.0 *Quick Start Guide*: Getting started with JBossTS; not for a novice user.
- 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

Administering JBossTS

Introduction

In this chapter we shall discuss how to administer an *JBossTS* installation.

With the release of JBossTS 4.1, we have merged the separate Web Services Transaction product into JBoss Transactions. This provides a single product that is compliant with all of the major distributed transaction standards and specifications. However, performing product administration for JBossTS 4.2 does not mean that you must know about Web Services if all you are interested in is the CORBA/J2EE component and vice versa. As a result, where appropriate we have separated the administration functions into separate sections.

JBossTS runtime information

Each module that comprises *JBossTS* possesses a class called Info. These classes all provide a single toString method that returns an XML document representing the configuration information for that module. So, for example:

```
<module-info name="arjuna"><source-identifier>unknown</source-
identifier><build-information>Arjuna Technologies [mlittle] (Windows 2000
5.0)</build-information><version>unknown</version><date>2002/06/15 04:06
PM</date><notes></notes><configuration><properties-file
dir="null">arjuna.properties</properties-file><object-store-
root>null</object-store-root></configuration></module-info>
```

How to manage the JBossTS Object Store

Within the transaction service installation, the object store is updated regularly whenever transactions are created, or when *Transactional Objects for Java* is used. In a failure free environment, the only object states which should reside within the object store are those representing objects created with the *Transactional Objects for Java* API. However, if failures occur, transaction logs may remain in the object store until crash recovery facilities have resolved the transactions they represent. As such it is very important that the contents of the object store are not deleted without due care and attention, as this will make it impossible to resolve in doubt transactions. In addition, if multiple users share the same object store it is important that they realize this and do not simply delete the contents of the object store assuming it is an exclusive resource.

Apart from ensuring that the run-time system is executing normally, there is little continuous administration needed for the *JBossTS* software. There are a few points however, that should be made:

- The present implementation of the *JBossTS* system provides no security or protection for data. It is recommended that the objects stored in the *JBossTS* are owned by user *arjuna*. 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 *JBossTS* 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.

OTS/J2EE Transaction service management

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: transaction managers are co-located with each application process to improve performance and improve application fault-tolerance (the application is no longer dependant upon another service in order to function correctly).

When running applications which require a separate transaction manager, you must set the `com.arjuna.ats.jts.transactionManager` 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).

It is possible to override the default registration mechanism by using the `com.arjuna.orbportability.resolveService` environment 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 register 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.
- *RESOLVE_INITIAL_REFERENCES*: *JBossTS* will attempt to register the transaction service with the ORBs initial service references.

Note: this may not be supported on all ORBs. An exception will be thrown in that case and another option will have to be used.

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 actual location is determined at runtime by searching the CLASSPATH for an old copy of the file resident within an `etc` directory, or, if one is not found, the location of the TransactionService properties file directory.

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. Machines which wish to share the same transaction server should have access to (or a copy of) this file.

The name and location of the file can be overridden using the `com.arjuna.orbportability.initialReferencesFile` and `com.arjuna.orbportability.initialReferencesRoot` property variables, respectively. It is recommended that the `com.arjuna.orbportability.initialReferencesRoot` variable is set. For example:

```
com.arjuna.orbportability.initialReferencesFile=myFile
com.arjuna.orbportability.initialReferencesRoot=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. This option is currently available for Orbix 2000 and HP-ORB.

resolve_initial_references

Currently this option is only supported for Orbix 2000 and JacORB.

Resolution service table

The following table summarizes the different ways in which the OTS transaction manager may be located on specific ORBs:

Resolution Mechanism	ORB
OTS configuration file	All available ORBs
Name Service	Orbix 2000, HP-ORB, JacORB
resolve_initial_references	Orbix 2000, JacORB

Table 2: Locating the OTS transaction manager server.

XA specific management

Each XA Xid that JBossTS creates must have a unique node identifier encoded within it and JBossTS will only recover transactions and states that match a specified node identifier. The node identifier to use should be provided to JBossTS via the `com.arjuna.ats.arjuna.xa.nodeIdentifier` property. You must make sure this value is unique across your JBossTS instances. If you do not provide a value, then JBossTS will fabricate one and report the value via the logging infrastructure.

When running XA recovery it is therefore necessary to tell JBossTS which types of Xid it can recover. Each Xid that JBossTS creates has a unique node identifier encoded within it and JBossTS will only recover transactions and states that match a specified node identifier. The node identifier to use should be provided to JBossTS via a property that starts with the name `com.arjuna.ats.jta.xaRecoveryNode`; multiple values may be provided. A value of '*' will force JBossTS to recover (and possibly rollback) all transactions irrespective of their node identifier and should be used with caution.

Selecting the JTA implementation

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

- A purely local JTA, which only allows non-distributed JTA transactions to be executed. This is the only version available with the JBossJTA product.
- A remote, CORBA-based JTA, which allows distributed JTA transactions to be executed. This version is only available with the JBossTS product and requires a supported CORBA ORB.

Note: both of these implementations are fully compatible with the transactional JDBC driver provided with JBossTS.

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.

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

In order to select the remote 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.

Web Service Transaction service management

Note: The Web Services transactions component is not available in the 4.2 release of JBossTransactions, but will be available in the 4.2.1 release.

The basic building blocks of a transactional Web Services application are the application itself, the Web services that the application consumes, the Transaction Manager, and the transaction participants which support those Web services. Though it is unlikely in a typical deployment that a single developer will be expected to support all of these roles, the concepts are presented here so that a global model can be visualized. It will often be the case that developers will produce services, or applications that consume services, whilst system administrators will run the transaction-management infrastructure.

The Transaction Manager

The transaction manager is a Web service which coordinates JBossTS transactions. It is the only software component in JBossTS that is meant to be run directly as a network service, rather than to support end-user code. The transaction manager runs as a JAXM request/response Web service.

Note: When starting up an application server instance that has JBossTS deployed within it, you may see various “error” messages in the console or log. For example 16:53:38,850 ERROR [STDERR] Message Listener Service: started, message listener jndi name activationcoordinator". These are for information purposes only and are not actual errors.

Configuring the Transaction Manager

The Transaction Manager and related infrastructure are configured by means of properties files: wscf.xml, wst.xml and wstx.xml. These files can be found in the `conf` directory and are used for configuring the demo application and the standalone module.

For the most part the default values in these files need not be altered. However, two values MUST be changed in order for the Transaction Manger to run. The `com.arjuna.ats.arjuna.objectstore.objectStoreDir` property determines the location of the persistent store used to record transaction state. The default value of `C:/temp/ObjectStore` should be changed to a value appropriate to your system. For

production environments this directory should reside on fault tolerant media such as a RAID array. The second critical property is `TxCore_LicenceKey` The value of this property should be set to the license string provided to you by Arjuna Technologies. If you do not have a license key, please email support@arjuna.com or visit our web site to obtain a free, time limited evaluation license.

When a standalone coordinator is being used by an application it is necessary to enable and modify two further properties in wstx.xml, `com.arjuna.mw.wst.coordinatorURL` and `com.arjuna.mw.wst.terminatorURL`. These contain the URLs necessary for the client application to contact the standalone coordinator and should be configured with the correct hostname and port of the standalone coordinator.

Please note that JBossTS is highly modular. In order to allow flexible deployment of individual components, the same property values are sometimes required to appear in more than one configuration file. For the majority of configurations, you should maintain consistent values for properties that are defined in more than one file.

Deploying the Transaction Manager

The JBossTS Web Service Transaction Manager component consists of a number of .jar files containing the application's class files, plus Web service (.war) files which expose the necessary services. Programmers will typically include all these components in their application .ear file during application development, simplifying deployment of the transaction infrastructure. For production usage the Transaction Manager may be installed as an application in its own right, allowing for centralized configuration and management at the server level, independent of specific applications. The demonstration application shipped with JBossTS provides a sample deployment descriptor illustrating how the Transaction Manager components can be included in an application.

JBossTS 4.0 uses fixed endpoints for its underlying protocol communication. Therefore, problems may arise if multiple applications using JBossTS are deployed to the same server concurrently. If you wish to deploy several transactional applications in the same server, the Transaction Manger must be deployed as a separate application and not embedded within the deployment of individual applications.

The `coordinator` directory in the JBossTS installation has been provided to assist in the configuration and deployment of a stand-alone transaction manager. In order to use this, you must:

- Have installed JBossTS 4.0.
- Have a separate application server installation for the coordinator; this can be on a separate machine. If you are using JBoss, then see http://www.yorku.ca/dkha/jboss/docs/MultipleInstances.htm
- Install ant 1.4 or later.

Note: It is important that a separate application server installation be used from the one that clients and services are deployed into since otherwise conflicts may occur between the various JBossTS components.

You must then edit the build.xml included with `coordinator` to point to the application server installation where the transaction coordinator will be deployed and the location of the JBossTS installation. The files ws-c_jaxm_web-app.xml and ws-t_jaxm_web-app.xml in the dd directory of `coordinator` are the deployment descriptors for the WS-C and WS-T war files. These files contain templated URLs. During the build phase, ant will substitute the hostname and port values you specify in the build.xml into these files.

Run ant, with target deploy-weblogic, deploy-jboss or deploy-webmethods, to create and deploy a new coordinator into the correct application server installation.

Now all you need to do is point your client at the required coordinator. To do this, generate the demo application specifying the port and hostname of the coordinator.

Deployment descriptors

In general, it should not be necessary to change the contents of the various deployment descriptors used by JBossTS. However, if you do need to modify them they are all included in the `coordinator` module.

Not all JBossTS components have ready access to the information in the deployment descriptors. Therefore, if you modify the JNDI names used by any of the WS-C or WS-T deployment descriptors you may need to inform other JBossTS components at runtime. This can be accomplished by setting an appropriate property in the wstx.xml configuration file.

The following table shows the default JNDI name used by the deployment descriptors and the corresponding property to set if the default value is changed.

JNDI Name	Property
Activationrequester	com.arjuna.mw.wst.at.activationrequester
Activationcoordinator	com.arjuna.mw.wst.at.activationcoordinator
Completionparticipant	com.arjuna.mw.wst.at.completionparticipant
Registrationrequester	com.arjuna.mw.wst.at.registrationrequester
durable2pcdispatcher	com.arjuna.mw.wst.at.durable2pcdispatcher
durable2pcparticipant	com.arjuna.mw.wst.at.durable2pcparticipant
volatile2pcdispatcher	com.arjuna.mw.wst.at.volatile2pcdispatcher
volatile2pcparticipant	com.arjuna.mw.wst.at.volatile2pcparticipant
businessagreementwithparticipantcompletiondispatcher	com.arjuna.mw.wst.ba.businessagreementwpcdispatcher
businessagreementwithparticipantcompletionparticipant	com.arjuna.mw.wst.ba.businessagreementwpcparticipant
businessagreementwithcoordinatorcompletiondispatcher	com.arjuna.mw.wst.ba.businessagreementwccdispatcher
Businessagreementwithcoordinatorcompletionparticipant	com.arjuna.mw.wst.ba.businessagreementwccparticipant

Table 3: Deployment descriptor values and properties.

Failure recovery administration

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.

Caution: 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.

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 class is located in the JAR `<ats_root>/lib/arjunacore.jar` within the package `com.arjuna.ats.arjuna.recovery`. 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
```

Configuring the Recovery Manager

The RecoveryManager reads the properties defined in the arjuna.properties file and then also reads the property file RecoveryManager.properties, 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 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 in the `jbossts-properties.xml` file 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 Jakarta log4j or JDK 1.4 logging API. CLF

indirects all logging via the Apache Commons Logging framework and configuration is assumed to occur through that framework.

With the CLF applications make logging calls on logger objects. Loggers 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 to filter logging messages according to finer granularity an application may define. That is, when a log message is provided to the logger 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.

According to these variables the CLF defines three interfaces. A particular product may implement its own classes according to 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. JBossTS uses the default level assigned to its logger objects (DEBUG). However, it uses the finer debugging features to disable or enable debug messages. Finer debugging values used by the JBossTS are defined below:

Debugging level – JBossTS uses the default values defined in the class `com.arjuna.common.util.logging.DebugLevel`

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

  Additional Debugging Values are:

- `CONSTRUCTORS`: Diagnostics from constructors.
- `DESTRUCTORS`: Diagnostics from finalizers.
- `CONSTRUCT_AND_DESTRUCT`: Diagnostics from constructors and finalizers.

- FUNCTIONS: Diagnostics from functions.
- OPERATORS: Diagnostics from operators, such as equals.
- FUNCS_AND_OPS: Diagnostics from functions and operations.
- ALL_NON_TRIVIAL: Diagnostics from all non-trivial operations.
- TRIVIAL_FUNCS: Diagnostics from trivial functions.
- TRIVIAL_OPERATORS: Diagnostics from trivial operations, and operators.
- ALL_TRIVIAL: Diagnostics from all trivial operations.

Visibility level – JBossTS uses the default values defined in the class `com.arjuna.common.util.logging.VisibilityLevel`

- VIS_NONE: No Diagnostic
- VIS_PRIVATE : only from private methods.
- VIS_PROTECTED only from protected methods.
- VIS_PUBLIC only from public methods.
- VIS_PACKAGE only from package methods.
- VIS_ALL: Full Diagnostic

Facility Code – JBossTS uses the following values defined in the class `com.arjuna.common.util.logging.VisibilityLevel`

- FAC_ATOMIC_ACTION = 0x0000001 (atomic action core module).
- FAC_BUFFER_MAN = 0x00000004 (state management (buffer) classes).
- FAC_ABSTRACT_REC = 0x00000008 (abstract records).
- FAC_OBJECT_STORE = 0x00000010 (object store implementations).
- FAC_STATE_MAN = 0x00000020 (state management and StateManager).
- FAC_SHMEM = 0x00000040 (shared memory implementation classes).
- FAC_GENERAL = 0x00000080 (general classes).
- FAC_CRASH_RECOVERY = 0x00000800 (detailed trace of crash recovery module and classes).
- FAC_THREADING = 0x00002000 (threading classes).
- AC_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 CommonLogging.xml file, to enable logging messages issued by the JBossTS module.

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 debugging level should be set appropriately. For instance, the following configuration, in the CommonLogging.xml, enables all debug messages related to the Crash Recovery protocol and issued by the JBossTS module.

```
<!--  Common logging related 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"/>
```

Note: Two logger objects are provided, one manages I18N messages and a second does not.

Setting the normal recovery messages to the INFO level allows the RecoveryManager producing 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: ERROR. Setting the level to ERROR means that the RecoveryManager will only produce error, warning or fatal 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 com.arjuna.ats.arjuna.recovery.periodicRecoveryPeriod), the RecoveryManager will call the first pass method on each property, then wait for a brief period (defined by property com.arjuna.ats.arjuna.recovery.recoveryBackoffPeriod), 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.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 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 the periodicRecoveryPeriod, recoveryBackoffPeriod 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 (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 – periodicRecoveryPeriod should be set accordingly (default is 120 seconds). The recoveryBackoffPeriod 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 "call to the original process to see if they are still in progress

Note: In previous versions of TxCore there was no contact mechanism, and the backoff period had 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 *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 that begin with "com.arjuna.ats.arjuna.recovery.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 Recovery Extension settings are:

```
com.arjuna.ats.arjuna.recovery.recoveryExtension1 =
com.arjuna.ats.internal.ts.arjuna.recovery.AtomicActionRecoveryModule

com.arjuna.ats.arjuna.recovery.recoveryExtension2 =
com.arjuna.ats.txoj.recovery.TORecoveryModule
```

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 interface. Implementations of this interface are loaded by giving the class name as the value of a property whose name begins with "com.arjuna.ats.arjuna.recovery.expiryScanner". The RecoveryManager calls the scan() method on each loaded Expiry Scanner implementation at an interval determined by the property "com.arjuna.ats.arjuna.recovery.expiryScanInterval". This value is given in *hours* – default is 12. An expiryScanInterval 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)

The kinds of item that are scanned for expiry are:

TransactionStatusManager items : one of these is created by every application process that uses TxCore – 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 com.arjuna.ats.arjuna.recovery.transactionStatusManagerExpiryTime (in hours – default is

12, zero means never expire). The expiry time should be greater than the lifetime of any single *JBossTS*-using process.

The Expiry Scanner properties for these are:

```
com.arjuna.ats.arjuna.recovery.expiryScannerTransactionStatusManager =
com.arjuna.ats.internal.ts.arjuna.recovery.ExpiredTransactionStatusManagerS
canner
```

ORB specific configurations

JacORB

For JacORB to function correctly it is necessary to ensure there is a valid `jacorb.properties` or `.jacorb_properties` file in one of the following places:

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

The above places are searched in the order given. A template `jacorb.properties` file can be found in the JacORB installation directory.

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

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

These properties specify the minimum and maximum number of request processing threads that JacORB will use in its thread pool. If there aren't a sufficient number of threads available in this thread pool then the application may appear to become deadlocked. For more information on configuring JacORB please reference the JacORB documentation.

Note: JacORB 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 JacORB jars.

When running the recovery manager it is important that it always uses the same well known port for each machine on which it runs. You should not use the OAPort property provided by JacORB unless the recovery manager has its own jacorb.properties file or this 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, then you should use the `com.arjuna.ats.jts.recoveryManagerPort` property.

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
```

Note: 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.

Initialising *JBossTS* applications

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

Errors and Exceptions

In this section we shall cover the types of errors and exceptions which may be thrown or reported during a transactional application and give probable indications of their causes.

- `NO_MEMORY`: the application has run out of memory (thrown an `OutOfMemoryError`) and *JBossTS* has attempted to do some cleanup (by running the garbage collector) before re-throwing the exception as a standard CORBA exception. This is probably a transient problem and retrying the invocation should succeed.
- `com.arjuna.ats.arjuna.exceptions.FatalError`: an error has occurred which means that the transaction system must shut down. Prior to this error being thrown the transaction service will have ensured that all running transactions have rolled back. If caught, the application should tidy up and exit. If further work is attempted, application consistency may be violated.
- `com.arjuna.ats.arjuna.exceptions.LicenceError`: an attempt has been made to use the transaction service in a manner inconsistent with the current license. The transaction service will not allow further forward progress for existing or new transactions.
- `com.arjuna.ats.arjuna.exceptions.ObjectStoreError`: an error occurred while the transaction service attempted to use the object store. Further forward progress is not possible.
- `com.arjuna.ats.jts.exceptions.OTS_Error`: an error occurred within the transaction service core which means that further progress cannot be made.

- Object store warnings about access problems on states may occur during the normal execution of crash recovery. This is the result of multiple concurrent attempts to perform recovery on the same transaction. It can be safely ignored.