JBoss Transactions API 4.2.2

Administration Guide

JBTA-AG-10/19/06

Legal Notices

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

Arjuna Technologies Limited 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. Arjuna Technologies Limited 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.

Copyright

Software Version

JBoss Transactions API 4.2.2

Restricted Rights Legend

Contents

About This Guide

What This Guide Contains

The Administration Guide contains information on how to administer JBoss Transactions API 4.2.2.

Audience

This guide is most relevant to engineers who are responsible for administration of JBoss Transactions API 4.2.2 installations.

Prerequisites

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

Organization

This guide contains the following chapters:

- **Chapter 1, Administration of ArjunaTA:** describes how to administer ArjunaTA, mainly by selecting the variant of JTA implementation: pure local or remote (allowing distributed transactions)**.**

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 API 4.2.2 documentation set:

- JBoss Transactions API 4.2.2 *Release Notes*: Provides late-breaking information about JBoss Transactions API 4.2.2.
- JBoss Transactions API 4.2.2 *Installation Guide*: This guide provides instructions for installing JBoss Transactions API 4.2.2.
- JBoss Transactions API 4.2.2 *Programmer's Guide*: Provides guidance for writing applications.

Contacting Us

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

Chapter 1

Administration of ArjunaTA

Introduction

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

- The present implementation of the *ArjunaTA* system provides no security or protection for data. The objects stored in the *ArjunaTA* 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 *ArjunaTA* 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.

ObjectStore management

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 realise this and do not simply delete the contents of the object store assuming it is an exclusive resource.

ArjunaTA runtime information

Each module that comprises *ArjunaTA* 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>
```

Failure recovery administration

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

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

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

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

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 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. ArjunaTA uses the default Debugging level and the default Visibility level provided by CLF, but it defines its own Facility Code. ArjunaTA 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 ArjunaTA are defined below:

Debugging level – ArjunaTA 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 – ArjunaTA 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 – ArjunaTA 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 ArjunaTA 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 ArjunaTA 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 ArjunaCore 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 *ArjunaTA*, 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 ArjunaCore – 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 *ArjunaTA*-using process.

The Expiry Scanner properties for these are:

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

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 *ArjunaTA* has attempted to do some cleanup (by running the garbage collector) before re-throwing the 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 licence. 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.
- 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.

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

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

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.