ArjunaCore 3.0

Failure Recovery Guide

AC-FRG-8/4/03

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.

Software Version

ArjunaCore 3.0

Restricted Rights Legend

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

Arjuna Technologies Limited
Nanotechnology Centre
Herschel Building
Newcastle Upon Tyne
NE1 7RU
United Kingdom

Contents

About This Guide

What This Guide Contains

The Failure Recovery Guide contains information on how to use ArjunaCore 3.0.

Audience

This guide is most relevant to engineers who are responsible for administering ArjunaCore 3.0 installations.

Prerequisites

You should have installed ArjunaCore 3.0.

Organization

This guide contains the following chapters:

- **Chapter 1, Architecture of the Recovery Manager:** explains the internal architecture of the Recovery Manager.

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 ArjunaCore 3.0 documentation set:

- ArjunaCore 3.0 *Release Notes*: Provides late-breaking information about ArjunaCore 3.0.
- ArjunaCore 3.0 *Installation Guide*: This guide provides instructions for installing ArjunaCore 3.0.
- ArjunaCore 3.0 *Users Guide*: Provides guidance for writing applications.

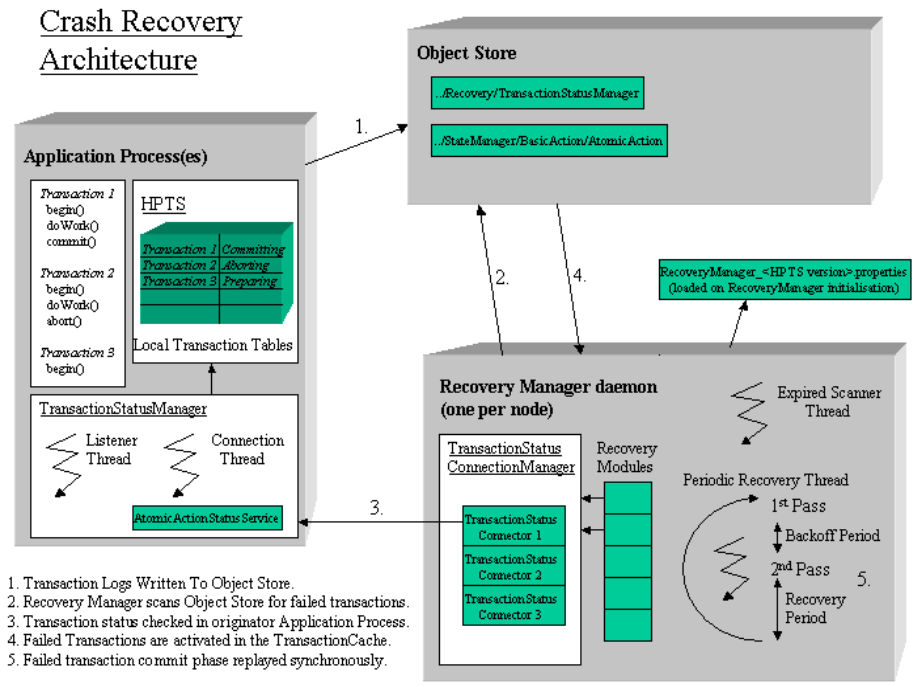
Contacting Us

Questions or comments about ArjunaCore 3.0 should be directed to our support team. Send email to **`support@arjuna.com`**.

Architecture Of the Recovery Manager

Crash Recovery Overview

The main architectural components within Crash Recovery are illustrated in the diagram below:



The Recovery Manager is a daemon process responsible for performing crash recovery. Only one Recovery Manager runs per node. The Object Store provides persistent data storage for transactions to log data. During normal transaction processing each transaction will log persistent data needed for the commit phase to the Object Store. On successfully committing a transaction this data is removed, however if the transaction fails then this data remains within the Object Store.

The Recovery Manager functions by:

- Periodically scanning the Object Store for transactions that may have failed. Failed transactions are indicated by the presence of log data after a period of time that the transaction would have normally been expected to finish.
- Checking with the application process which originated the transaction whether the transaction is still in progress or not.
- Recovering the transaction by re-activating the transaction and then replaying phase two of the commit protocol.

The following sections describe the architectural components in more detail.

Recovery Manager

On initialisation the Recovery Manager first loads in configuration information via a properties file. This configuration includes a number of recovery activators and recovery modules, which are then dynamically loaded.

Each recovery activator is used to instantiate a recovery class related to the underlying communication protocol. Indeed, since the version 3.0 of ArjunaTS, the Recovery Manager is not specifically tied to an Object Request Broker or ORB, that is to specify a recovery instance able to manage the OTS recovery protocol the new interface RecoveryActivator is provided to identify specific transaction protocol. For instance, when used with OTS, the RecoveryActivitor has the responsibility to create a RecoveryCoordinator object able to respond to the replay_completion operation.

All RecoveryActivator instances inherit the same interface. They are loaded via the following recovery extension property:

```
<property
   name="com.arjuna.ats.arjuna.recovery.recoveryActivator_<number>"
    value="RecoveryClass"/>
```

For instance the RecoveryActivator provided in the distribution of JTS/OTS, which shall not be commented, is as follow:

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

When loaded all RecoveryActivator instances provide the method startRCservice invoked by the Recovery Manager and used to create the appropriate Recovery Component able to receive recovery requests according to a particular transaction protocol. For instance the RecoveryCoordinator for the OTS protocol.

Each recovery module is used to recover a different type of transaction/resource, however each recovery module inherits the same basic behaviour.

Recovery consists of two separate passes/phases separated by two timeout periods. The first pass examines the object store for potentially failed transactions; the second pass performs crash recovery on failed transactions. The timeout between the first and second pass is known as the backoff period. The timeout between the end of the second pass and the start of the first pass is the recovery period. The recovery period is larger than the backoff period.

The Recovery Manager invokes the first pass upon each recovery module, applies the backoff period timeout, invokes the second pass upon each recovery module and finally applies the recovery period timeout before restarting the first pass again.

The recovery modules are loaded via the following recovery extension property:

```
com.arjuna.ats.arjuna.recovery.recoveryExtension<number>=<RecoveryClass>
```

The backoff period and recovery period are set using the following properties:

```
com.arjuna.ats.arjuna.recovery.recoveryBackoffPeriod  (default 10 secs)
com.arjuna.ats.arjuna.recovery.periodicRecovery       (default 120 secs)
```

The following java classes are used to implement the Recovery Manager:

package com.arjuna.ats.arjuna.recovery :

RecoveryManager

Daemon process which starts up by instanciating an instance of the RecoveryManagerImple class.

RecoveryEnvironment - Properties used by the recovery manager.

RecoveryConfiguration - Specifies the name of the Recovery Manager property file.

(ie RecoveryManager-properties.xml)

package com.arjuna.ats.internal.ts.arjuna.recovery :

RecoveryManagerImple

Creates and starts instances of the RecActivatorLoader, the PeriodicRecovery thread and the ExpiryEntryMonitor thread.

RecActivatorLoader

Dynamically loads in the RecoveryActivator specified in the Recovery Manager property file. Each RecoveryActicator is specified as a recovery extension in the properties file

PeriodicRecovery

Thread which loads each recovery module, then calls the first pass method for each module, applies the backoff period timeout, calls the second pass method for each module and applies the recovery period timeout.

RecoveryClassLoader

Dynamically loads in the recovery modules specified in the Recovery Manager property file. Each module is specified as a recovery extension in the properties file

(ie.

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

Recovery Modules

There are two recovery modules used in the ArjunaTS. One for recovering Atomic Action transactions and one to recover Transactional Objects for java. The following pseudo code describes the basic algorithm used for Atomic Action transactions and Transactional Objects for java.

AtomicAction pseudo code

First Pass:

< create a transaction vector for transaction Uids. >

< read in all transactions for a transaction type AtomicAction. >

while < there are transactions in the vector of transactions. >

do

< add the transaction to the vector of transactions. >

end while.

Second Pass:

while < there are transactions in the transaction vector >

do

```
  if < the intention list for the transaction still exists >
  then
    < create new transaction cached item >
    < obtain the status of the transaction >

    if < the transaction is not in progress >
    then
      < replay phase two of the commit protocol >
    endif.
  endif.
end while.
```

Transactional Object pseudo code

```
  First Pass:
< Create a hash table for uncommitted transactional objects. >
< Read in all transactional objects within the object store. >
while < there are transactional objects >
do
  if < the transactional object has an Uncommited status in the object store >
  then
    < add the transactional Object o the hash table for uncommitted transactional objects >
  end if.
end while.
```

Second Pass:

while < there are transactions in the hash table for uncommitted transactional objects >

do

if < the transaction is still in the Uncommitted state >

then

if < the transaction is not in the Transaction Cache >

then

< check the status of the transaction with the original application process >

if < the status is Rolled Back or the application process is inactive >

< rollback the transaction by removing the Uncommitted status from the Object Store >

endif.

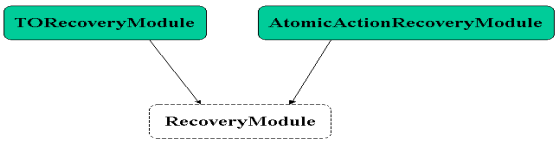
endif.

endif.

end while.

Recovery Module Classes

The classes used for recovery modules are illustrated below:



com.arjuna.ats.arjuna.recovery.RecoveryModule

Interface with the periodicWorkFirstPass and periodicWorkSecondPass methods.

com.arjuna.ats.internal.arjuna.recovery.AtomicActionRecoveryModule

Recovers AtomicAction transactions.

com.arjuna.ats.internal.txoj.recovery.TORecoveryModule

Recovers Transactional Objects for Java.

com.arjuna.ats.internal.jts.recovery.transactions.TransactionRecoveryModule

Recovers JTS Transactions. This is a generic class from which TopLevel and Server transaction recovery modules inherit, respectively

com.arjuna.ats.internal.jts.recovery.transactions.TopLevelTransactionRecoveryModule

com.arjuna.ats.internal.jts.recovery.transactions.ServerTransactionRecoveryModule

TransactionStatusConnectionManager

The TransactionStatusConnectionManager object is used by the recovery modules to retrieve the status of transactions and acts like a proxy for TransactionStatusManager objects. It maintains a table of TransactionStatusConnector obects each of which connects to a TransactionStatusManager object in an Application Process.

The transactions status is retrieved using the getTransactionStatus methods which take a transaction Uid and if available a transaction type as parameters. The process Uid field in the transactions Uid parameter is used to lookup the target TransactionStatusManagerItem host/port pair in the Object Store. The host/port pair are used to make a TCP connection to the target TransactionStatusManager object by a TransactionStatusConnector object. The TransactionStatusConnector passes the transaction Uid/transaction type to the TransactionStatusManager in order to retrieve the transactions status.

Expired Scanner Thread

When the Recovery Manager initialises an expiry scanner thread ExpiryEntryMonitor is created which is used to remove long dead items from the ObjectStore. A number of scanner modules are dynamically loaded which remove long dead items for a particular type.

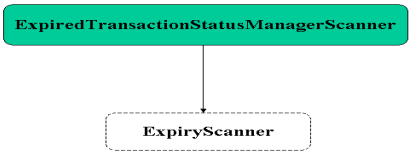
Scanner modules are loaded at initialisation and are specified as properties beginning with

```
com.arjuna.ats.arjuna.recovery.expiryScanner<Scanner Name>=<Scanner
Class>
```

All the scanner modules are called periodically to scan for dead items by the ExpiryEntryMonitor thread. This period is set with the property:

```
com.arjuna.ats.arjuna.recovery.expiryScanInterval
```

All scanners inherit the same behaviour from the java interface ExpiryScanner as illustrated in diagram below:



A scan method is provided by this interface and implemented by all scanner modules, this is the method that gets called by the scanner thread.

The ExpiredTransactionStatusManagerScanner removes long dead TransactionStatusManagerItems from the Object Store. These items will remain in the Object Store for a period of time before they are deleted. This time is set by the property:

```
com.arjuna.ats.arjuna.recovery.transactionStatusManagerExpiryTime
(default 12 hours)
```

Application Process

This represents the user transactional program. A Local transaction (hash) table, maintained within the running application process keeps trace of the current status of all transactions created by that application process, The Recovery Manager needs access to the transaction tables so that it can determine whether a transaction is still in progress, if so then recovery does not happen.

The transaction tables are accessed via the TransactionStatusManager object. On application program initialisation the host/port pair that represents the TransactionStatusManager is written to the Object Store in '../Recovery/TransactionStatusManager' part of the Object Store file hierarchy and identified by the process Uid of the application process.

The Recovery Manager uses the TransactionStatusConnectionManager object to retrieve the status of a transaction and a TransactionStatusConnector object is used to make a TCP connection to the TransactionStatusManager.

TransactionStatusManager

This object acts as an interface for the Recovery Manager to obtain the status of transactions from running HPTS application processes. One TransactionStatusManager is created per application process by the class com.arjuna.ats.arjuna.coordinator.InitAction. Currently a tcp connection is used for communication between the RecoveryManager and TransactionStatusManager. Any free port is used by the TransactionStatusManager by default, however the port can be fixed with the property:

com.arjuna.ats.arjuna.recovery.transactionStatusManagerPort

On creation the TransactionStatusManager obtains a port which it stores with the host in the Object Store as a TransactionStatusManagerItem. A Listener thread is started which waits for a connection request from a TransactionStatusConnector. When a connection is established a Connection thread is created which runs a Service (AtomicActionStatusService) which accepts a transaction Uid and a transaction type (if available) from a TransactionStatusConnector, the transaction status is obtained from the local thransaction table and returned back to the TransactionStatusConnector.

Object Store

All objects are stored in a file path which is equivalent to their class inheritance. Thus AtomicAction transactions are stored in file path ../StateManager/BasicAction/AtomicAction.

All objects are identified by a unique identifier Uid. One of the values of which is a process id in which the object was created. The Recovery Manager uses the process id to locate transaction status manager items when contacting the originator application process for the transaction status.

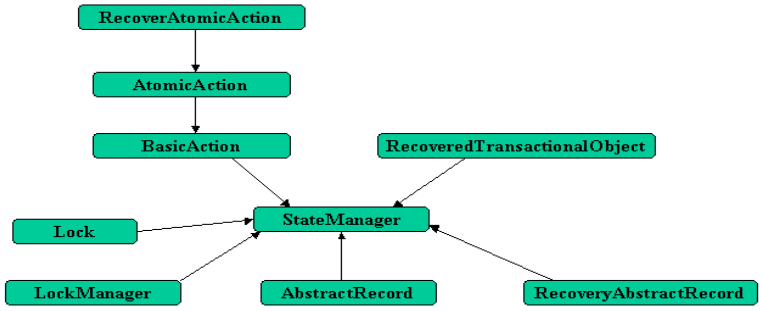
ArjunaTS Recovery

The class hierarchy within the ArjunaTS is illustrated below. StateManager is the root class for persistence objects and BasicAction is the root class for transactions.

AtomicAction transactions are recovered with the RecoverAtomicAction class.

Transactional Objects are recovered with the RecoveredTransactionalObject class.

All classes based on the AbstractRecord class are recovered with the RecoveryAbstractRecord class.



Core Classes

StateManager

This class performs persistence; it contains a number of operations pertinent to crash recovery, namely:

activate - this method re-activates the persistent object.

restore_state - this method restores the persistent state data saved with the save_state method.

Comment: But crash recovery does not call this method directly. If it did then it would break the model.

type - returns a string which represents the path within the Object Store for this persistent object.

get_uid - returns the Uid which uniquely identifies this persistent object.

unpackHeader - returns state data specific for crash recovery like the process Uid and transaction id.

BasicAction

This class performs transactions; it contains a number of lists which together comprise the intentions list.

pendingList - contains records upon which prepare is to be called or for which prepare has failed.

preparedList - contains records which have successfully prepared.

readonlyList - contains records which are read-only and prepared successfully.

failedList - contains records which have failed during commit

heuristicList - contains records during commit.

Comment: Not correct.

The first phase of commit calls prepare on the pending list. The first prepare that fails will result in the method phase2Abort being called. If phase one is successful then the intentions list will be saved to the object store before the second phase is applied.

The method phase2Commit is called to commit records on the preparedList. All records which successfully commit have their state data removed from the Object Store. Other records are either added to the failedList or heuristicList and their state data is not removed from the Object Store.

The operations of this class which are pertinent to crash recovery are:

activate - this method re-activates the transaction and all uncommitted records.

restore_state - this method restores the persistent state data saved with the save_state method.

type - returns a string which represents the path within the Object Store for this transaction.

getSavingUid - returns the Uid which uniquely identifies this transaction in the transaction log.

phase2Commit - called to during phase 2 of the commit protocol on the transaction.

Crash Recovery works by re-activating the transaction via the activate method, which calls the restore_state method which will re-constitute the failedList and heuristicList on the preparedList and then calling the phase2Commit method.

AtomicAction

This is the user level class used by the application programmer for transactions. It provides thread safety/scoping for the BasicAction class from which it inherits. The operations pertinent to crash recovery are:

type - returns a string which represents the path within the Object Store for this AtomicAction.

All other crash recovery operations are inherited from the superclass BasicAction.

AbstractRecord

All records which sub-class this class populate the intentions list via the add method in BasicAction. The methods used during crash recovery are:

topLevelCommit - for committing a top level transaction.

nestedCommit - for committing a nested transaction. Nested transactions do not require failure recovery mechanisms.

restore_state - this method restores the persistent state data saved with the save_state method.

type - returns a string which represents the path within the Object Store for this abstract record.

typeIs - record type

Phase one of the commit protocol will persist any state data associated with the record to the Object Store using the save_state operation. During phase two of the commit protocol any unsuccessfully commited records are added to failed or heuristic lists.

Crash recovery uses the RecoveryAbstractRecord class to recover abstract records. This class creates a new instance of the abstract record using a class that has been registered with the static inventory. The restore_state method is called on the abstract record before it is inserted into the prepared list.

Recovery Classes

RecoverAtomicAction

The class com.arjuna.ats.arjuna.recovery.RecoverAtomicAction when created will re-activate the transaction with the activate operation. If successfully activated then the status of the of the transaction retrieved from the TransactionStatusManager and passed as a parameter to the constructor determines whether the phase2Commit or phase2Abort operation is called.

The operations provided for crash recovery are:

replayPhase2 - called to replay phase 2 of the commit protocol on the transaction.

RecoveredTransactionalObject

The class com.arjuna.ats.internal.ts.txoj.recovery.RecoveredTransactionalObject when created instantiates a TransactionStatusConnectionManager proxy object. The operations provided for crash recovery are:

replayPhase2 - called to replay phase 2 of the commit protocol on the transaction.

findHoldingTransaction - determines if transaction has an Uncommitted state in the Object Store.

rollback - removes the transactions Uncommitted state from the Object Store.

The replayPhase2 operation firstly checks that the associated transaction Object Store state is uncommitted using the findHoldingTransaction operation. If this is so, then the status of the transaction in the originator Application Process is obtained via the TransactionStatusConnectionManager proxy operation getTransactionStatus. If the status can not be retrieved or the status returned is Aborted then the transaction state is removed with the rollback operation.

RecoveryAbstractRecord

All records are recovered using this class. The methods used during crash recovery are:

topLevelCommit - for committing a top level transaction.

nestedCommit - for committing a nested transaction.

restore_state - this method restores the persistent state data saved with the save_state method.

type - returns a string which represents the path within the Object Store for this abstract record.

typeIs - record type

During transaction recovery, the restore_state method of the BasicAction restores all records by creating an instance of the RecoveryAbstractRecord class and calling its restore_state method. All records are restored upon the pendingList.

The topLevelCommit method is called for each record by the phase2Commit method.

Chapter 2

Recovery in ArjunaTS

Recovery Protocol in OTS - Overview

To manage recovery in case of failure, the OTS specification has defined a recovery protocol. Transaction's participants in a doubt status could use the RecoveryCoordinator to determine the status of the transaction. According to that transaction status, those participants can take appropriate decision either by roll backing or committing.

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

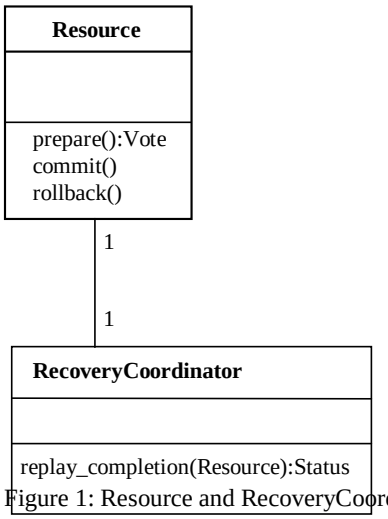


Figure 1: Resource and RecoveryCoordinator relationship.

RecoveryCoordinator in ArjunaTS

Rather than to create a RecoveryCoordinator object with its associated servant on each register_resource, ArjunaTS enhances performance by avoiding the creation of servants but it relies on a default RecoveryCoordinator object with it's associated default servant to manage all replay_completion invocations.

The default RecoveryCoordinator in Orbix

- Explain the domain Orb Name

- The default POA
- Port numbe …..
-

The default RecoveryCoordinator in JacOrb

…….