ArjunaCore 4.15.0

ArjunaCore Failure Recovery Guide

Failure Recovery for TxCore and TXOJ

Mark Little

ArjunaCore 4.15.0 ArjunaCore Failure Recovery Guide
Failure Recovery for TxCore and TXOJ
Edition 0

Author Mark Little *mlittle@redhat.com*

The ArjunaCore Failure Recovery Guide contains information on how to use JBoss Transaction Service to develop applications that use transaction technology to manage business processes.

Preface **v**
1. Document Conventions v
    1.1. Typographic Conventions v
    1.2. Pull-quote Conventions vi
    1.3. Notes and Warnings vii
2. We Need Feedback! vii

1. Introduction **1**
1.1. Embedding the Recovery Manager 1
    1.1.1. Additional Recovery Module Classes 1
1.2. Understanding Recovery Modules 1
    1.2.1. The Recovery Manager 2
    1.2.2. Configuring the Recovery Manager 2
    1.2.3. Periodic Recovery 2
    1.2.4. Expired entry removal 3
1.3. Writing a Recovery Module 5
    1.3.1. A basic scenario 5
    1.3.2. Another scenario 9

A. Revision History **11**

Preface

1. Document Conventions

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

In PDF and paper editions, this manual uses typefaces drawn from the *Liberation Fonts*[1] set. The Liberation Fonts set is also used in HTML editions if the set is installed on your system. If not, alternative but equivalent typefaces are displayed. Note: Red Hat Enterprise Linux 5 and later includes the Liberation Fonts set by default.

1.1. Typographic Conventions

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

`Mono-spaced Bold`

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

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

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

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

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

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

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

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

Proportional Bold

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

> Choose **System** → **Preferences** → **Mouse** from the main menu bar to launch **Mouse Preferences**. In the **Buttons** tab, click the **Left-handed mouse** check box and click

---

[1] https://fedorahosted.org/liberation-fonts/

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

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

`Mono-spaced Bold Italic` or ***`Proportional Bold Italic`***

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

> To connect to a remote machine using ssh, type **`ssh`** ***`username@domain.name`*** at a shell prompt. If the remote machine is **`example.com`** and your username on that machine is john, type **`ssh john@example.com`**.
>
> The **`mount -o remount`** ***`file-system`*** command remounts the named file system. For example, to remount the **`/home`** file system, the command is **`mount -o remount /home`**.
>
> To see the version of a currently installed package, use the **`rpm -q`** ***`package`*** command. It will return a result as follows: ***`package-version-release`***.

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

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

> Publican is a *DocBook* publishing system.

1.2. Pull-quote Conventions

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

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

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

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

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

import javax.naming.InitialContext;
```

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

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

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

1.3. Notes and Warnings

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

Note

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

Important

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

Warning

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

2. We Need Feedback!

If you find a typographical error in this manual, or if you have thought of a way to make this manual better, we would love to hear from you! Please submit a report in the JBoss Issue Tracker: *https://jira.jboss.org/* against the product **JBoss Transactions.**

When submitting a bug report, be sure to mention the manual's identifier: *ArjunaCore_Failure_Recover_Guide*

If you have a suggestion for improving the documentation, try to be as specific as possible when describing it. If you have found an error, please include the section number and some of the surrounding text so we can find it easily.

Introduction

In this chapter we shall cover information on failure recovery that is specific to TxCore, TXOJ or using JBossTS outside the scope of a supported application server.

1.1. Embedding the Recovery Manager

In some situations it may be required to embed the RecoveryManager in the same process as the transaction service. In this case you can create an instance of the RecoveryManager through the manager method on com.arjuna.ats.arjuna.recovery.RecoveryManager. A RecoveryManager can be created in one of two modes, selected via the parameter to the manager method:

- i. INDIRECT_MANAGEMENT: the manager runs periodically but can also be instructed to run when desired via the scan operation or through the RecoveryDriver class to be described below.
- ii. DIRECT_MANAGEMENT: the manager does not run periodically and must be driven directly via the scan operation or RecoveryDriver.

Warning

By default, the recovery manager listens on the first available port on a given machine. If you wish to control the port number that it uses, you can specify this using the com.arjuna.ats.arjuna.recovery.recoveryPort attribute.

1.1.1. Additional Recovery Module Classes

JBossTS provides a set of recovery modules that are responsible to manage recovery according to the nature of the participant and its position in a transactional tree. The provided classes over and above the ones covered elsewhere (that all implements the RecoveryModule interface) are:

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

  Recovers Transactional Objects for Java.

1.2. Understanding Recovery Modules

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.

Warning

If the ObjectStore is destroyed or modified, recovery may not be possible.

Until the recovery procedures are complete, resources affected by a transaction that was in progress at the time of the failure may be inaccessible. For database resources, this may be reported as tables or rows held by “in-doubt transactions”. For TransactionalObjects for Java resources, an attempt to activate the Transactional Object (as when trying to get a lock) will fail.

1.2.1. 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 file is located in 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
```

1.2.2. 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).

1.2.3. 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 JBossTS 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 RecoveryEnvironmentBean.recoveryExtensions. 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:

Example 1.1. Recovery Environment Bean XML

```
<entry key="RecoveryEnvironmentBean.recoveryExtensions">
  com.arjuna.ats.internal.arjuna.recovery.AtomicActionRecoveryModule
  com.arjuna.ats.internal.txoj.recovery.TORecoveryModule
</entry>
```

1.2.4. 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 names as the value of a property RecoveryEnvironmentBean.expiryScanners. The RecoveryManager calls the scan() method on each loaded Expiry Scanner implementation at an interval determined by the property RecoveryEnvironmentBean.expiryScanInterval". This value is given in hours – 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 JBossTS – 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:

Example 1.2. Recovery Environment Bean XML

```
<entry key="RecoveryEnvironmentBean.expiryScanners">
  com.arjuna.ats.internal.arjuna.recovery.ExpiredTransactionStatusManagerScanner
</entry>
```

To illustrate the behavior of a recovery module, the following pseudo code describes the basic algorithm used for Atomic Action transactions and Transactional Objects for java.

Example 1.3. 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.
```

Example 1.4. 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.
```

1.3. Writing a Recovery Module

In order to recover from failure, we have seen that the Recovery Manager contacts recovery modules by invoking periodically the methods periodicWorkFirstPass and periodicWorkSecondPass. Each Recovery Module is then able to manage recovery according the type of resources that need to be recovered. The JBoss Transaction product is shipped with a set of recovery modules (TOReceveryModule, XARecoveryModule…), but it is possible for a user to define its own recovery module that fit his application. The following basic example illustrates the steps needed to build such recovery module

1.3.1. A basic scenario

This basic example does not aim to present a complete process to recover from failure, but mainly to illustrate the way to implement a recovery module.

The application used here consists to create an atomic transaction, to register a participant within the created transaction and finally to terminate it either by commit or abort. A set of arguments are provided:

- to decide to commit or abort the transaction,
- to decide generating a crash during the commitment process.

The code of the main class that control the application is given below

Example 1.5. TestRecoveryModule.java

```
package com.arjuna.demo.recoverymodule;

import com.arjuna.ats.arjuna.AtomicAction;
import com.arjuna.ats.arjuna.coordinator.*;

public class TestRecoveryModule {
 public static void main(String args[]) {
  try {
   AtomicAction tx = new AtomicAction();
   tx.begin(); // Top level begin

   // enlist the participant
   tx.add(SimpleRecord.create());

   System.out.println("About to complete the transaction ");
   for (int i = 0; i < args.length; i++) {
    if ((args[i].compareTo("-commit") == 0))
     _commit = true;
    if ((args[i].compareTo("-rollback") == 0))
     _commit = false;
    if ((args[i].compareTo("-crash") == 0))
     _crash = true;
   }
   if (_commit)
    tx.commit(); // Top level commit
   else
    tx.abort(); // Top level rollback
  } catch (Exception e) {
   e.printStackTrace();
  }
 }

 protected static boolean _commit = true;
 protected static boolean _crash = false;
}
```

The registered participant has the following behavior:

- During the prepare phase, it writes a simple message - “I’m prepared”- on the disk such The message is written in a well known file
- During the commit phase, it writes another message - “I’m committed”- in the same file used during prepare
- If it receives an abort message, it removes from the disk the file used for prepare if any.
- If a crash has been decided for the test, then it crashes during the commit phase – the file remains with the message “I’m prepared”.

The main portion of the code illustrating such behavior is described hereafter.

Warning

that the location of the file given in variable filename can be changed

Example 1.6. SimpleRecord.java

```
package com.arjuna.demo.recoverymodule;

import com.arjuna.ats.arjuna.coordinator.*;
import java.io.File;

public class SimpleRecord extends AbstractRecord {
 public String filename = "c:/tmp/RecordState";

 public SimpleRecord() {
  System.out.println("Creating new resource");
 }

 public static AbstractRecord create() {
  return new SimpleRecord();
 }

 public int topLevelAbort() {
  try {
   File fd = new File(filename);
   if (fd.exists()) {
    if (fd.delete())
     System.out.println("File Deleted");
   }
  } catch (Exception ex) {
   // …
  }
  return TwoPhaseOutcome.FINISH_OK;
 }

 public int topLevelCommit() {
  if (TestRecoveryModule._crash)
   System.exit(0);
  try {
   java.io.FileOutputStream file = new java.io.FileOutputStream(
     filename);
   java.io.PrintStream pfile = new java.io.PrintStream(
     file);
   pfile.println("I'm Committed");
   file.close();
  } catch (java.io.IOException ex) {
   // ...
  }
  return TwoPhaseOutcome.FINISH_OK;
 }

 public int topLevelPrepare() {
  try {
   java.io.FileOutputStream file = new java.io.FileOutputStream(
     filename);
   java.io.PrintStream pfile = new java.io.PrintStream(
     file);
   pfile.println("I'm prepared");
   file.close();
  } catch (java.io.IOException ex) {
   // ...
  }
  return TwoPhaseOutcome.PREPARE_OK;
 }
 // …
}
```

The role of the Recovery Module in such application consists to read the content of the file used to store the status of the participant, to determine that status and print a message indicating if a recovery action is needed or not.

Example 1.7. SimpleRecoveryModule.java

```
package com.arjuna.demo.recoverymodule;

import com.arjuna.ats.arjuna.recovery.RecoveryModule;

public class SimpleRecoveryModule implements RecoveryModule {
 public String filename = "c:/tmp/RecordState";

 public SimpleRecoveryModule() {
  System.out
    .println("The SimpleRecoveryModule is loaded");
 }

 public void periodicWorkFirstPass() {
  try {
   java.io.FileInputStream file = new java.io.FileInputStream(
     filename);
   java.io.InputStreamReader input = new java.io.InputStreamReader(
     file);
   java.io.BufferedReader reader = new java.io.BufferedReader(
     input);
   String stringState = reader.readLine();
   if (stringState.compareTo("I'm prepared") == 0)
    System.out
      .println("The transaction is in the prepared state");
   file.close();
  } catch (java.io.IOException ex) {
   System.out.println("Nothing found on the Disk");
  }
 }

 public void periodicWorkSecondPass() {
  try {
   java.io.FileInputStream file = new java.io.FileInputStream(
     filename);
   java.io.InputStreamReader input = new java.io.InputStreamReader(
     file);
   java.io.BufferedReader reader = new java.io.BufferedReader(
     input);
   String stringState = reader.readLine();
   if (stringState.compareTo("I'm prepared") == 0) {
    System.out
      .println("The record is still in the prepared state");
    System.out.println("- Recovery is needed");
   } else if (stringState
     .compareTo("I'm Committed") == 0) {
    System.out
      .println("The transaction has completed and committed");
   }
   file.close();
  } catch (java.io.IOException ex) {
   System.out.println("Nothing found on the Disk");
   System.out
     .println("Either there was no transaction");
   System.out.println("or it as been rolled back");
  }
 }
}
```

The recovery module should now be deployed in order to be called by the Recovery Manager. To do so, we just need to add an entry in the the config file for the extension:

Example 1.8. Recovery Environment Bean Recovery Extensions XML

```
<entry key="RecoveryEnvironmentBean.recoveryExtenstions">
  com.arjuna.demo.recoverymodule.SimpleRecoveryModule
</entry>
```

Once started, the Recovery Manager will automatically load the listed Recovery modules.

Note

The source of the code can be retrieved under the trailmap directory of the JBossTS installation.

1.3.2. Another scenario

As mentioned, the basic application presented above does not present the complete process to recover from failure, but it was just presented to describe how the build a recovery module. In case of the OTS protocol, let's consider how a recovery module that manages recovery of OTS resources can be configured.

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. Asking the RecoveryCoordinator object to determine the status consists to invoke the replay_completion operation on the RecoveryCoordinator.

For each OTS Resource in a doubt status, it is well known which RecoveyCoordinator to invoke to determine the status of the transaction in which the Resource is involved – It's the RecoveryCoordinator returned during the Resource registration process. Retrieving such RecoveryCoordinator per resource means that it has been stored in addition to other information describing the resource.

A recovery module dedicated to recover OTS Resources could have the following behavior. When requested by the recovery Manager on the first pass it retrieves from the disk the list of resources that are in the doubt status. During the second pass, if the resources that were retrieved in the first pass still remain in the disk then they are considered as candidates for recovery. Therefore, the Recovery Module retrieves for each candidate its associated RecoveryCoordinator and invokes the replay_completion operation that the status of the transaction. According to the returned status, an appropriate action would be taken (for instance, rollback the resource is the status is aborted or inactive).

Appendix A. Revision History

Revision 1	**Tue Apr 13 2010**	**Tom Jenkinson** *`tom.jenkinson@redhat.com`*

Initial creation of book by publican