JBoss Transactions 4.6.0

Failure Recovery Guide

TX-FRG-1/26/09

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 4.6.0

Restricted Rights Legend

Contents

Table of Contents

About This Guide.....4

What This Guide Contains.....4
Audience.....4
Prerequisites.....4
Organization.....4
Documentation Conventions.....4
Additional Documentation.....5
Contacting Us.....5

Architecture of the Recovery Manager.....6

Crash Recovery Overview.....6
Recovery Manager.....7
Embedding the Recovery Manager.....9
Managing recovery directly.....9
Separate Recovery Manager.....9
In process Recovery Manager.....9
Recovery Modules.....10
JBossTS Recovery Module Classes.....10
A Recovery Module for XA Resources.....12
Assumed complete.....15
Writing a Recovery Module.....15
A basic scenario.....15
Another scenario.....19
TransactionStatusConnectionManager.....19
Expired Scanner Thread.....20
Application Process.....21
TransactionStatusManager.....21
Object Store.....22
Socket free operation.....22

How JBossTS manages the OTS Recovery Protocol.....23

Recovery Protocol in OTS - Overview.....23
RecoveryCoordinator in JBossTS.....24
Understanding POA 24
The default RecoveryCoordinator in JacOrb...26
How Does it work.....26

About This Guide

What This Guide Contains

The Failure Recovery Guide contains information on how to use JBoss Transactions 4.6.0

Audience

This guide is most relevant to engineers who are responsible for administering JBoss Transactions 4.6.0 installations.

Prerequisites

You should have installed JBoss Transactions 4.6.0

Organization

This guide contains the following chapters:

- **Chapter 1, Architecture of the Recovery Manager:** explains the internal architecture of the Recovery Manager.
- **Chapter 2, How JBossTS manages the OTS Recovery Protocol:** explains how JBossTS deals with particular features of Object Request Brokers to implement the recovery defined by the OTS specification in a optimistic way.

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 4.6.0 documentation set:

- **JBoss Transactions 4.6.0** *Release Notes*: Provides late-breaking information about JBoss Transactions 4.6.0.
- **JBoss Transactions 4.6.0** *Installation Guide*: This guide provides instructions for installing JBoss Transactions 4.6.0.
- **JBoss Transactions 4.6.0** *Users Guide*: Provides guidance for writing applications.

Contacting Us

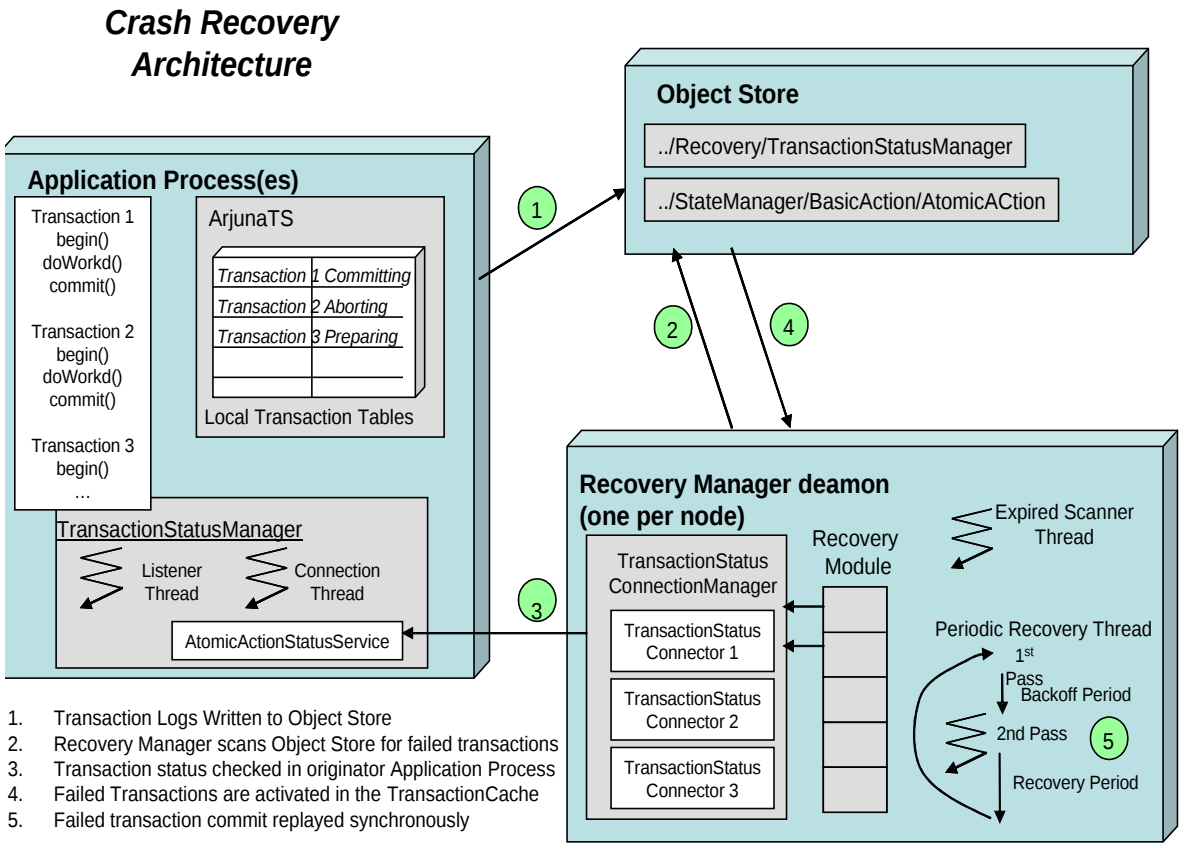
Questions or comments about JBoss Transactions 4.6.0 should be directed to our support team.

Chapter 1

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.

Architecture of the Recovery ManagerThe 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 initialization 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.

Since the version 3.0 of JBossTS, the Recovery Manager is not specifically tied to an Object Request Broker or ORB. Hence, the OTS recovery protocol is not implicitly enabled. To enable such protocol, we use the concept of recovery activator, defined with the interface RecoveryActivator, which is used to instantiate a recovery class related to the underlying communication 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 defined by the OTS protocol.

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

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 – The daemon process that starts up by instantiating 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 (i.e., com.arjuna.ats.arjuna.recovery.recoveryExtension1= com.arjuna.ats.internal.ts.arjuna.recovery.AtomicActionRecoveryModule).

Caution: 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.

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.

Managing recovery directly

As already mentioned, recovery typically happens at periodic intervals. If you require to drive recovery directly, then there are two options, depending upon how the RecoveryManager has been created.

Separate Recovery Manager

You can either use the com.arjuna.ats.arjuna.tools.RecoveryMonitor program to send a message to the Recovery Manager instructing it to perform recovery, or you can create an instance of the com.arjuna.ats.arjuna.recovery.RecoveryDriver class to do likewise. There are two types of recovery scan available:

i. ASYNC_SCAN: here a message is sent to the RecoveryManager to instruct it to perform recovery, but the response returns before recovery has completed.

ii. SYNC: here a message is sent to the RecoveryManager to instruct it to perform recovery, and the response occurs only when recovery has completed.

In process Recovery Manager

You can invoke the `scan` operation on the RecoveryManager. This operation returns only when recovery has completed. However, if you wish to have an asynchronous interaction pattern, then the `RecoveryScan` interface is provided:

```
public interface RecoveryScan
{
 public void completed ();
}
```

An instance of an object supporting this interface can be passed to the `scan` operation and its `completed` method will be called when recovery finishes. The `scan` operation returns immediately, however.

Recovery Modules

As stated before each recovery module is used to recover a different type of transaction/resource, but each recovery module must implement the following RecoveryModule interface, which defines two methods: periodicWorkFirstPass and periodicWorkSecondPass invoked by the Recovery Manager.

```
public interface RecoveryModule
{
 /**
 * Called by the RecoveryManager at start up, and then
 * PERIODIC_RECOVERY_PERIOD seconds after the completion, for all
 * RecoveryModules of the second pass
 */
 public void periodicWorkFirstPass ();

 /**
 * Called by the RecoveryManager RECOVERY_BACKOFF_PERIOD seconds
 * after the completion of the first pass
 */
 public void periodicWorkSecondPass ();
}
```

JBossTS 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 (that all implements the RecoveryModule interface) are:

- 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

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.

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.
   endif.
end while.
```

A Recovery Module for XA Resources

To manage recovery, we have seen in the previous chapter that the Recovery Manager triggers a recovery process by calling a set of recovery modules that implements the two methods defined by the RecoveryModule interface.

To enable recovery of participants controlled via the XA interface, a specific recovery module named XARecoveryModule is provided. The XARecoveryModule, defined in the packages com.arjuna.ats.internal.jta.recovery.arjunacore and com.arjuna.ats.internal.jta.recovery.jts, handles recovery of XA resources (databases etc.) used in JTA.

Caution:JBossTS supports two JTA implementations: a purely local version (no distributed transactions) and a version layered on the JTS. Recovery for the former is straightforward. In the following discussion we shall implicitly consider on the JTS implementation.

Its behavior consists of two aspects: “transaction-initiated” and “resource-initiated” recovery. Transaction-initiated recovery is possible where the particular transaction branch had progressed far enough for a JTA Resource Record to be written in the ObjectStore, as illustrated in Figure 2.

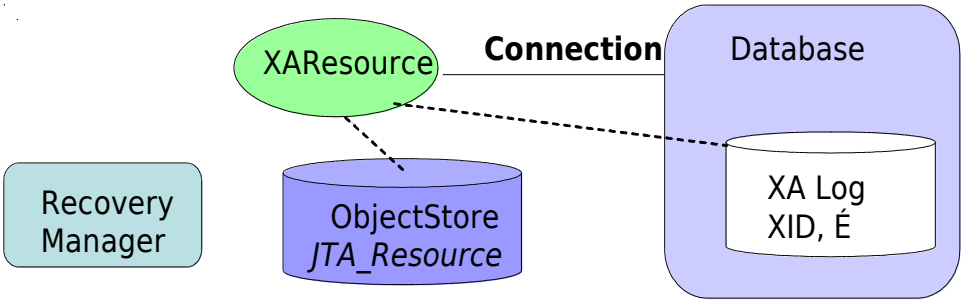


Figure 2 – JTA/JDBC information stored in the ObjectStore

A JTA Resource record contains the information needed to link the transaction, as known to the rest of JBossTS, to the database. Resource-initiated recovery is necessary for branches where a failure occurred after the database had made a persistent record of the transaction, but before the JTA ResourceRecord was persisted. Resource-initiated recovery is also necessary for datasources for which it is not possible to hold information in the JTA Resource record that allows the recreation in the RecoveryManager of the XAConnection/XAResource that was used in the original application.

Caution: When running XA recovery it is 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. The contents of `com.arjuna.ats.jta.xaRecoveryNode` should be alphanumeric and match the values of `com.arjuna.ats.arjuna.xa.nodeIdentifier`.

Transaction-initiated recovery is automatic. The XARecoveryModule finds the JTA Resource Record that need recovery, then uses the normal recovery mechanisms to find the status of the transaction it was involved in (i.e., it calls replay_completion on the RecoveryCoordinator for the transaction branch), (re)creates the appropriate XAResource and issues commit or rollback on it as appropriate. The XAResource creation will use the same information, database name, username, password etc., as the original application.

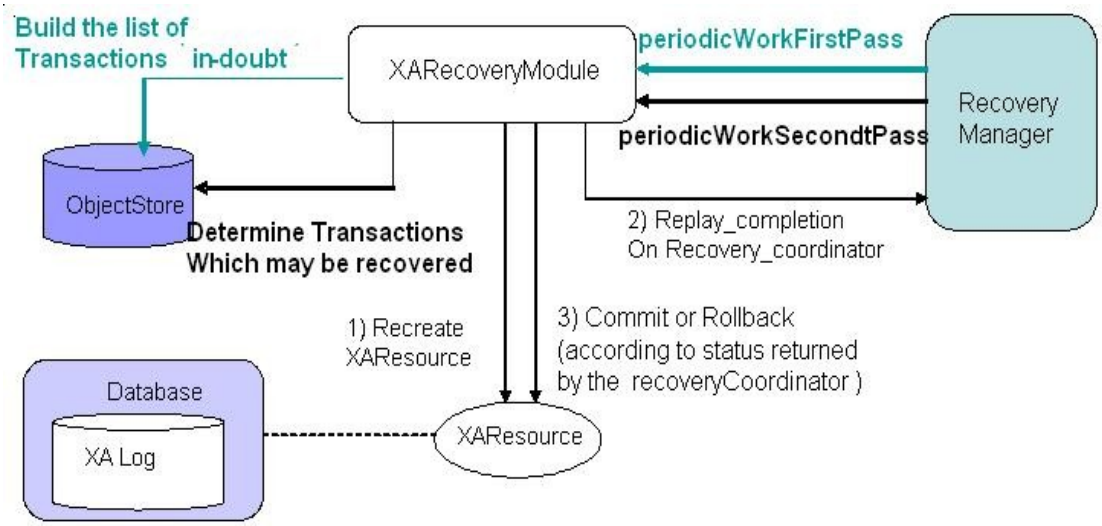


Figure 3 - Transaction-Initiated Recovery and XA Recovery

Resource-initiated recovery has to be specifically configured, by supplying the Recovery Manager with the appropriate information for it to interrogate all the databases (XADataSources) that have been accessed by any JBossTS application. The access to each XADataSource is handled by a class that implements the com.arjuna.ats.jta.recovery.XAResourceRecovery interface, as illustrated in Figure 4. Instances of classes that implements the XAResourceRecovery interface are dynamically loaded, as controlled by properties with names beginning "com.arjuna.ats.jta.recovery.XAResourceRecovery".

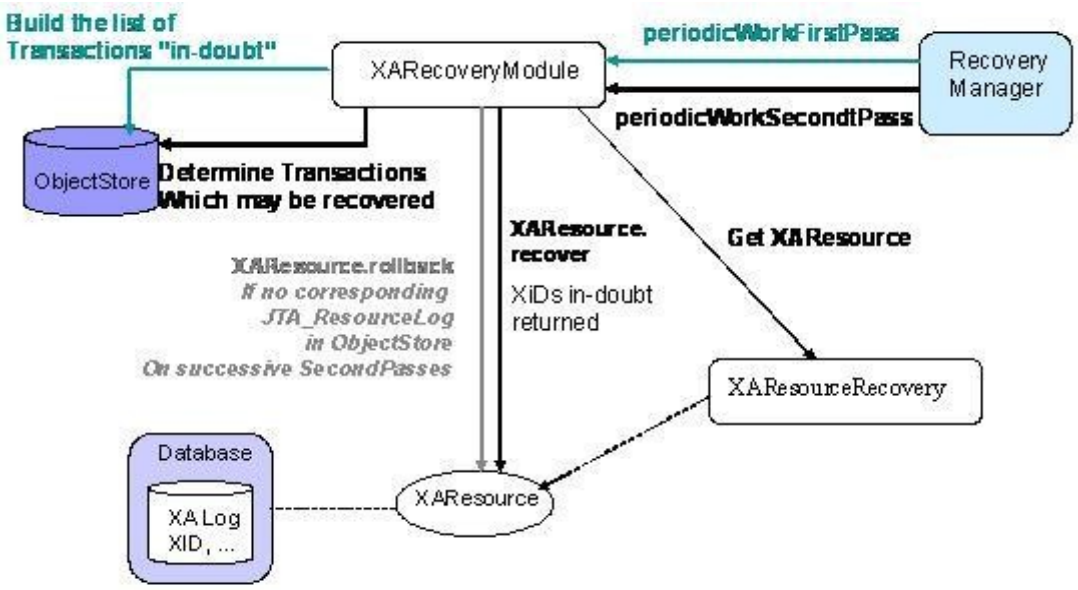


Figure 4 – Resource-initiated recovery and XA Recovery

The XARecoveryModule will use the XAResourceRecovery implementation to get a XAResource to the target datasource. On each invocation of periodicWorkSecondPass, the recovery module will issue an XAResource.recover request – this will (as described in the XA specification) return a list of the transaction identifiers (Xid's) that are known to the datasource and are in an indeterminate (in-doubt) state. The list of these in-doubt Xid's received on successive passes (i.e. periodicWorkSecondPass-es) is compared. Any Xid that appears in both lists, and for which no JTA ResourceRecord was found by the intervening transaction-initiated recovery is assumed to belong to a transaction that was involved in a crash before any JTA ResourceRecord was written, and a rollback is issued for that transaction on the XAResource.

This double-scan mechanism is used because it is possible the Xid was obtained from the datasource just as the original application process was about to create the corresponding JTA_ResourceRecord. The interval between the scans should allow time for the record to be written unless the application crashes (and if it does, rollback is the right answer).

An XAResourceRecovery implementation class can be written to contain all the information needed to perform recovery to some datasource. Alternatively, a single class can handle multiple datasources. The constructor of the implementation class must have an empty parameter list (because it is loaded dynamically), but the interface includes an `initialise` method which passes in further information as a string. The content of the string is taken from the property value that provides the class name: everything after the first semi-colon is passed as the value of the string. The use made of this string is determined by the XAResourceRecovery implementation class.

For further details on the way to implement a class that implements the interface XAResourceRecovery, read the JDBC chapter of the JTA Programming Guide. An implementation class is provided that supports resource-initiated recovery for any

XADataSource. This class could be used as a template to build your own implementation class.

Assumed complete

If a failure occurs in the transaction environment after the transaction coordinator had told the XAResource to commit but before the transaction log has been updated to remove the participant, then recovery will attempt to replay the commit. In the case of a Serialized XAResource, the response from the XAResource will enable the participant to be removed from the log, which will eventually be deleted when all participants have been committed. However, if the XAResource is not recoverable then it is extremely unlikely that any XAResourceRecovery instance will be able to provide the recovery sub-system with a fresh XAResource to use in order to attempt recovery; in which case recovery will continually fail and the log entry will never be removed.

There are two possible solutions to this problem:

- Rely on the relevant ExpiryScanner to eventually move the log elsewhere. Manual intervention will then be needed to ensure the log can be safely deleted. If a log entry is moved, suitable warning messages will be output.
- Set the com.arjuna.ats.jta.xaAssumeRecoveryComplete to true. This option is checked whenever a new XAResource instance cannot be located from any registered XAResourceRecovery instance. If false (the default), recovery assumes that there is a transient problem with the XAResourceRecovery instances (e.g., not all have been registered with the sub-system) and will attempt recovery periodically. If true then recovery assumes that a previous commit attempt succeeded and this instance can be removed from the log with no further recovery attempts. This option is global, so needs to be used with care since if used incorrectly XAResource instances may remain in an uncommitted state.

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

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

```
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.

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

```
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.

```
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 –
 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 - Either there was
 no transaction 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 RecoveryManager-properties.xml by adding a new property as follow:

```
<property
name="com.arjuna.ats.arjuna.recovery.recoveryExtension<i>"
value="com.arjuna.demo.recoverymodule.SimpleRecoveryModule"/>
```

Where <i> represent the new occurrence number that follows the last that already exists in the file. Once started, the Recovery Manager will automatically load the added Recovery module.

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

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).

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.

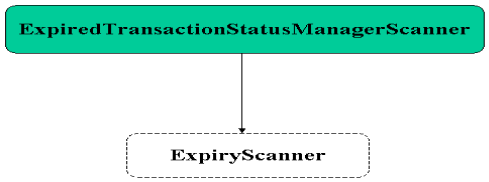
Scanner modules are loaded at initialisation and are specified as properties beginning with

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)
```

The AtomicActionExpiryScanner moves transaction logs for AtomicActions that are assumed to have completed. For instance, if a failure occurs after a participant has been told to commit but before the transaction system can update the log, then upon recovery JBossTS recovery will attempt to replay the commit request, which will obviously fail, thus preventing the log from being removed.

Note: AtomicActionExpiryScanner is disabled by default. To enable it simply add it to the JBossTS properties file.

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 JBossTS application processes. One TransactionStatusManager is created per application process by the class com.arjuna.ats.arjuna.coordinator.TxControl. 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. Therefore, exactly one recovery manager per ObjectStore must run on each nodes and ObjectStores must not be shared by multiple nodes.

Socket free operation

The use of TCP/IP sockets for TransactionStatusManager and RecoveryManager provides for maximum flexibility in the deployment architecture. It is often desirable to run the RecoveryManager in a separate JVM from the Transaction manager(s) for increased reliability. In such deployments, TCP/IP provides for communication between the RecoveryManager and transaction manager(s), as detailed in the preceding sections. Specifically, each JVM hosting a TransactionManager will run a TransactionStatusManager listener, through which the RecoveryManager can contact it to determine if a transaction is still live or not. The RecoveryManager likewise listens on a socket, through which it can be contacted to perform recovery scans on demand. The presence of a recovery listener is also used as a safety check when starting a RecoveryManager, since at most one should be running for a given ObjectStore.

There are some deployment scenarios in which there is only a single TransactionManager accessing the ObjectStore and the RecoveryManager is co-located in the same JVM. For such cases the use of TCP/IP sockets for communication introduces unnecessary runtime overhead. Additionally, if several such distinct processes are needed for e.g. replication or clustering, management of the TCP/IP port allocation can become unwieldy. Therefore it may be desirable to configure for socketless recovery operation.

The property com.arjuna.ats.arjuna.coordinator.transactionStatusManagerEnable can be set to a value of NO to disable the TransactionStatusManager for any given TransactionManager. Note that this must not be done if recovery runs in a separate process, as it may lead to incorrect recovery behavior in such cases. For an in-process recovery manager, the system will use direct access to the ActionStatusService instead.

The property com.arjuna.ats.arjuna.recovery.recoveryListener can likewise be used to disable the TCP/IP socket listener used by the recovery manager. Care must be taken not to inadvertently start multiple recovery managers for the same ObjectStore, as this error, which may lead to significant crash recovery problems, cannot be automatically detected and prevented without the benefit of the socket listener.

Chapter 2

How JBossTS manages the OTS Recovery Protocol

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.

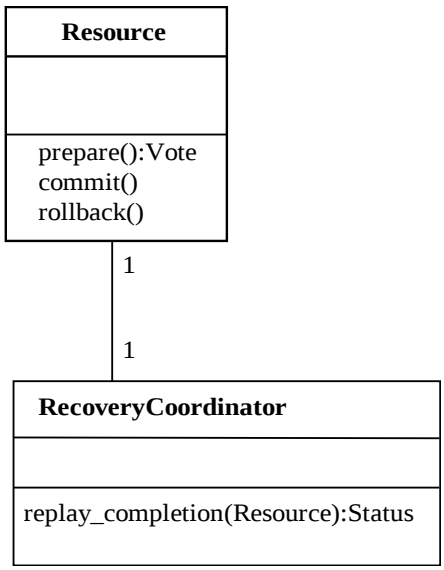


Figure 5: Resource and RecoveryCoordinator relationship.

RecoveryCoordinator in JBossTS

On each resource registration a RecoveryCoordinator Object is expected to be created and returned to the application that invoked the ***register_resource*** operation. Behind each CORBA object there should be an object implementation or Servant object, in POA terms, which performs operations made on a RecoveryCoordinator object. Rather than to create a RecoveryCoordinator object with its associated servant on each register_resource, JBossTS 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.

In the next sections we first give an overview of the Portable Object Adapter architecture, then we describe how this architecture is used to provide RecoveryCoordinator creation with optimization as explained above.

Understanding POA

Basically, the Portable Object Adapter, or POA is an object that intercepts a client request and identifies the object that satisfies the client request. The Object is then invoked and the response is returned to the client.

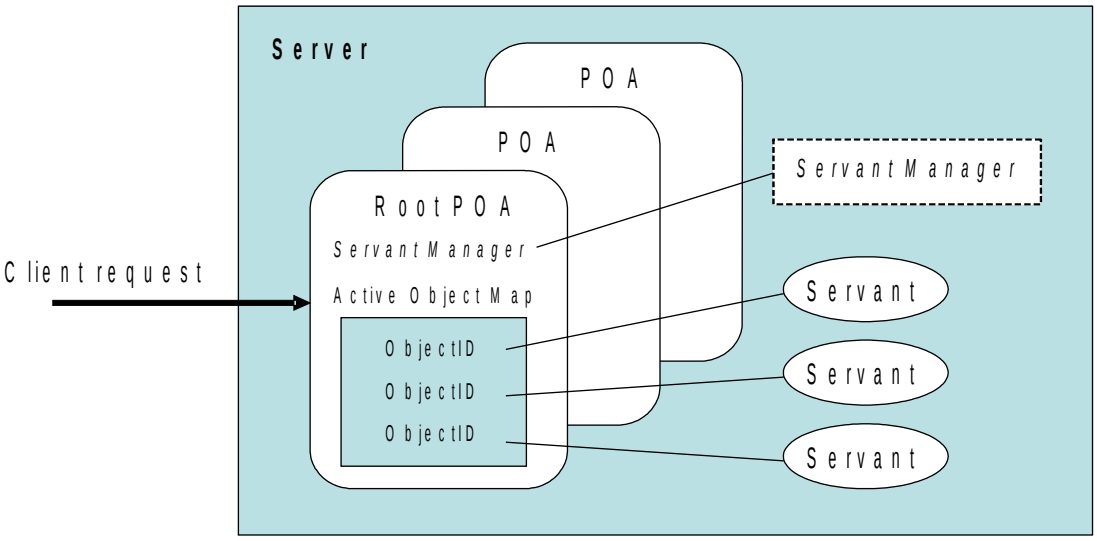


Figure 6 - Overview of the POA

The object that performs the client request is referred as a servant, which provides the implementation of the CORBA object requested by the client. A servant provides the implementation for one or more CORBA object references. To retreive a servant, each POA maintains an *Active Object Map* that maps all objects that have been activated in the POA to a servant. For each incoming request, the POA looks up the object reference in the Active Object Map and tries to find the responsible servant. If none is found, the request is either

delegated to a default servant, or a servant manager is invoked to activate or locate an appropriate servant. In addition to the name space for the objects, which are identified by Object Ids, a POA also provides a name space for POAs. A POA is created as a child of an existing POA, which forms a hierarchy starting with the root POA.

Each POA has a set of policies that define its characteristics. When creating a new POA, the default set of policies can be used or different values can be assigned that suit the application requirements. The POA specification defines

- *Thread policy* – Specifies the threading model to be used by the POA. Possible values are:
  - ORB_CTRL_MODEL – (default) The POA is responsible for assigning requests to threads.
  - SINGLE_THREAD_MODEL – the POA processes requests sequentially
- *Lifespan policy* - specifies the lifespan of the objects implemented in the POA. The lifespan policy can have the following values:
  - TRANSIENT (Default) Objects implemented in the POA cannot outlive the process in which they are first created. Once the POA is deactivated, an OBJECT_NOT_EXIST exception occurs when attempting to use any object references generated by the POA.
  - PERSISTENT Objects implemented in the POA can outlive the process in which they are first created.
- *Object ID Uniqueness policy* - allows a single servant to be shared by many abstract objects. The Object ID Uniqueness policy can have the following values:
  - UNIQUE_ID (Default) Activated servants support only one Object ID.
  - MULTIPLE_ID Activated servants can have one or more Object IDs. The Object ID must be determined within the method being invoked at run time.
- *ID Assignment policy* - specifies whether object IDs are generated by server applications or by the POA. The ID Assignment policy can have the following values:
  - USER_ID is for persistent objects, and
  - SYSTEM_ID is for transient objects

- *Servant Retention policy* - specifies whether the POA retains active servants in the Active Object Map. The Servant Retention policy can have the following values:
  - RETAIN (Default) The POA tracks object activations in the Active Object Map. RETAIN is usually used with ServantActivators or explicit activation methods on POA.
  - NON_RETAIN The POA does not retain active servants in the Active Object Map. NON_RETAIN is typically used with ServantLocators.

- *Request Processing policy* - specifies how requests are processed by the POA.
  - USE_ACTIVE_OBJECT_MAP (Default) If the Object ID is not listed in the Active Object Map, an OBJECT_NOT _EXIST exception is returned. The POA must also use the RETAIN policy with this value.
  - USE_DEFAULT_SERVANT If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the

request is dispatched to the default servant. If no default servant has been registered, an OBJ_ADAPTER exception is returned. The POA must also use the MULTIPLE_ID policy with this value.
    - USE_SERVANT_MANAGER If the Object ID is not listed in the Active Object Map or the NON_RETAIN policy is set, the servant manager is used to obtain a servant.
- *Implicit Activation policy* - specifies whether the POA supports implicit activation of servants. The Implicit Activation policy can have the following values:
    - IMPLICIT_ACTIVATION The POA supports implicit activation of servants. Servants can be activated by converting them to an object reference with org.omg.PortableServer.POA.servant_to_reference() or by invoking _this()on the servant. The POA must also use the SYSTEM_ID and RETAIN policies with this value.
    - NO_IMPLICIT_ACTIVATION (Default) The POA does not support implicit activation of servants.

It appears that to redirect replay_completion invocations to a default servant we need to create a POA with the Request Processing policy assigned with the value set to USE_DEFAULT_SERVANT. However to reach that default Servant we should first reach the POA that forward the request to the default servant. Indeed, the ORB uses a set of information to retrieve a POA; these information are contained in the object reference used by the client. Among these information there are the IP address and the port number where resides the server and also the POA name. To perform replay_completion invocations, the solution adopted by JBossTS is to provide one Servant, per machine, and located in the RecoveryManager process, a separate process from client or server applications. The next section explains how the indirection to a default Servant located on a separate process is provided for JacORB.

The default RecoveryCoordinator in JacOrb

JacORB does not define additional policies to redirect any request on a RecoveryCoordinator object to a default servant located in the Recovery Manager process. However it provides a set of APIs that allows building object references with specific IP address, port number and POA name in order to reach the appropriate default servant.

How Does it work

When the Recovery Manager is launched it seeks in the configuration the RecoveryActivator that need be loaded. Once done it invokes the startRCservice method of each loaded instances. As seen in in the previous chapter (Recovery Manager) the class to load that implements the RecoveryActivator interface is the class RecoveryEnablement. This generic class, located in the package com.arjuna.ats.internal.jts.orbspecific.recovery, hides the nature of the ORB being used by the application (JacORB). The following figure illustrates the behavior of the RecoveryActivator that leads to the creation of the default servant that performs replay_completion invocations requests.

In addition to the creation of the default servant, an object reference to a RecoveryCoordinator object is created and stored in the ObjectStore. As we will see this object reference will be used to obtain its IP address, port number and POA name and assign them to any RecoveryCoordinator object reference created on register_resource.

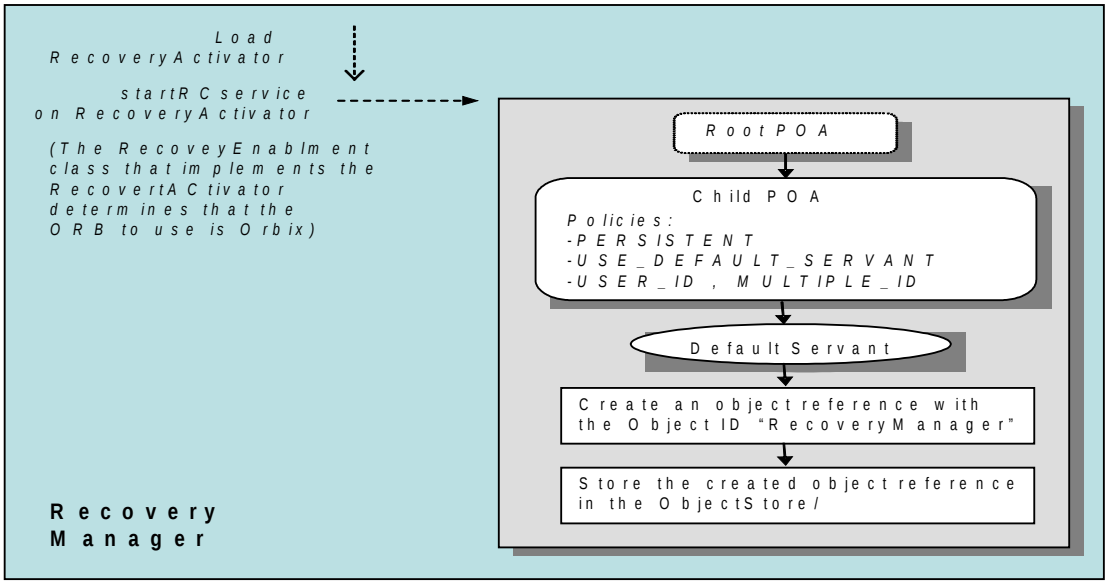


Figure 10 – Default RecoveryCoordinator created in the RecoveryManager

When an application registers a resource with a transaction, a RecoveryCoordinator object reference is expected to be returned. To build that object reference, the Transaction Service uses the RecoveryCoordinator object reference created within the Recovery Manager as a template. The new object reference contains practically the same information to retrieve the default servant (IP address, port number, POA name, etc.), but the Object ID is changed; now, it contains the Transaction ID of the transaction in progress and also the Process ID of the process that is creating the new RecoveryCoordinator object reference, as illustrated in Figure 11.

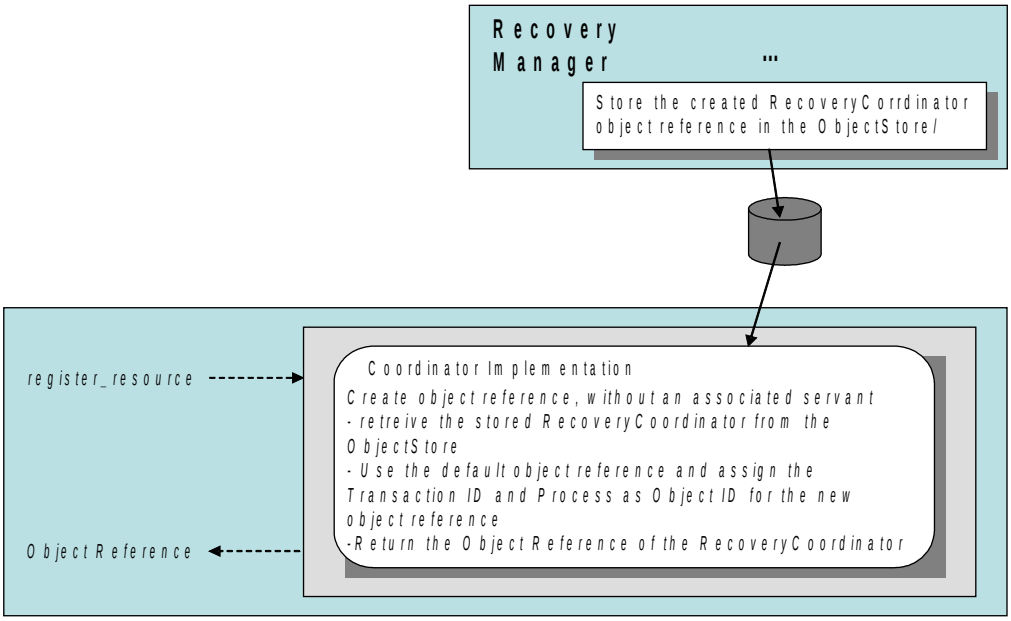


Figure 11 - Resource registration and returned RecoveryCoordinator Object reference build from a reference stored in the ObjectStore.

Since a RecoveryCoordintaor object reference returned to an application contains all information to retrieve the POA then the default servant located in the Recovery Manager, all replay_completion invocation, per machine, are forwarded to the same default RecoveryCoordinator that is able to retreive the Object ID from the incoming request to extract the transaction identifier and the process identifier needed to determine the status of the requested transaction.