JBoss Transactions 4.16

Failure Recovery Guide

Mark Little

JBoss Transactions 4.16 Failure Recovery Guide

Author Mark Little mlittle@redhat.com

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

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. Getting Help and Giving Feedback

2.1. Do You Need Help?

If you experience difficulty with a procedure described in this documentation, visit the Red Hat Customer Portal at *http://access.redhat.com*. Through the customer portal, you can:

- search or browse through a knowledgebase of technical support articles about Red Hat products.
- submit a support case to Red Hat Global Support Services (GSS).

- access other product documentation.

Red Hat also hosts a large number of electronic mailing lists for discussion of Red Hat software and technology. You can find a list of publicly available mailing lists at *https://www.redhat.com/mailman/listinfo*. Click on the name of any mailing list to subscribe to that list or to access the list archives.

2.2. Give us Feedback

If you find a typographical error, or know how this guide can be improved, we would love to hear from you. Submit a report in Bugzilla against the product **Fedora** and the component **jboss-jts**. The following link will take you to a pre-filled bug report for this product: https://bugzilla.redhat.com.

Fill out the following template in Bugzilla's **Description** field. Be as specific as possible when describing the issue; this will help ensure that we can fix it quickly.

```
Document URL:

Section Number and Name:

Describe the issue:

Suggestions for improvement:

Additional information:
```

Be sure to give us your name so that you can receive full credit for reporting the issue.

About This Guide

The Failure Recovery Guide contains information on how to use JBossTS.

1.1. Audience

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

1.2. Prerequisites

You should have installed JBossTS.

Architecture of the Recovery Manager

2.1. Crash Recovery Overview

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

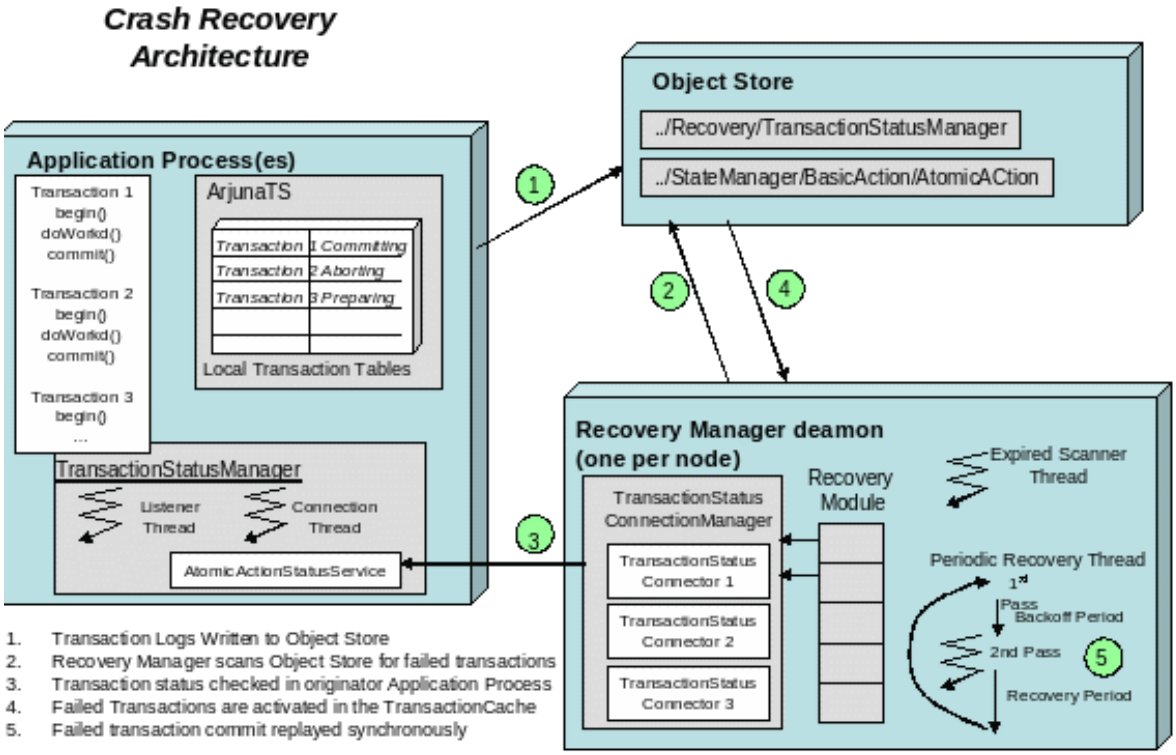

Figure 2.1. Recovery Manager Architecture

The Recovery Manager is a daemon process1 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.

2.2. 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.

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:

```
<entry key="RecoveryEnvironmentBean.recoveryActivators">
  list_of_class_names
<entry>
```

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

```
<entry key="RecoveryEnvironmentBean.recoveryActivators">
      com.arjuna.ats.internal.jts.orbspecific.recovery.RecoveryEnablement
<entry>
```

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:

```
<entry key="RecoveryEnvironmentBean.recoveryExtenstions">
   list_of_class_names
<entry>
```

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

```
<entry key="RecoveryEnvironmentBean.recoveryBackoffPeriod">
```

```
<entry key="RecoveryEnvironmentBean.periodicRecoveryPeriod">
```

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 (e.g., com.arjuna.ats.arjuna.recovery.recoveryExtension1= com.arjuna.ats.internal.ts.arjuna.recovery.AtomicActionRecoveryModule).

Note

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.

2.2.1. 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.

2.2.2. 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.

2.2.3. 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:

Example 2.1. RecoveryScan interface

```
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.

2.2.4. Recovering For Multiple Transaction Coordinators

Sometimes a single Recovery Manager can be made responsible for recovering transactions executing on behalf of multiple transaction coordinators. Conversely, due to specific configurations it may be that multiple Recovery Managers share the same Object Store and in which case should not conflict with each other, e.g., roll back transactions that they do not understand. Therefore, when running recovery it is necessary to tell JBossTS which types of transactions it can recover and which transaction identifiers it should ignore.

When necessary each transaction identifier that JBossTS creates may have a unique node identifier encoded within it and JBossTS will only recover transactions and states that match a specified node identifier. The node identifier for each JBossTS instance should be set via the com.arjuna.ats.arjuna.nodeIdentifier property. This value must be unique across JBossTS instances. The contents of this should be alphanumeric and not exceed 10 bytes in length. If you do not provide a value, then JBossTS will fabricate one and report the value via the logging infrastructure.

How this value is used will depend upon the type of resources being recovered and will be discussed within the relevant sections for the Recovery Modules.

2.3. 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.

Example 2.2. RecoveryModule interface

```
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();
}
```

2.3.1. 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.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

2.4. A Recovery Module for XA Resources

During recovery, the Transaction Manager needs to be able to communicate to all resource managers that are in use by the applications in the system. For each resource manager, the Transaction Manager uses the XAResource.recover method to retrieve the list of transactions that are currently in a prepared or heuristically completed state. Typically, the system administrator configures all transactional resource factories that are used by the applications deployed on the system. An example of such a resource factory is the JDBC XADataSource object, which is a factory for the JDBC XAConnection objects.

Because XAResource objects are not persistent across system failures, the Transaction Manager needs to have some way to acquire the XAResource objects that represent the resource managers which might have participated in the transactions prior to the system failure. For example, a Transaction Manager might, through the use of JNDI lookup mechanism, acquire a connection from each of the transactional resource factories, and then obtain the corresponding XAResource object for each connection. The Transaction Manager then invokes the XAResource.recover method to ask each resource manager to return the transactions that are currently in a prepared or heuristically completed state.

Note

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 the property JTAEnvironmentBean.xaRecoveryNodes; multiple values may be provided in a list. 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.nodeIdentifier.

One of the following recovery mechanisms will be used:

- If the XAResource is serializable, then the serialized form will be saved during transaction commitment, and used during recovery. It is assumed that the recreated XAResource is valid and can be used to drive recovery on the associated database.
- The com.arjuna.ats.jta.recovery.XAResourceRecovery, com.arjuna.ats.jta.recovery.XARecoveryResourceManager and com.arjuna.ats.jta.recovery.XARecoveryResource interfaces are used. These are described in detail later in this document.

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.

Note

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.

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.

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.

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”.

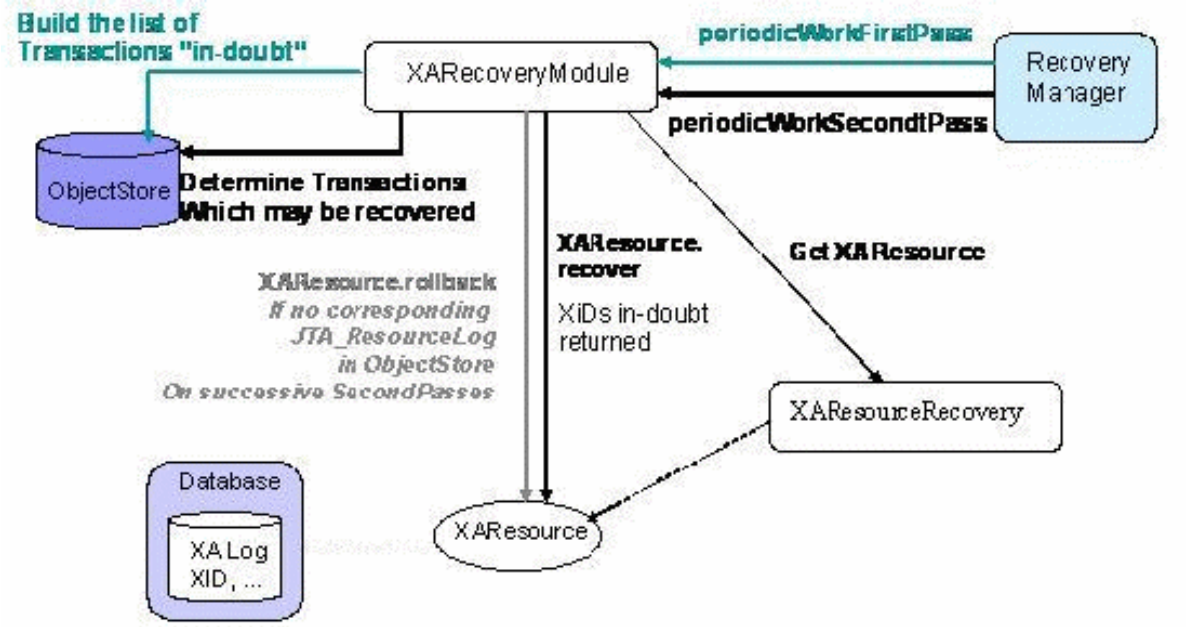

Figure 2.2. Resource-initiated recovery and XA Recovery

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. periodicWorkSecondPasses) 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.

2.4.1. 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.

2.5. Recovering XAConnections

When recovering from failures, JBossTS requires the ability to reconnect to databases that were in use prior to the failures in order to resolve any outstanding transactions. Most connection information will be saved by the transaction service during its normal execution, and can be used during recovery to recreate the connection. However, it is possible that not all such information will have been saved prior to a failure (for example, a failure occurs before such information can be saved, but after the database connection is used). In order to recreate those connections it is necessary to provide implementations of the following JBossTS interface com.arjuna.ats.jta.recovery.XAResourceRecovery, one for each database that may be used by an application.

Note

if using the transactional JDBC driver provided with JBossTS, then no additional work is necessary in order to ensure that recovery occurs.

To inform the recovery system about each of the XAResourceRecovery instances, it is necessary to specify their class names through the JTAEnvironmentBean.xaResourceRecoveryInstances property variable, whose values is a list of space separated strings, each being a classname followed by optional configuration information.

JTAEnvironmentBean.xaResourceRecoveryInstances=com.foo.barRecovery

Additional information that will be passed to the instance when it is created may be specified after a semicolon:

JTAEnvironmentBean.xaResourceRecoveryInstances=com.foo.barRecovery;myData=hello

Note

These properties need to go into the JTA section of the property file.

Any errors will be reported during recovery.

Example 2.3. XAResourceRecovery interface

```
public interface XAResourceRecovery {
 public XAResource getXAResource() throws SQLException;

 public boolean initialise(String p);

 public boolean hasMoreResources();
};
```

Each method should return the following information:

- initialise: once the instance has been created, any additional information which occurred on the property value (anything found after the first semi-colon) will be passed to the object. The object can then use this information in an implementation specific manner to initialise itself, for example.
- hasMoreResources: each XAResourceRecovery implementation may provide multiple XAResource instances. Before any call to getXAResource is made, hasMoreResources is called to determine whether there are any further connections to be obtained. If this returns false, getXAResource will not be called again during this recovery sweep and the instance will not be used further until the next recovery scan. It is up to the implementation to maintain the internal state backing this method and to reset the iteration as required. Failure to do so will mean that the second and subsequent recovery sweeps in the lifetime of the JVM do not attempt recovery.
- getXAResource: returns an instance of the XAResource object. How this is created (and how the parameters to its constructors are obtained) is up to the XAResourceRecovery implementation. The parameters to the constructors of this class should be similar to those used when creating the initial driver or data source, and should obviously be sufficient to create new XAResources that can be used to drive recovery.

Note

If you want your XAResourceRecovery instance to be called during each sweep of the recovery manager then you should ensure that once hasMoreResources returns false to indicate the end of work for the current scan it then returns true for the next recovery scan.

2.6. Alternative to XAResourceRecovery

The iterator based approach used by XAResourceRecovery leads to a requirement for implementations to manage state, which makes them more complex than necessary.

As an alternative, starting with JBossTS 4.4, users may provide an implementation of the public interface

Example 2.4. XAResourceRecoveryHelper

```
public interface com.arjuna.ats.jta.recovery.XAResourceRecoveryHelper {
    public boolean initialise(String p) throws Exception;
    public XAResource[] getXAResources() throws Exception;
}
```

During each recovery sweep the getXAResources method will be called and recovery attempted on each element of the array. For the majority of resource managers it will be necessary to have only one XAResource in the array, as the recover() call on it can return multiple Xids.

Unlike XAResourceRecovery instances, which are configured via the xml properties file and instantiated by JBossTS, instances of XAResourceRecoveryHelper and constructed by the application code and registered with JBossTS by calling

```
XARecoveryModule.addXAResourceRecoveryHelper(...)
```

The initialize method is not called by JBossTS in the current implementation, but is provided to allow for the addition of further configuration options in later releases.

XAResourceRecoveryHelper instances may be deregistered, after which they will no longer be called by the recovery manager. Deregistration may block for a time if a recovery scan is in progress.

```
XARecoveryModule.removeXAResourceRecoveryHelper(...)
```

The ability to dynamically add and remove instances of XAResourceRecoveryHelper whilst the system is running makes this approach an attractive option for environments in which e.g. datasources may be deployed or undeployed, such as application servers. Care should be taken with classloading behaviour in such cases.

2.7. Shipped XAResourceRecovery implementations

Recovery of XA datasources can sometimes be implementation dependant, requiring developers to provide their own XAResourceRecovery instances. However, JBossTS ships with several out-of-the-box implementations that may be useful.

Note

These XAResourceRecovery instances are primarily intended for when running JBossTS outside of a container such as JBossAS, since they rely upon XADataSources as the primary handle to drive recovery. If you are not running JBossTS stand-alone then you should consult the relevant integration documentation to ensure that the right recovery modules are being used.

- `com.arjuna.ats.internal.jdbc.recovery.BasicXARecovery`

  : this expects an XML property file to be specified upon creation and from which it will read the configuration properties for the datasource. For example:

  Example 2.5. XML datasource

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"><
<properties>
   <entry key="DB_X_DatabaseUser">username</entry>
   <entry key="DB_X_DatabasePassword">password"</entry>
   <entry key="DB_X_DatabaseDynamicClass">DynamicClass</entry>
   <entry key="DB_X_DatabaseURL">theURL</entry>
</properties>
```

- `com.arjuna.ats.internal.jdbc.recovery.JDBCXARecovery`

  : this recovery implementation should work on any datasource that is exposed via JNDI. It expects an XML property file to be specified upon creation and from which it will read the database JNDI name, username and password. For example:

  Example 2.6. JNDI datasource

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE properties SYSTEM "http://java.sun.com/dtd/properties.dtd"><
<properties>
        <entry key="DatabaseJNDIName">java:ExampleDS</entry>
        <entry key="UserName">username</entry>
        <entry key="Password">password</entry>
</properties>
```

Because these classes are XAResourceRecovery instances they are passed any necessary initialization information via the initialise operation. In the case of BasicXARecovery and JDBCXARecovery this should be the location of a property file and is specified in the JBossTS configuration file. For example:

```
com.arjuna.ats.jta.recovery.XAResourceRecoveryJDBC=com.arjuna.ats.internal.jdbc.recovery.JDBCXAResourceR
```

2.8. 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.

2.9. 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

```
<entry key="RecoveryEnvironmentBean.expiryScanners">
  list of class names
</entry>
```

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

```
<entry key="RecoveryEnvironmentBean.expiryScanInterval">
  number_of_hours
</entry>
```

All scanners inherit the same behaviour from the java interface ExpiryScanner. 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:

```
<entry key="RecoveryEnvironmentBean.transactionStatusManagerExpiryTime">
  number_of_hours
</entry> (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. This is also used when logs cannot be recovered automatically for other reasons, such as being corrupt or zero length. All logs are moved to a location based on the old location appended with /Expired.

Note

AtomicActionExpiryScanner is disabled by default. To enable it simply add it to the JBossTS properties file. You do not need to enable it in order to cope with (move) corrupt logs.

2.10. 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.

2.11. 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:

```
<entry key="RecoveryEnvironmentBean.transactionStatusManagerPort">
  port
</entry>
```

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

2.12. Object Store

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.

2.13. 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 CoordinatorEnvironmentBean.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 RecoveryEnvironmentBean.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.

How JBossTS manages the OTS Recovery Protocol

3.1. 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.

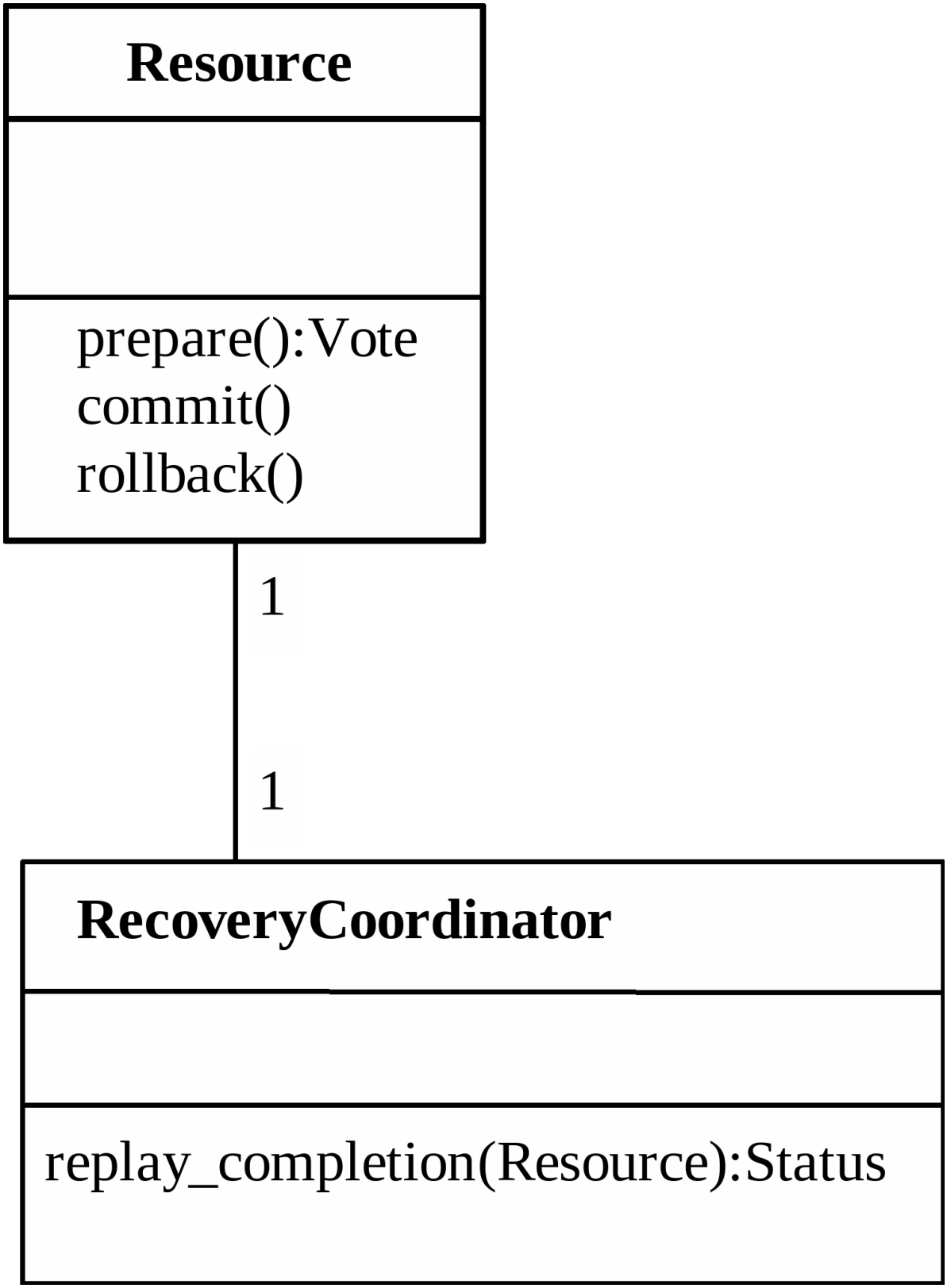

Figure 3.1. Resource and RecoveryCoordinator relationship

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.

3.2. 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.

3.2.1. 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.

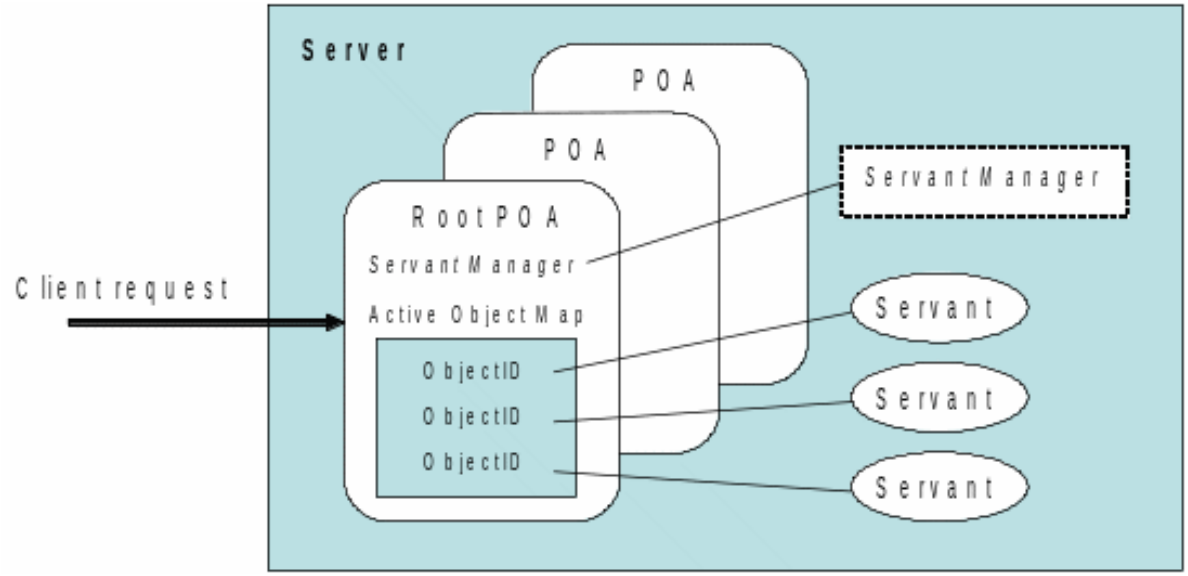

Figure 3.2. Overview of the POA

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.

3.3. 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.

3.3.1. 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.

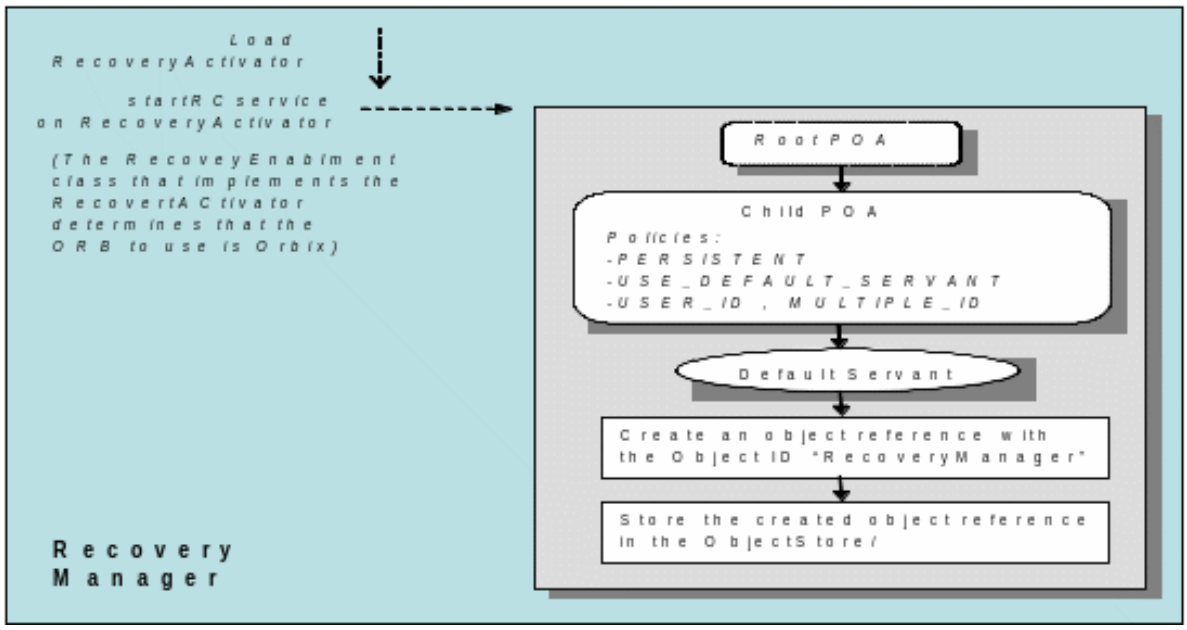

Figure 3.3. Recovery Manager

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.

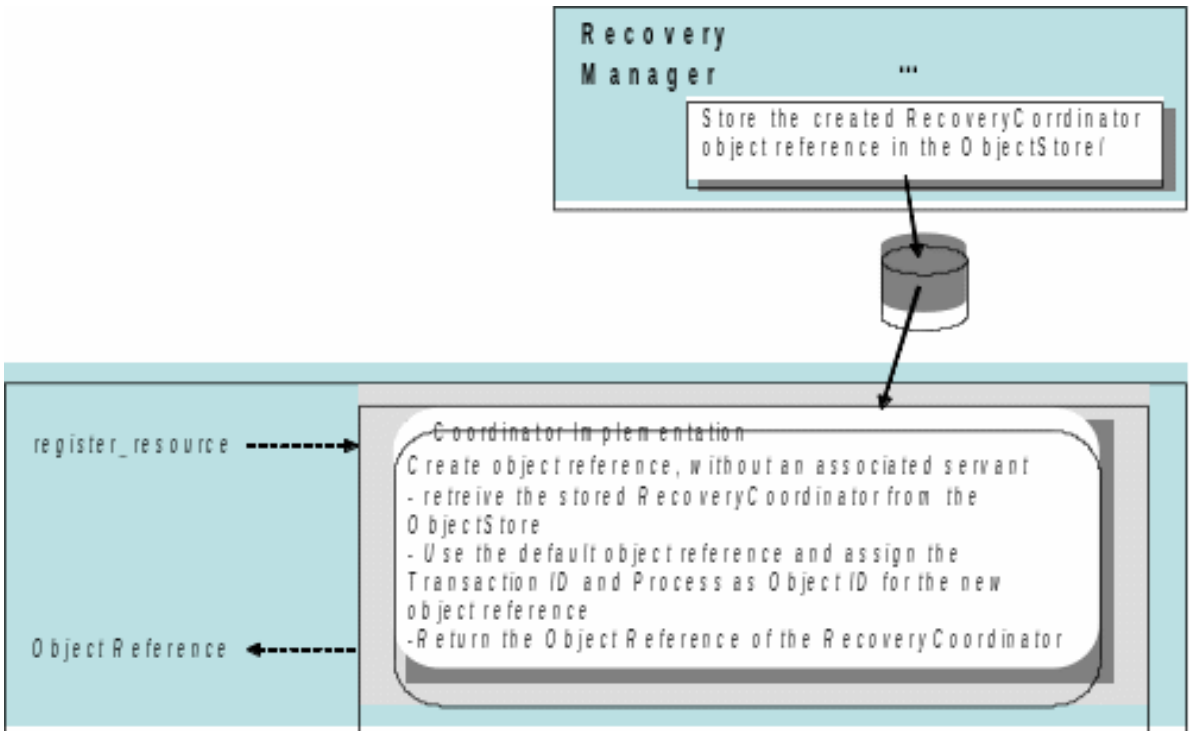

Figure 3.4. Resource registration and returned RecoveryCoordinator Object reference build from a reference stored in the ObjectStore.

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.

Configuration Options

4.1. Recovery Protocol in OTS - Overview

JBossTS is highly configurable. For full details of the configuration mechanism used, see the Programmer's Guide.

The following table shows the configuration features, with default values shown in italics. More details about each option can be found in the relevant sections of this document.

Note

You need to prefix each property in this table with the string com.arjuna.ats.arjuna.recovery. The prefix has been removed for formatting reasons, and has been replaced by ...

Configuration Name	Possible Values	Description
...periodicRecoveryPeriod	120/any positive integer	Interval between recovery attempts, in seconds.
...recoveryBackoffPeriod	10/any positive integer	Interval between first and second recovery passes, in seconds.
...expiryScanInterval	12/any integer	Interval between expiry scans, in hours. 0 disables scanning. Negative values postpone the first run.
...transactionStatusManagerExpiryTime	12/any positive integer	Interval after which a non-contactable process is considered dead. 0 = never.

Appendix A. Revision History

Revision 1 **Tue Apr 12 2010** **Tom Jenkinson**
tom.jenkinson@redhat.com

Initial creation of book by publican