Artix™ ESB

Transactions Guide, Java

Version 5.5, December 2008

Contents

List of Tables

LIST OF TABLES

List of Figures

LIST OF FIGURES

Preface

What is Covered in this Book

This book explains how to program and configure Artix transactions in Java, where the program is written using the JAX-RPC API and the Artix C++ runtime.

Who Should Read this Book

This guide is intended for Artix Java programmers. This guide assumes that the reader is familiar with WSDL and XML schemas.

The Artix Documentation Library

For information on the organization of the Artix library, the document conventions used, and where to find additional resources, see Using the Artix Library

Introduction to Transactions

This chapter provides an introduction to transaction concepts and to the transaction features supported by Artix.

In this chapter

This chapter discusses the following topics:

Basic Transaction Concepts

What is a transaction?

Artix gives separate software objects the power to interact freely even if they are on different platforms or written in different languages. Artix adds to this power by permitting those interactions to be transactions.

What is a transaction? Ordinary, non-transactional software processes can sometimes proceed and sometimes fail, and sometimes fail after only half completing their task. This can be a disaster for certain applications. The most common example is a bank fund transfer: imagine a failed software call that debited one account but failed to credit another. A transactional process, on the other hand, is secure and reliable as it is guaranteed to succeed or fail in a completely controlled way.

Example

The classical illustration of a transaction is that of funds transfer in a banking application. This involves two operations: a debit of one account and a credit of another (perhaps after extracting an appropriate fee). To combine these operations into a single unit of work, the following properties are required:

- If the debit operation fails, the credit operation should fail, and vice-versa; that is, they should both work or both fail.
- The system goes through an inconsistent state during the process (between the debit and the credit). This inconsistent state should be hidden from other parts of the application.
- It is implicit that committed results of the whole operation are permanently stored.

Properties of transactions

The following points illustrate the so-called ACID properties of a transaction.

Atomic — A transaction is an all or nothing procedure – individual updates are assembled and either committed or aborted (rolled back) simultaneously when the transaction completes.

Consistent — A transaction is a unit of work that takes a system from one consistent state to another.

Isolated — While a transaction is executing, its partial results are hidden from other entities accessing the transaction.

Durable — The results of a transaction are persistent.

Thus a transaction is an operation on a system that takes it from one persistent, consistent state to another.

Artix Transaction Features

Overview

This section gives a short overview of the main features supported by Artix transactions. The Artix transaction API is designed to be compatible with a variety of different underlying transaction systems. Generally, you can access the transaction system using a technology-neutral API, but the technology-specific APIs are also available, in case you need to access more advanced functionality.

The main features of Artix transactions are as follows:

- Supported protocols
- Client-side transaction support.
- Server-side transaction support.
- Compatibility with Orbix.
- Pluggable transaction system.
- One-phase commit.
- Two-phase commit.
- Transaction propagation.

Supported protocols

Artix supports distributed transactions using the following protocols:

- CORBA binding over IIOP.
- SOAP binding over any compatible transport.

Client-side transaction support

Transaction demarcation methods (`beginTransaction()`, `commitTransaction()` and `rollbackTransaction()`) can be used on the client side to initiate and terminate a transaction. While the transaction is active, all of the operations called from the current thread are included in the transaction (that is, the operations' request headers include a transaction context).

Server-side transaction support

On the server side, an API is provided that enables you to implement *transaction participants* (sometimes referred to as transactional resources). Using transaction participants, you can implement servers that participate in a distributed transaction with the ACID transaction properties (*Atomicity*, *Consistency*, *Integrity*, and *Durability*).

Artix supports several different approaches to implementing a transaction participant, depending on what kind of transaction system is loaded into your application. For example, you might take a technology-neutral approach by implementing the `TransactionParticipant` class, or you might decide to exploit the special features of a particular transaction system instead.

Compatibility with Orbix

The Artix transaction facility is fully compatible with CORBA OTS in Orbix. Hence, if you already have a transactional server implemented with Orbix ASP, you can easily integrate this with an Artix client, as shown in Figure 1.

Figure 1: *Artix Client Invokes a Transactional Operation on a CORBA OTS Server*
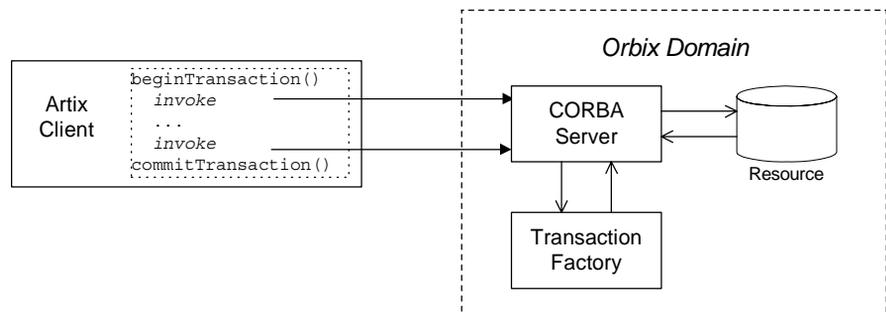
Pluggable transaction system

The underlying transaction system used by Artix can be replaced within a pluggable framework. Currently, the following transaction systems are supported by Artix:

- OTS Lite.
- OTS Encina.
- WS-AtomicTransactions.

One-phase commit

Artix supports the one-phase commit (1PC) protocol for transactions. This protocol can be used if there is only one resource participating in the transaction. The 1PC protocol essentially delegates the transaction completion to the single resource manager. Figure 2 shows a schematic overview of the 1PC protocol for a simple client-server system.

Figure 2: *One-Phase Commit Protocol*
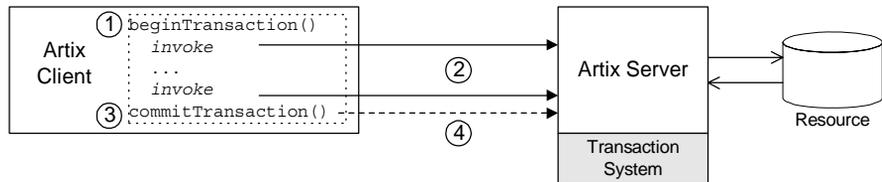
The 1PC protocol progresses through the following stages:

1. The client calls `beginTransaction()` to initiate the transaction.

2. Within the transaction, the client calls one or more WSDL operations on the remote server. The WSDL operations are transactional, requiring updates to a persistent resource.

3. The client calls `commitTransaction()` to make permanent any changes caused during the transaction (alternatively, the client could call `rollbackTransaction()` to abort the transaction).

4. The transaction system performs the commit phase by sending a notification to the server that it should perform a 1PC commit.

Two-phase commit

The two-phase commit (2PC) protocol enables multiple resources to participate in a transaction. In order to preserve the essential properties of a transaction involving multiple distributed resources, it is necessary to use a more elaborate algorithm. The 2PC algorithm consists of the following two phases:

*   *Prepare phase*—the transaction system notifies all of the participants to prepare the transaction. The participants prepare the transaction by saving the information that would be required to redo or undo the changes made during the transaction. At the end of this phase, the participants vote whether to commit or roll back the transaction.

• *Commit (or rollback) phase*—if all of the participants vote to commit the transaction, the transaction system notifies the participants to commit the changes. On the other hand, if one or more participants vote to roll back the transaction, the transaction system notifies the participants to roll back the changes.

Figure 3 shows a schematic overview of the 2PC protocol for a client and two remote servers.

Figure 3: *Two-Phase Commit Protocol*
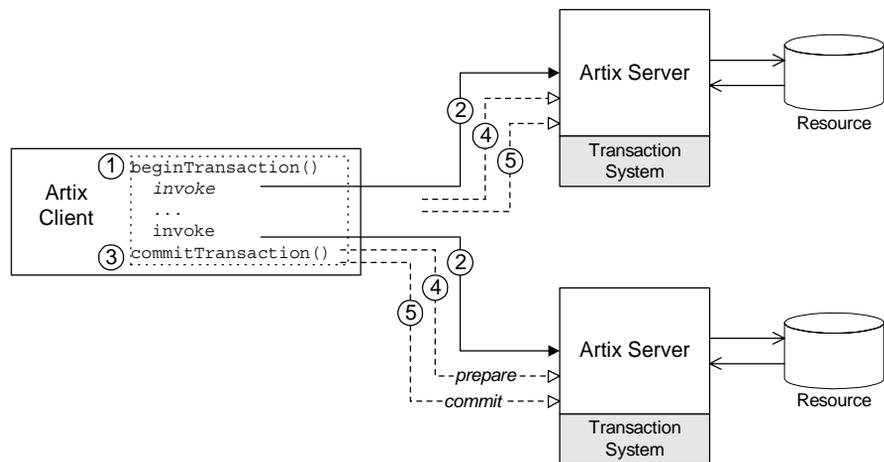
The 2PC protocol progresses through the following stages:

1. The client calls beginTransaction() to initiate the transaction.

2. Within the transaction, the client calls one or more WSDL operations on both of the remote servers.

3. The client calls commitTransaction() to make permanent any changes caused during the transaction (alternatively, the client could call rollbackTransaction() to abort the transaction).

4. The transaction system performs the prepare phase by polling all of the remote transaction participants (the first phase of a two-phase commit).

5. The transaction system performs the commit or rollback phase by sending a notification to all of the remote transaction participants (the second phase of a two-phase commit).

Transaction propagation

If you have a section of code executing within a transaction context, Artix automatically propagates a transaction context with the request message, whenever a remote operation is called.

For example, consider a three-tier system, where a client initiates a transaction, invokes an operation on server 1, and then server 1 makes a further call on server 2. In this scenario, Artix automatically propagates the transaction to server 2. The transaction is propagated, even if the protocol between the client and server 1 differs from the protocol used between server 1 and server 2.

Selecting a Transaction System

Using the Artix plug-in architecture, you can choose between a number of different transaction system implementations. Because the Artix transaction API is designed to be independent of the underlying transaction system, it is possible to select a particular transaction system at runtime. Typically, you would choose the transaction system that provides the best match for your services. For example, if the majority of your services are SOAP-based, you would select the WS-AT transaction system.

In this chapter

This chapter discusses the following topics:

Configuring OTS Lite

Overview

The *OTS Lite plug-in* is a lightweight transaction manager, which is subject to the following restrictions: it supports the 1PC protocol only and it lets you register only one resource. This plug-in allows applications that only access a single transactional resource to use the OTS APIs without incurring a large overhead, but allows them to migrate easily to the more powerful 2PC protocol by switching to a different transaction manager. Figure 4 shows a client-server deployment that uses the OTS Lite plug-in.

Figure 4: *Overview of a Client-Server System that Uses OTS Lite*
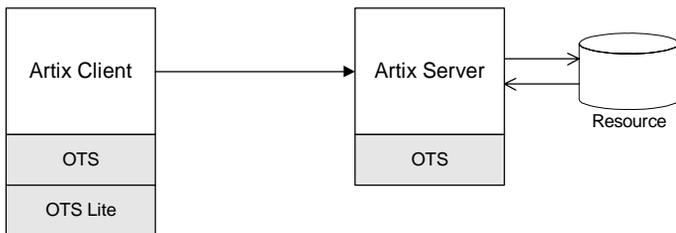
OTS Lite and interposition

If you plan to use OTS Lite in an application that needs to propagate transactions between different transaction systems, you should be aware that OTS Lite is subject to certain limitations in the context of interposition. See "Limitation of using OTS Lite with propagation" on page 46 for details.

Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

`plugins:bus:default_tx_provider:plugin`

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

In order to use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
# Artix Configuration File
ots_lite_client_or_server {
    plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
    orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Lite plug-in

The OTS Lite plug-in, which is capable of managing 1PC transactions, can be loaded on the client side, but it is not usually needed on the server side. You can load the OTS Lite plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_lite` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

```
# Artix Configuration File
ots_lite_client_or_server {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots"];
  initial_references:TransactionFactory:plugin = "ots_lite";
  ...
};
```

  This style of configuration has the advantage that the OTS Lite plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_lite` plug-in by adding it to the list of `orb_plugins`, as follows:

```
# Artix Configuration File
ots_lite_client {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots", "ots_lite"];
  ...
};
```

Sample configuration

The following example shows a sample configuration for using the OTS Lite transaction manager:

```
# Artix Configuration File

# Basic configuration for transaction plug-ins (shared library
#   names and so on) included in the global configuration scope.
#    ... (not shown)

ots_lite_client_or_server {
    plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
    orb_plugins = ["xmlfile_log_stream", "iiop_profile", "giop",
    "iiop", "ots"];
    initial_references:TransactionFactory:plugin = "ots_lite";
};
```

Configuring OTS Encina

Overview

The Encina OTS Transaction Manager provides full recoverable 2PC transaction coordination implemented on top of the industry proven Encina Toolkit from IBM/Transarc. Encina supports both 1PC and 2PC protocols and allows you to register multiple resources. Figure 5 shows a client/server deployment that uses the OTS Encina plug-in.

Figure 5: *Overview of a Client-Server System that Uses OTS Encina*
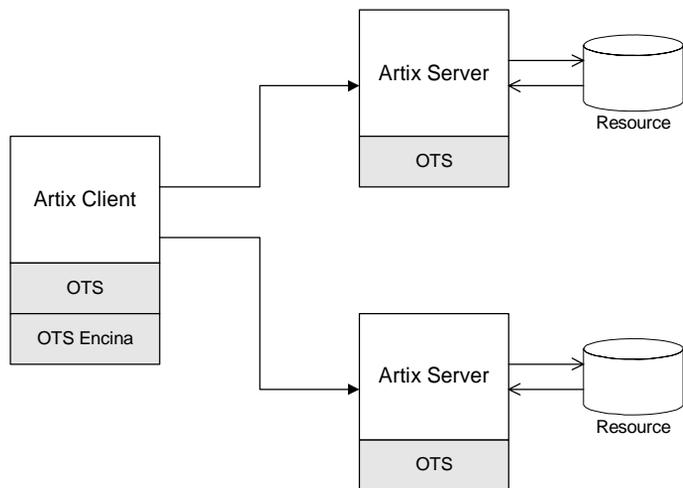
Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the CORBA OTS transaction system, you must initialize this configuration variable with the value, `ots_tx_provider`.

Loading the OTS plug-in

For applications that use the CORBA OTS transaction system, the OTS plug-in must be loaded both by the client and by the server. To load the OTS plug-in, include the `ots` plug-in name in the `orb_plugins` list. For example:

```
# Artix Configuration File
ots_encina_client_or_server {
    plugins:bus:default_tx_provider:plugin = "ots_tx_provider";
    orb_plugins = [ ..., "ots"];
};
```

Loading the OTS Encina plug-in

The OTS Encina plug-in, which is capable of managing 1PC and 2PC transactions, can be loaded on the client side, but it is not usually needed on the server side. You can load the OTS Encina plug-in in one of the following ways:

- *Dynamic loading*—configure Artix to load the `ots_encina` plug-in dynamically, if it is required. For this approach, you need to configure the `initial_references:TransactionFactory:plugin` variable as follows:

```
# Artix Configuration File
ots_encina_client_or_server {
  plugins:bus:default_tx_provider:plugin="ots_tx_provider";
  orb_plugins = [ ..., "ots"];
  initial_references:TransactionFactory:plugin="ots_encina";
  ...
};
```

This style of configuration has the advantage that the OTS Encina plug-in is loaded only if it is actually needed.

- *Explicit loading*—load the `ots_encina` plug-in by adding it to the list of `orb_plugins`, as follows:

```
# Artix Configuration File
ots_lite_client {
  plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
  orb_plugins = [ ..., "ots", "ots_encina"];
  ...
};
```

Sample configuration

Example 1 shows a complete configuration for using the OTS Encina transaction manager:

Example 1: *Sample Configuration for OTS Encina Plug-In*

```
# Artix Configuration File
ots_encina_client_or_server {
1     plugins:bus:default_tx_provider:plugin= "ots_tx_provider";
      orb_plugins = [ ..., "ots"];

2     initial_references:TransactionFactory:plugin = "ots_encina";

3     plugins:ots_encina:direct_persistence = "true";
      plugins:ots_encina:iiop:port = "3213";

4     plugins:ots_encina:initial_disk = "../../log/encina.log";
5     plugins:ots_encina:initial_disk_size = "1";
6     plugins:ots_encina:restart_file =
      "../../log/encina_restart";
7     plugins:ots_encina:backup_restart_file =
      "../../log/encina_restart.bak";

      # Boilerplate configuration settings for OTS Encina:
      # (you should never need to change these)
8     plugins:ots_encina:shlib_name = "it_ots_encina";
      plugins:ots_encina_adm:shlib_name = "it_ots_encina_adm";
      plugins:ots_encina_adm:grammar_db =
      "ots_encina_adm_grammar.txt";
      plugins:ots_encina_adm:help_db = "ots_encina_adm_help.txt";
};
```

The preceding configuration can be described as follows:

1. These two lines configure Artix to use the CORBA OTS transaction system and load the OTS plug-in.

2. This line configures Artix to load the `ots_encina` plug-in dynamically, if it is needed by the application (typically needed on the client side).

3. Configuring Encina to use direct persistence means that the Encina transaction manager service listens on a fixed IP port. The port on which the transaction manager listens is specified by the `plugins:ots_encina:iiop:port` variable.

4. The `plugins:ots_encina:initial_disk` variable specifies the path for the initial file used by the Encina OTS for its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).

5. The `plugins:ots_encina:initial_disk_size` variable specifies the size of the initial file used by the Encina OTS for its transaction logs. Defaults to 2.

6. The `plugins:ots_encina:restart_file` variable specifies the path for the restart file, which Encina OTS uses to locate its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).

7. The `plugins:ots_encina:backup_restart_file` variable specifies the path for the backup restart file, which Encina OTS uses to locate its transaction logs.

   If this file does not exist when you start the client, Encina OTS automatically creates it (cold start).

8. The settings in the next few lines specify the basic configuration of the OTS Encina plug-in. It should not be necessary ever to change the values of these configuration settings.

Configuring Non-Recoverable WS-AT

Overview

The WS-AtomicTransactions (WS-AT) transaction system uses SOAP headers to transmit transaction contexts between the participants in a transaction. The lightweight WS-AT transaction system supports the 2PC protocol and allows you to register multiple resources; unlike OTS Encina, however, it does not support recovery. Figure 6 shows a client/server deployment that uses the lightweight WS-AT transaction system.

Figure 6: *Client-Server System that Uses Non-Recoverable WS-AT*
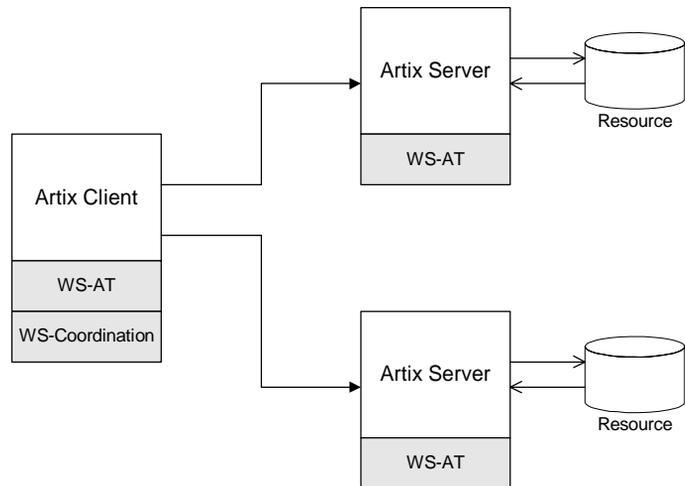
Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the WS-AT transaction system, you must initialize this configuration variable with the value, wsat_tx_provider.

Disabling recovery

Since Artix version 4.0, the WS-AT transaction system is recoverable by default (by layering itself over OTS Encina). Hence, to use the lightweight, non-recoverable version of WS-AT in your application, you need to explicitly disable recovery by setting the following configuration variable to true:

```
plugins:ws_coordination_service:disable_tx_recovery = "true";
```

Plug-ins for WS-AT

The division of the WS-AT transaction system into separate plug-ins reflects the fact that the WS-AT specification has two distinct parts: WS-AtomicTransactions and WS-Coordination.

The following plug-ins are required to support the WS-AT transaction system:

- `wsat_protocol` plug-in—implements WS-AtomicTransactions. It is required by all services and clients that use WS-AT transactions. This plug-in enables an Artix executable to receive and transmit WS-AT transaction contexts.

- `ws_coordination_service` plug-in—implements WS-Coordination. Only one instance of this plug-in is required (typically, loaded into a client). This plug-in coordinates the two-phase commit protocol.

Sample configuration

Example 2 shows a complete configuration for using the non-recoverable WS-AT transaction manager:

Example 2: *Sample Configuration for Non-Recoverable WS-AT*

```
# Artix Configuration File
ws_atomic_transactions {
    client
    {
1      orb_plugins = ["local_log_stream",
    "ws_coordination_service"];
2      plugins:bus:default_tx_provider:plugin ="wsat_tx_provider";
3      plugins:ws_coordination_service:disable_tx_recovery ="true";
    };

    server
    {
4      orb_plugins = ["local_log_stream", "wsat_protocol",
    "coordinator_stub_wsdl"];
    plugins:ws_coordination_service:disable_tx_recovery ="true";
```

Example 2: *Sample Configuration for Non-Recoverable WS-AT*

```
5           // No need to specify default_tx_provider here.
        };
};
```

The preceding configuration can be described as follows:

1. The `ws_coordination_service` plug-in is needed only on the client
    side. Artix does *not* support auto-loading of this plug-in; you must
    explicitly include it in the `orb_plugins` list.

    The `ws_coordination_service` plug-in implicitly loads the
    `wsat_protocol` plug-in as well. Hence, it is unnecessary to include
    `wsat_protocol` plug-in in the `orb_plugins` list on the client side.

2. This line specifies that WS-AT is the default transaction provider. This
    implies that whenever a client initiates a transaction (for example, by
    calling `begin_transaction()`), Artix creates a new WS-AT transaction
    by default.

3. This line specifies that transaction recovery is disabled. The effect of
    this setting is that the transaction system relies on a lightweight,
    non-recoverable implementation of WS-AT.

4. The server needs to load the `wsat_protocol` plug-in, in order to
    process incoming atomic transactions coordination contexts and to
    propagate transaction contexts. The `coordinator_stub_wsdl` plug-in
    enables the server to talk to the WS-Coordination service on the client
    side.

5. Strictly speaking, it is unnecessary to specify a default transaction
    provider on the server side. On the server side, the transaction provider
    is automatically determined by the incoming transaction context.

    If the server needs to initiate its own transactions, however, it would be
    appropriate to set the default transaction provider here also.

References

The specifications for WS-AtomicTransactions and WS-Coordination are available at the following locations:

- WS-AtomicTransactions (http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-AtomicTransaction.pdf).

- WS-Coordination (http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf).

Configuring Recoverable WS-AT

Overview

In order to provide enterprise-level transaction management using the WS-AT protocols, Artix supports an option to layer WS-AT over the OTS Encina transaction manager. With this configuration, WS-AT becomes a fully recoverable transaction system. Figure 7 shows a client/server deployment that uses the recoverable WS-AT transaction system.

Figure 7: *Client-Server System that Uses Recoverable WS-AT*

Default transaction provider

The following variable specifies the default transaction system used by an Artix client or server:

```
plugins:bus:default_tx_provider:plugin
```

To select the WS-AT transaction system, you must initialize this configuration variable with the value, `wsat_tx_provider`.

Example 3:   *Sample Configuration for Recoverable WS-AT*

```
      plugins:ots_encina_adm:shlib_name = "it_ots_encina_adm";
      plugins:ots_encina_adm:grammar_db =
    "ots_encina_adm_grammar.txt";
      plugins:ots_encina_adm:help_db = "ots_encina_adm_help.txt";
    };

    server
    {
4       orb_plugins = ["local_log_stream", "wsat_protocol",
    "coordinator_stub_wsdl"];
5       // No need to specify default_tx_provider here.
    };
};
```

The preceding configuration can be described as follows:

1. The `ws_coordination_service` plug-in is needed only on the client
    side. Artix does *not* support auto-loading of this plug-in; you must
    explicitly include it in the `orb_plugins` list.

    The `ws_coordination_service` plug-in implicitly loads the
    `wsat_protocol`, `ots`, and `ots_encina` plug-ins as well. Hence, it is
    unnecessary to include the `wsat_protocol`, `ots`, and `ots_encina`
    plug-ins in the `orb_plugins` list on the client side.

2. This line specifies that WS-AT is the default transaction provider. This
    implies that whenever a client initiates a transaction (for example, by
    calling `begin_transaction()`), Artix creates a new WS-AT transaction
    by default.

3. From this line up to the end of the client scope shows the OTS Encina
    configuraion settings. For detailed descriptions of the OTS Encina
    settings, see "Sample configuration" on page 25.

4. The server needs to load the `wsat_protocol` plug-in, in order to
    process incoming WS-AT coordination contexts and to propagate
    transaction contexts. The `coordinator_stub_wsdl` plug-in enables the
    server to talk to the WS-Coordination service on the client side.

5.   Strictly speaking, it is unnecessary to specify a default transaction provider on the server side. On the server side, the transaction provider is automatically determined by the incoming transaction context.

     If the server needs to initiate its own transactions, however, it would be appropriate to set the default transaction provider here also.

Basic Transaction Programming

This chapter covers the basics of programming transactional clients and servers. For simple applications, this probably covers all you need to know about transaction programming.

In this chapter

This chapter discusses the following topics:

Artix Transaction Interfaces

Overview

Figure 8 shows an overview of the main classes that make up the Artix transaction API. The Artix transaction API is designed to function as a generic wrapper for a wide variety of specific transaction systems. As long as you use the Artix APIs, you will be able to switch between any of the transaction systems supported by Artix.

Figure 8: *Overview of the Artix Transaction API*

```
┌────────────────────┐  getTransactionSystem()  ┌──────────────────────────────────────────┐
│ com.iona.jbus.Bus  │─────────────────────────▶│ com.iona.jbus.transactions.TransactionSystem │
└────────────────────┘                          └──────────────────────────────────────────┘
                                                        │ getTransactionManager()
                                                        ▼
                                              ┌──────────────────────────────────────────────┐
                                              │ com.iona.jbus.transactions.TransactionManager  │
                                              └──────────────────────────────────────────────┘

                                              ┌──────────────────────────────────────────────────┐
                                              │ com.iona.jbus.transactions.TransactionParticipant  │
                                              └──────────────────────────────────────────────────┘

                                              ┌────────────────────────────────────────────────────────┐
                                              │ com.iona.jbus.transactions.TransactionNotificationHandler │
                                              └────────────────────────────────────────────────────────┘
```
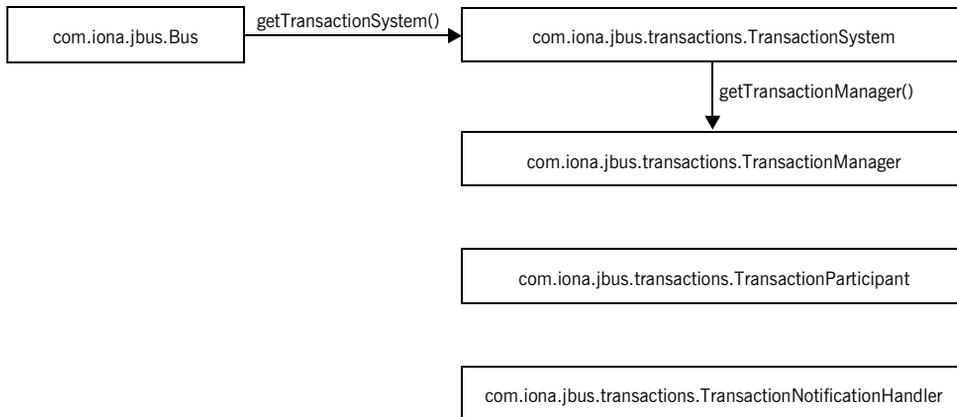
Accessing the transaction system

To access the Artix transaction system, call the `getTransactionSystem()` method on the bus. The returned `com.iona.jbus.transaction.TransactionSystem` object provides the starting point for accessing all aspects of Artix transactions.

The signature of `Bus.getTransactionSystem()` is shown in Example 4.

Example 4: *Signature for getTransactionSystem()*

```
TransactionSystem getTransactionSystem() throws BusException;
```

TransactionSystem class

The `TransactionSystem` class provides the basic methods needed for transaction demarcation (`beginTransaction()`, `commitTransaction()` and `rollbackTransaction()`). For more details see "Beginning and Ending Transactions" on page 38.

In addition to providing access the transaction demarcation method the `TransactionSystem` object provides two other methods:

- `getTransactionManager()` returns a `com.iona.jbus.transaction.TransactionManager` object that provides access to some of the more advanced transaction features.
- `withinTransaction()` returns true if it is called within an active transaction.

TransactionManager class

The `TransactionManager` class provides advanced transaction functionality. The most important method it provides is `enlist()`, which enables you to implement a transactional resource by enlisting a transaction participant object. It also provides methods for attaching and detaching threads from a transaction. See "Threading" on page 49.

TransactionParticipant interface

The `com.iona.jbus.transaction.TransactionParticipant` interface is used to create transactional resources. An implementation of `TransactionParticipant` acts as the resource manager for the datastore involved in the transaction. It receives callbacks from the transaction manager that are used to coordinate the commit or rollback steps with other transaction participants. For more details, see "Recoverable Resources" on page 69.

TransactionNotificationHandler interface

The `com.iona.jbus.transacation.TransactionNotificationHandler` interface is used to create objects that receive notification callbacks from the transaction manager whenever a transaction is either committed or rolled back.

Beginning and Ending Transactions

Overview

On the client side, the functions for beginning and committing (or rolling back) a transaction are collectively referred to as *transaction demarcation* methods. Within a given thread, any Artix operations invoked after the transaction *begin* and before the transaction *commit* (or *rollback*) are implicitly associated with the transaction. The transaction demarcation methods are typically the only methods that you need on the client side.

TransactionSystem methods

Example 5 shows the methods belonging to the TransactionSystem interface.

Example 5:   *The TransactionSystem Interface*

```java
// Java
package com.iona.jbus.transaction

public interface TransactionSystem {
    void beginTransaction()
    throws TransactionAlreadyActiveException,
            TransactionSystemUnavailableException,
            BusException;

    boolean commitTransaction(boolean reportHeuristics)
    throws NoActiveTransactionException, BusException;

    void rollbackTransaction()
    throws NoActiveTransactionException, BusException;

    TransactionManager getTransactionManager(
                              String transactionManagerType
                         )
    throws TransactionSystemUnavailableException, BusException;

    boolean withinTransaction();
};
```

Client transaction functions

The following functions are used to demarcate transactions on the client side:

- beginTransaction()—creates a new transaction on the client side and associates it with the current thread. This method takes no arguments and has no return value.

  This method can throw the following exceptions:

  - TransactionAlreadyActiveException is thrown if beginTransaction() is called inside an already active transaction.

  - TransactionSystemUnavailableException is thrown if the transaction system cannot be loaded. This usually points to a configuration problem.

- `commitTransaction()`—ends the transaction normally, making any changes permanent. This method takes a single boolean argument, `reportHeuristics`, and returns `true`, if the transaction is commited successfully.

  This method can throw the following exception:

  ◆ `NoActiveTransactionException` is thrown if there is there is no transaction associated with the current thread.

- `rollbackTransaction()`—aborts the transaction, rolling back any changes.

  This method can throw the following exception:

  ◆ `NoActiveTransactionException` is thrown if there is there is no transaction associated with the current thread.

Other transaction functions

In addition to the preceding demarcation functions, which are intended for use on the client side, the `TransactionSystem` class also provides the following functions, which can be used both on the client side and on the server side:

- `withinTransaction()`—returns `true` if the current thread is associated with a transaction; otherwise, `false`.

- `getTransactionManager()`—returns a reference to a `TransactionManager` object, which provides access to advanced transaction features.

  Typically, a `TransactionManager` object is needed on the server side in order to enlist participants in a transaction (for example, see "Recoverable Resources" on page 69).

  This method can throw the following exception:

  ◆ `TransactionSystemUnavailableException` is thrown if the transaction system cannot be loaded.

Example

Example 6 shows an Artix client that invokes a series of operations as an atomic transaction. The client invokes on single service called `Data`. `Data` provides a `read` and a `write` function.

Example 6: *Transactional Client Example*

```
import java.util.*;
import java.io.*;
import java.net.*;
import java.rmi.*;

import javax.xml.namespace.QName;
import javax.xml.rpc.*;

import com.iona.jbus.Bus;
import com.iona.jbus.transaction.*;

public class Transaction Client
{
  public static void main(String args[]) throws Exception
  {
    Bus bus = Bus.init(args);
    String serviceName = "DataService";
    String wsdlName = "soap_tx_demo.wsdl";
    QName serviceQName = new QName("http://transaction_demo",
                                   serviceName);
    QName portQName = new QName("", "DataSOAPPort");
    Data client = null;
    URL wsdlLocation = new URL(wsdlName);
    ServiceFactory factory = ServiceFactory.newInstance();
    Service service = factory.createService(wsdlLocation,
                                             serviceQName);
    client = (Data)service.getPort(portQName,Data.class);

    TransactionSystem txSystem = bus.getTransactionSystem();

    txSystem.beginTransaction();
```

Line markers: **1** `Bus bus = Bus.init(args);` **2** `String serviceName...` **3** `TransactionSystem txSystem...` **4** `txSystem.beginTransaction();`

Example 6:   *Transactional Client Example*

```
5    try
     {
       int value = client.read();
       System.out.println("value: " + value);
       System.out.println("Incrementing the value" );
       client.write(value + 1);
       System.out.println("New values are" );
       int value2 = client.read();
       System.out.println("value: " + value2);
     }
6    catch (Trowable T)
     {
       System.out.println("rolling back transaction...");
       txSystem.rollbackTransation();
       System.exit(1);
     }
7    System.out.println("committing transaction...");
     boolean result = txSystem.commitTransaction(true);
     if (result)
     {
       System.out.println("Transaction committed!");
     }
     else
     {
       System.out.println("Transaction *not* Committed!!");
     }
   }
 }
```

The code in Example 6 does the following:

1. Initializes the bus.

2. Creates a proxy for the Data service.

3. Gets the transaction system.

4. Begins a transaction.

5. Invokes operations on the service.

6. Rolls back the transaction if an exception is thrown while invoking operations on the service.

7. Commits the transaction if all of the operations succeeded.

Transaction Propagation

Transaction propagation refers to the implicit propagation of transaction context data in message headers.

In this chapter

This chapter discusses the following topics:

Transaction Propagation and Interposition

Overview

In a multi-tier application, Artix automatically propagates transactions from tier to tier. This ensures that all of the processes that are relevant to the outcome of a transaction can participate in the transaction. You do not have to do anything special to switch on transaction propagation; it is enabled by default. However, the receiver of a transaction context must have a transaction plug-in loaded, otherwise the transaction context would be ignored.

Transaction contexts

A transaction context is a data structure that is transmitted to a remote server and used to recreate the transaction at a remote location. The type of transaction context that is transmitted depends on the middleware protocol. Artix supports the following kinds of transaction context:

- *OTS transaction context*—a transaction context that is sent in a GIOP header (part of the CORBA standard).
- *WS-AT transaction context*—a transaction context that is embedded in a SOAP header.

Propagation scenario

The propagation scenario shown in Figure 9 shows two different kinds of transaction propagation, as follows:

- *Transaction propagation within a single middleware technology*—the OTS transaction context, which propagates across the top half of Figure 9, illustrates a simple kind of propagation, where the client and the servers all use the same CORBA OTS transaction technology.
- *Transaction propagation across middleware technologies*—the WS-AT transaction context, which propagates across the bottom half of Figure 9, illustrates a kind of propagation, where the transaction crosses technology domains. While the client uses OTS Encina to

manage the transaction, it must generate a WS-AT transaction context to send to the server. The ability to transform transaction contexts is known as *interposition*.

Figure 9: *Overview of Different Kinds of Transaction Propagation*
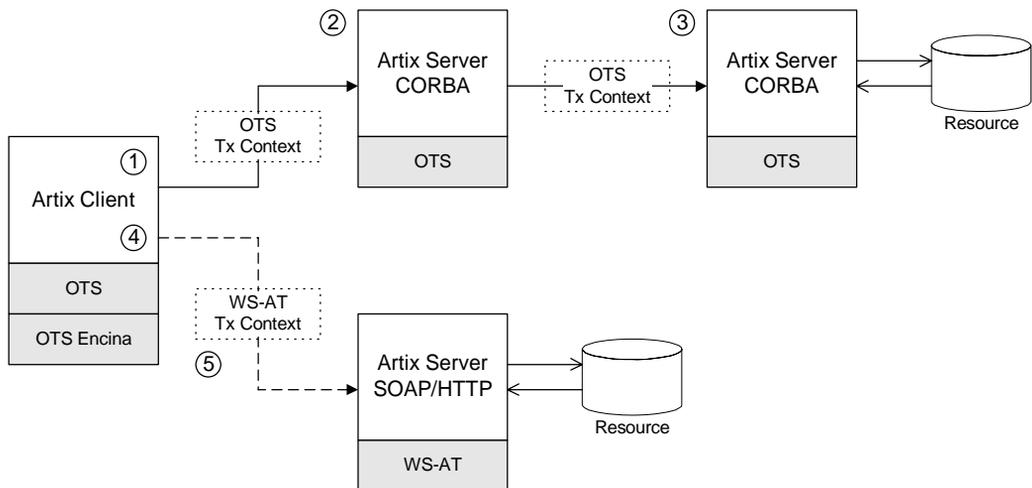
Scenario steps

The propagation scenario shown in Figure 9 can be described as follows:

Stage	Description
1	The Artix client (which is configured to use the OTS Encina transaction system) initiates a transaction by calling the `beginTransaction()` method. The client then invokes a remote operation, which results in a request message being sent over an IIOP connection.
2	The request received by the server includes an OTS transaction context embedded in a GIOP header. Although this server does not participate directly in the transaction (it registers no resources), it is capable of propagating the transaction context to the next tier in the application.

Stage	Description
3	The third tier of the application receives a request containing an OTS transaction context. This server participates in the transaction by registering a database resource with the OTS transaction manager.
4	The client invokes a remote operation, which results in a request message being sent over a SOAP/HTTP connection.
5	In this case, Artix automatically translates the OTS transaction into a WS-AT transaction context, which is suitable for transmission in the header of the SOAP/HTTP request. There is no need to perform any special configuration or programming to enable interposition; it occurs automatically.

Limitation of using OTS Lite with propagation

Figure 10 shows an interposition scenario where the client, which uses an OTS transaction system, connects to a SOAP/HTTP server, which uses the WS-AT transaction system.

Figure 10: *Limitation of Transaction Propagation Using OTS Lite*
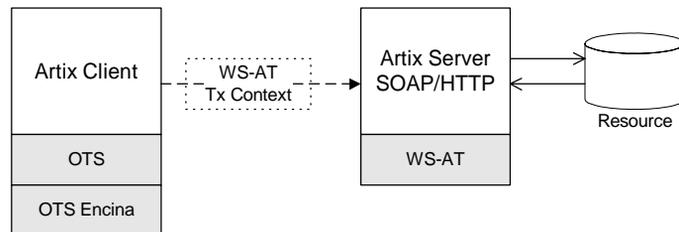
Because there is only one explicitly registered resource in this scenario (the database connected to the server), it would seem that the client could use an OTS Lite transaction manager for this scenario. In reality, however, the client *must* use the OTS Encina transaction manager. The reason for this is that Artix implicitly registers an interposition resource to bridge the OTS-to-WS-AT middleware boundary. Therefore, there are really two resources in this scenario.

In summary, interposition requires additional resources as follows:

- *OTS-to-WS-AT middleware boundary*—one interposition resource is registered automatically. Applications with one explicitly registered resource must use OTS Encina.
- *WS-AT-to-OTS middleware boundary*—no interposition resource required. Applications with one explicitly registered resource may use OTS Lite.

Suppressing propagation

Once you have selected a transaction system (for example, the application loads an OTS plug-in or a WS-AT plug-in), transaction contexts are propagated by default.

It is possible, however, to suppress transaction propagation selectively using the `detachThread()` and `attachThread()` methods. After calling `detachThread()`, subsequent operation invocations do not participate in the transaction and, therefore, do not propagate any transaction context. You can re-establish an association with a transaction by calling `attachThread()`.

For more details on these functions, see "Threading" on page 49.

Threading

This chapter discusses the thread affinity of transactions and how you can modify thread affinities using the Artix transaction API.

In this chapter

This chapter discusses the following topics:

Client Threading

Overview

Artix supports a threading API that enables you to change the thread affinity of a given transaction. Using the `attachThread()` and `detachThread()` methods, you can flexibly re-assign threads to a transaction (subject to the limitations imposed by the underlying transaction system).

Default client threading model

Figure 11 shows the default threading model for transaction on the client side. When you call `beginTransaction()`, Artix creates a new transaction and attaches it to the current thread. So long as the transaction remains attached, any WSDL operations called from the current thread become part of the transaction. When you call `commitTransaction()` (or `rollbackTransaction()`, if the transaction must be aborted), the transaction is deleted.

Figure 11: *Default Client Threading Model*
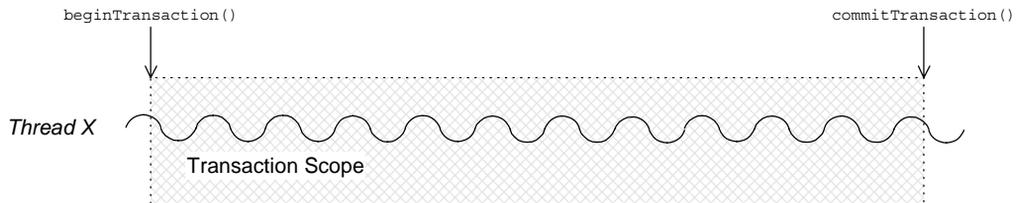
Transaction identifiers

A *transaction identifier* is an opaque identifier of type `com.iona.jbus.transaction.TransactionIdentifier` that uniquely identifies a transaction.

Controlling thread affinity

On the client side, thread affinity is controlled by the following `TransactionManager` methods:

Example 7:   *Functions for Controlling Thread Affinity*

```
public class TransactionManager
{
  public TransactionIdentifier detachThread();

  public boolean attachThread(TransactionIdentifier
    transactionIdentifier)
  throws InvalidTransactionIdentifierException

  public TransactionIdentifier getTransactionIdentifier()
    ...
}
```

These functions can be explained as follows:

- `detatchThread()`

  Detach the transaction from the current thread. After the call to `detatchThread()`, WSDL operations called from the current thread do not participate in the transaction. The returned transaction identifier can be used to re-attach the transaction to the current thread at a later stage.

- `attachThread()`

  Attach the transaction, specified by the `transactionIdentifier` argument, to the current thread.

- `getTransactionIdentifier()`

  Return the identifier of the transaction that is attached to the current thread. If no transaction is attached, return `null`.

Detaching and re-attaching a transaction to a thread

Figure 12 shows how to use the detachThread() and attachThread() methods to suspend temporarily the association between a transaction and a thread. This can be useful if, in the midst of a transaction, you need to perform some non-transactional tasks.

Figure 12: *Detaching and Re-Attaching a Transaction to a Thread*
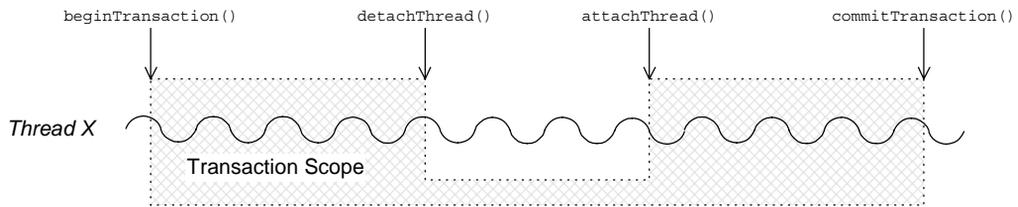
Attaching a transaction to multiple threads

Figure 13 shows how to use the getTransactionIdentifier() and attachThread() methods to associate a transaction with multiple threads. The getTransactionIdentifier() method is called from within the thread that initiated the transaction. The transaction ID can then be passed to the other threads, Y and Z, enabling them to attach the transaction.

Figure 13: *Attaching a Transaction to Multiple Threads*
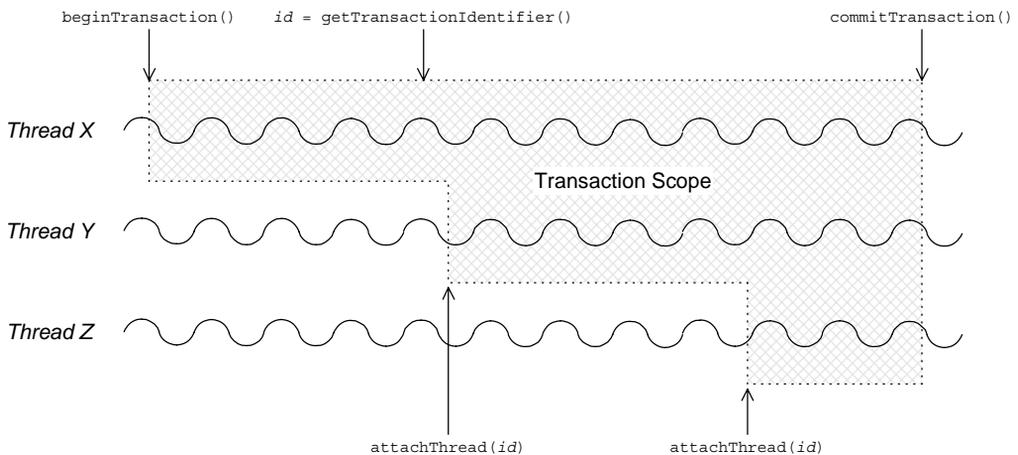
> **Note:** Some transaction systems do not allow you to associate multiple threads with a transaction. In this case, an `attachThread()` call fails (returning `false`), if you attempt to attach a second thread to the transaction.

Transferring a transaction from one thread to another

Figure 14 shows how to use the `detachThread()` and `attachThread()` methods to transfer a transaction from thread X to thread Y. The transaction ID returned from the `detachThread()` call must be passed to thread Y, enabling it to attach the transaction.

Figure 14: *Transferring a Transaction from One Thread to Another*
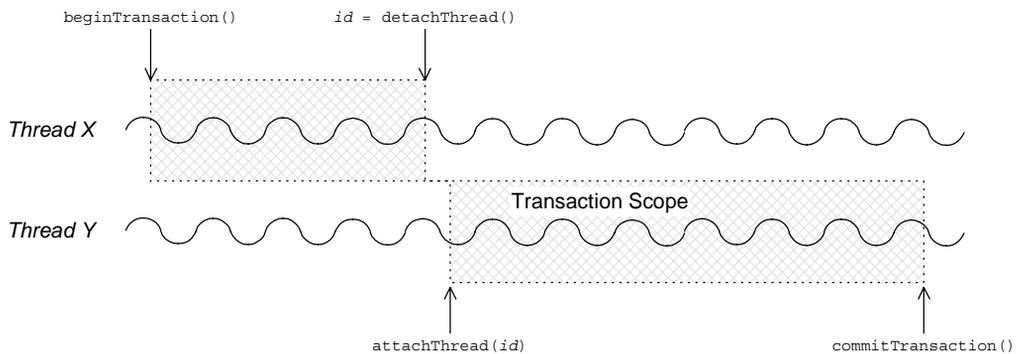
> **Note:** Some transaction systems do not allow you to transfer a transaction from one thread to another. In this case, an `attachThread()` call fails (returning `false`), unless you are re-attaching the original thread to the transaction.

Threading and XA Resources

Overview

This section discusses the following threading models for XA resources:

- Auto-association.
- Multiple registered resources.
- Multi-threaded resource connections.
- Dynamic registration.

Auto-association

When an Artix server receives a transactional request (that is, a request accompanied by a transaction context), Artix *automatically* creates an association between the current thread and locally registered resources. For each registered resource, the Artix transaction manager creates a transaction branch, which participates in the global transaction.

Figure 15 shows the sequence of events that occur when a transactional request arrives at an Artix server that has one registered resource.

Figure 15: *Auto-Association with a Single Registered Resource*
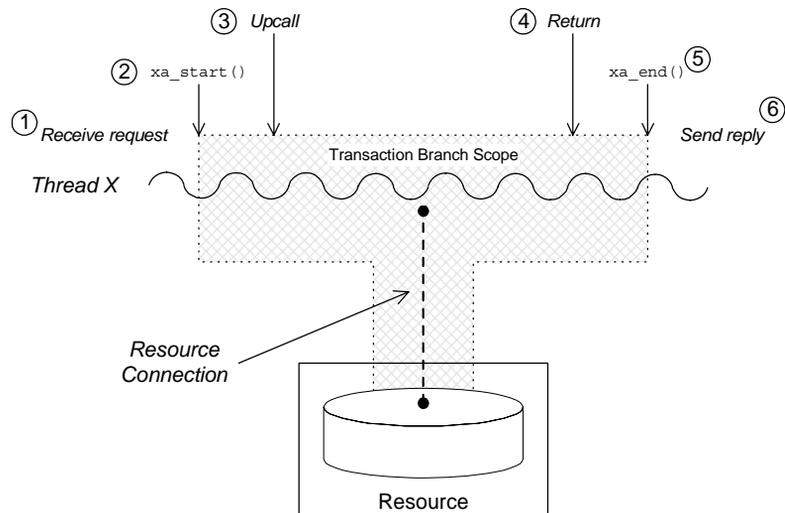
The sequence of events shown in Figure 15 on page 54 can be explained as follows:

1. *Request is received*—an operation request is received, which contains a transaction context.

2. *Artix calls* xa_start()—to create a temporary association between the current thread and the local resource. The resource creates a new transaction branch, which performs work on behalf of the global transaction.

3. *Artix calls servant function*—control is passed to the servant function that implements the WSDL operation. Any interactions and updates you make to the resource are now governed implicitly by the global transaction.

4. *Servant function returns*—control passes back to the Artix runtime.

5. *Artix calls* xa_end()—to end the association between the current thread and the resource. Effectively, the local transaction branch is terminated (but the global transaction is still active).

6. *Reply is sent*—and the thread becomes available to process another request.

Multiple registered resources

Figure 16 shows how auto-association works with multiple registered resources. When the Artix server receives a transactional request, it obtains a list of all registered resources. Artix then creates a new transaction branch for *each* resource, before making an upcall to the relevant servant function.

Figure 16: *Auto-Association with Multiple Registered Resources*
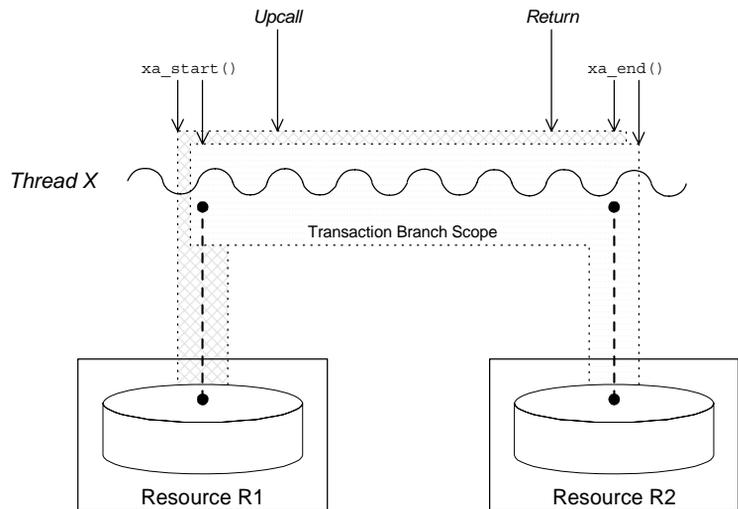
After the upcall, any application code in the servant function that interacts with one of the resources (either resource R1 or resource R2) is implicitly governed by a global transaction, where the global transaction ID has been obtained from the received transaction context.

Multi-threaded resource connections

Most modern databases offer the option of running in a *multi-threaded mode*. What this means is that instead of having a single connection to the database, which must be shared between all threads in the server, the database allows the transaction manager to open a dedicated connection for each server thread. This has the advantage of reducing contention between the server threads.

Figure 17 shows an example of a resource configured to use multi-threaded mode, where the server threads each open an independent connection to the resource. This enables the threads to access the resource concurrently.

Figure 17: *Database Resource Operating in Multi-Threaded Mode*
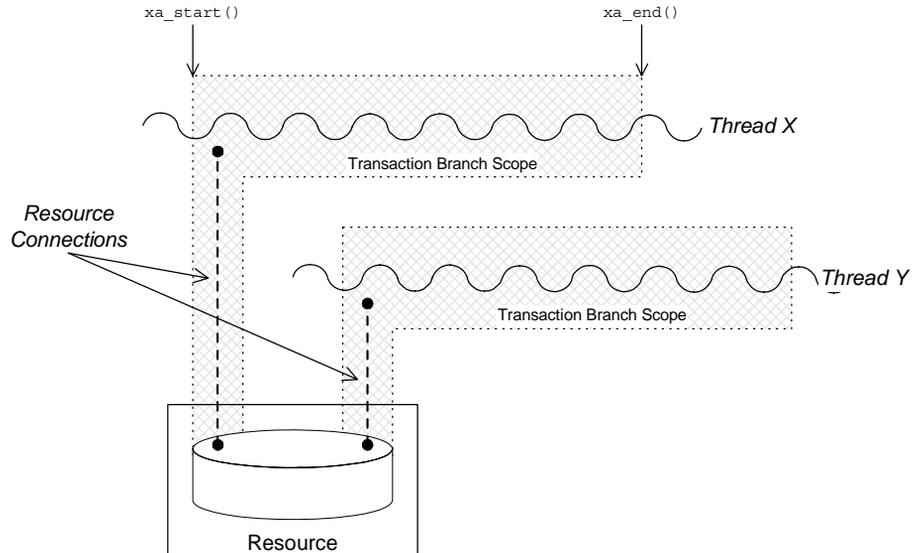
To use the multi-threaded resource mode, both the resource manager and the Artix transaction manager must be configured appropriately.

Dynamic registration

As shown in Figure 18, some XA resources support an alternative algorithm, *dynamic registration*, for associating a global transaction with a locally registered resource.

Figure 18: *Threading for a Dynamically Registered Resource*
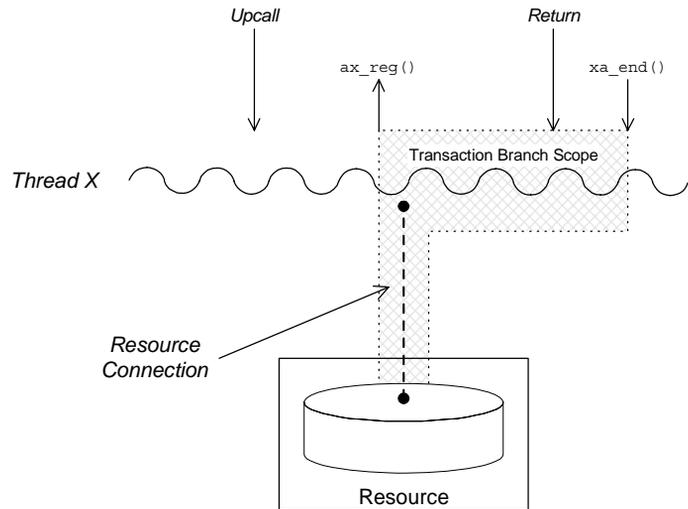
When dynamic registration is enabled, the transaction manager does *not* automatically create a transaction branch for an incoming request (that is, the transaction manager does not call xa_start()). Instead, the transaction manager waits until it receives a callback, ax_reg(), from the resource manager. This callback indicates to the transaction manager that the application code has attempted to update the resource in some way (for example, by calling EXEC SQL UPDATE). The transaction manager responds to this by creating a new transaction branch, which it associates with a global transaction (assuming the incoming request has a transaction context).

The advantage of this algorithm is that the transaction branch is created only when necessary. In some cases, if the application code does not make any resource updates, it might not be necessary to create a transaction branch at all.

Transaction Recovery

Transaction recovery is an enterprise-level feature that ensures a transaction system can cope with any kind of crash or system failure, without losing data or getting into an inconsistent state. In Artix, transaction recovery is implemented by the Encina transaction engine.

In this chapter

This chapter discusses the following topics:

Transactions Systems and Recovery

Overview

Not all of the Artix transaction systems support recovery. It is important to distinguish between the lightweight transactions systems, which are non-recoverable, and the enterprise-level transactions systems, which are recoverable. Table 1 summarizes the characteristics of the various Artix transaction systems.

Table 1: *Transaction Systems and Recoverability*

Transaction System	Single or Multiple Resources?	Recoverable?
OTS Lite	Single	*No*
OTS Encina	Multiple	*Yes*
Non-recoverable WS-AT	Multiple	*No*
Recoverable WS-AT	Multiple	*Yes*

OTS Lite

OTS Lite is a lightweight transaction system, whose programming interface is based on the CORBA OTS standard. The OTS Lite system can manage a *single* resource only and is not recoverable.

OTS Encina

OTS Encina is a complete, enterprise-level transaction system, whose programming interface is based on the CORBA OTS standard. The OTS Encina system can manage multiple resources and is recoverable.

Recoverability is the key property that distinguishes an enterprise-level transaction systems from lightweight transaction systems. Recoverability ensures that the system can always be brought back into a consistent state, irrespective of when or how a transaction participant fails.

Non-recoverable WS-AT

The non-recoverable WS-AT transaction system is a lightweight transaction system based on the WS-AtomicTransactions and WS-Coordination standards. The non-recoverable WS-AT transaction system (in contrast to OTS Lite) *can* manage multiple resources.

Recoverable WS-AT

The recoverable WS-AT transaction system is layered on top of the OTS Encina transaction engine to give enterprise-level transaction support. From Artix 4.0 onwards, WS-AT is layered over OTS by default and the relevant OTS plug-ins are automatically loaded when WS-AT is enabled. If the `plugins:ws_coordination_service:disable_tx_recovery` variable appears in your Artix configuration file, it must be set as follows to ensure recoverability:

```
# Artix Configuration File
plugins:ws_coordination_service:disable_tx_recovery = "false";
```

When WS-AT is layered over Encina, the initiation of a transaction in WS-Coordination effectively initiates an OTS transaction. The coordination context returned from the WS-Coordination service (and subsequently propagated on SOAP calls) includes an identifier indicating that it is OTS based and also includes an encoded form of the relevant OTS propagation context. That is, all transactions, including WS-AT initiated ones, are always OTS transactions. If a participant enlistment is required then the WS-AT system will completely bypass the WS-AT protocols and enlist the participant directly with OTS. This means that at completion time, OTS is aware of, and in control of, all resources in the system, be they native OTS resources, WSAT Participants, XA resources and so on.

Note: It is also possible to layer WS-AT over OTS Lite, but there is no benefit in doing so, because OTS Lite is more limited than plain WS-AT.

Transaction Recovery Scenarios

Overview

The whole point of transaction recovery is that it enables a transaction system to recover to a consistent state, irrespective of what kind of system failures occur. This section discusses a variety of different failure scenarios in order to illustrate how Encina recovers the transactional system.

In this section

This section contains the following subsections:

Server Crash before or during Prepare Phase

Overview

Figure 19 shows a scenario involving two transactional resources, one attached to server 1 and another attached to server 2, and a client, which initiates a transaction involving server 1 and server 2. This scenario uses the OTS Encina transaction system, where the OTS Encina transaction coordinator is loaded into the client and the two servers participate in the transaction.

The mode of failure described in this scenario involves server 1 crashing either before or during the prepare phase of the two-phase commit protocol.

Figure 19: *Server Crash before or during the Prepare Phase*

Steps leading to crash

As shown in Figure 19, the steps leading to a server crash before or during the prepare phase of a two-phase commit can be described as follows:

1. The client calls `begin_transaction()` to initiate the transaction.

2. Within the transaction, the client calls one or more WSDL operations on both of the remote servers.

3. The client calls `commit_transaction()` to make permanent any changes caused during the transaction.

4. The transaction coordinator initiates the prepare phase of the two-phase commit. At some point either before or during the prepare phase, server 1 crashes. That is, the transaction coordinator never receives a *vote commit* or *vote rollback* from server 1.

Transaction system recovery

If the transaction coordinator does not receive a reply from the prepare call on server 1 (for example, the connection to server 1 breaks or the transaction times out), the transaction coordinator will presume that the transaction is to be rolled back (this rule is called *presumed rollback)*.

The transaction system also rolls back the transaction on all of the other transaction participants.

Server 1 recovery

The manner in which server 1 recovers depends on whether it wrote anything into its log during the prepare phase. When server 1 re-starts after crashing, the transaction is recovered in one of the following ways:

• *No record of prepare phase in log*—in this case, server 1 knows that a transaction was begun (this is recorded in its log) and that the transaction was interrupted before the prepare phase. Server 1 automatically rolls back the transaction (presumed rollback), bringing it back to a state that is consistent with the rest of the system.

• *Prepare phase recorded in log*—in this case, it is possible that the prepare phase had completed successfully. Server 1, therefore, needs to contact the transaction coordinator to discover the outcome of the transaction. From its log, it can retrieve a recovery coordinator reference, which it uses to query the transaction state. Depending on the reply, it will either commit or roll back the transaction (in the scenario shown in Figure 19, it will be a rollback).
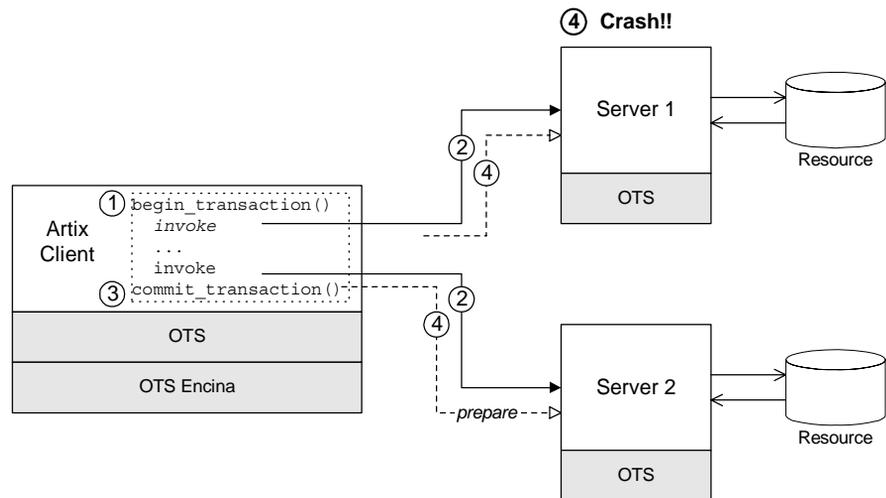
Server Crash after Prepare Phase

Overview

Figure 20 shows a scenario involving two transactional resources, one attached to server 1 and another attached to server 2, and a client, which initiates a transaction involving server 1 and server 2. This scenario uses the OTS Encina transaction system.

The mode of failure described in this scenario involves server 1 crashing *after* the prepare phase of the two-phase commit protocol.

Figure 20: *Server Crash after the Prepare Phase*
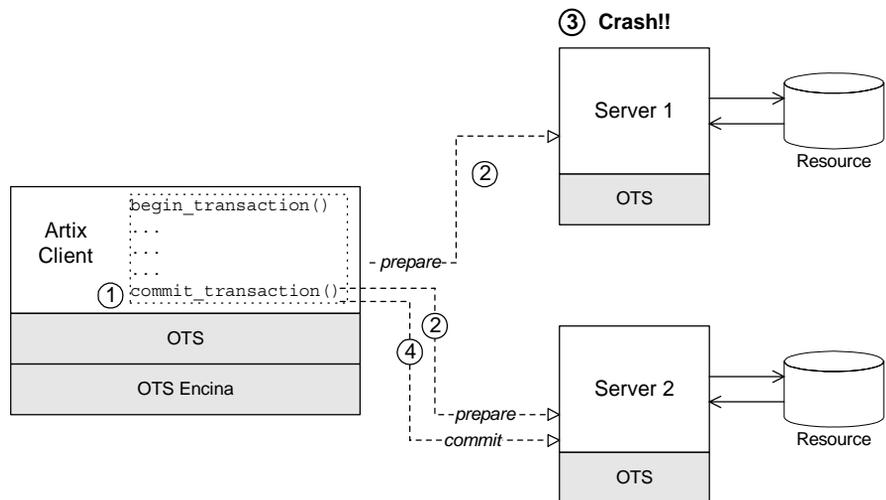
Steps leading to crash

As shown in Figure 20, the steps leading to a server crash after the prepare phase of a two-phase commit can be described as follows:

1. The client calls `commit_transaction()` to make permanent any changes caused during the transaction.

2. The transaction system performs the prepare phase by polling all of the remote transaction participants.

3. After replying to the prepare call, but before receiving the commit call, server 1 crashes. For this scenario, it is assumed that server 1 replied to the prepare call with a *vote commit*.

4. Assuming that the other transaction participants all reply to the prepare phase with a *vote commit*, the transaction coordinator decides to commit the transaction and sends a commit notification to the participants.

Transaction system recovery

If the prepare phase has completed successfully (that is, the prepare call returned from all of the transaction participants), the transaction coordinator determines the outcome of the transaction to be either *commit* or *rollback*. In the present scenario, it is assumed that the outcome is commit.

When the transaction coordinator attempts to send a commit notification to server 1, it discovers that server 1 has crashed. The transaction coordinator reacts to this situation by retrying the commit call forever.

Server 1 recovery

When server 1 is restarted, it knows from its own log that a transaction was prepared but not commited. Therefore, it expects to receive either a commit or a rollback call from the transaction coordinator. Because the transaction coordinator retries the commit call forever, server 1 is bound to receive a commit call shortly after it starts up, thereby resolving the transaction.

Transaction Coordinator Crash

Overview

Another mode of failure can occur where the process hosting the transaction coordinator crashes (for example, in Figure 20 this would be the client process). The transaction coordinator has its own log, which it uses as the basis for recovery.

Encina logs

To enable the transaction coordinator to recover gracefully after a crash, it writes whatever information would be needed for recovery into a log file or partition as it goes along.

Transaction system recovery

After a transaction coordinator crash, the possible recovery scenarios can be reduced essentially to two cases, as follows:

- *The coordinator determined the transaction outcome before crashing*—upon restarting, the transaction coordinator will try forever to notify the participants of the transaction outcome (commit or rollback).

- *The coordinator did* not *determine the transaction outcome before crashing*—the presumed rollback rule is used here. Transaction participants that were not prepared will simply presume a rollback, after a timeout has elapsed. Prepared participants will use the coordinator reference to contact the transaction coordinator and query the outcome of the transaction.

Recoverable Resources

This section describes those aspects of server side programming which enable you to update a persistent resource transactionally.

In this chapter

This chapter discusses the following topics:

Transaction Participants

Overview

When Artix uses a persistent resource, the easiest way to integrate that resource within the Artix transaction system is to enlist the resource's XA switch. If the resource does not support the XA standard, however, you need to implement a *transaction participant* instead. A transaction participant is an object usually on the server side that interfaces between the Artix transaction manager and a persistent resource. The role of the transaction participant is to receive callbacks from the transaction manager, which tell the participant whether to make pending changes permanent or whether to abort the current transaction and return the resource to its previous consistent state.

Participants in a 2-phase commit   Figure 21 shows an example of a two-phase commit involving two transaction participant instances. Any operations meant to be transactional should start by creating a transaction participant object and enlisting it with the transaction manager.

Figure 21: *Transaction Participants in a 2-Phase Commit Protocol*
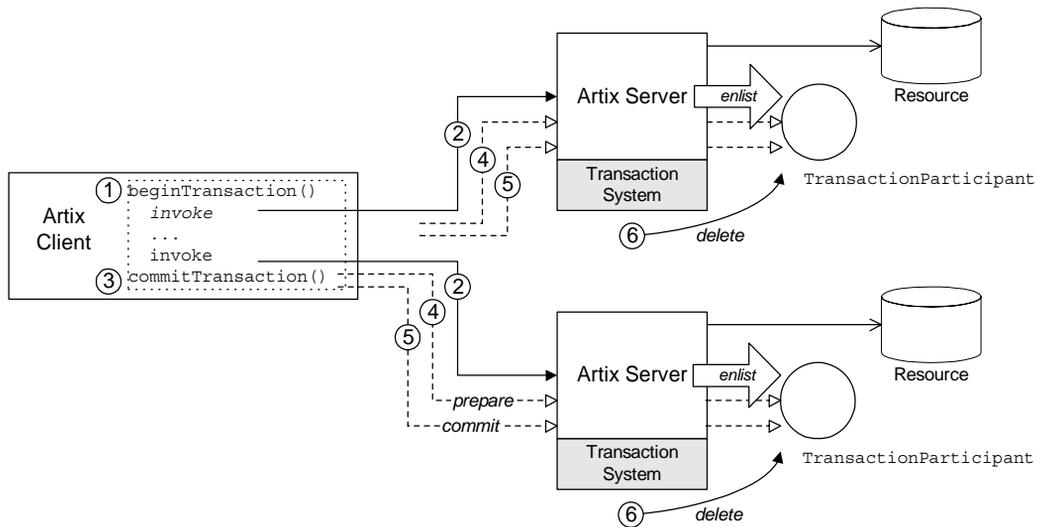
Participants in a 2-phase commit   As shown in Figure 21, the transaction participants participate in a two-phase commit as follows:

Stage	Description
1	The client calls beginTransaction() to initiate a distributed transaction.
2	Within the transaction, the client calls transactional operations on Server A and on Server B. In order to participate in the distributed transaction, the servant code creates a new transaction participant and enlists it with the transaction manager.

Stage	Description
3	The client calls `commitTransaction()` to make permanent any changes caused during the transaction.
4	The transaction system performs the prepare phase by calling `prepare()` on all of the transaction participants. Each participant can vote either to commit or to rollback the current transaction by returning a flag from the `prepare()` function.
5	The transaction system performs the commit or rollback phase by calling `commit()` or `rollback()` on all of the transaction participants.
6	When the transaction is finished, the transaction manager automatically deletes the associated transaction participant instances.

Implementing a transaction participant

To create a transaction participant, define a class that implements the `com.iona.jbus.transaction.TransactionParticipant` interface.

TransactionParticipant methods

Example 8 shows the public member functions of the TransactionParticipant interface.

Example 8: *The TransactionParticipant Interface*

```java
// Java
package com.iona.jbus.transaction;

import com.iona.jbus.BusException;

public interface TransactionParticipant
{
  void commitOnePhase() throws BusException;

  VoteOutcome prepare();

  void commit();

  void rollback();

  void setTransactionManager(TransactionManager txManager);

  String preferredTransactionManager();
}
```

1PC callback method

The following method is called during a one-phase commit:

- commitOnePhase()—this method should make permanent any changes associated with the current transaction.

2PC callback functions

The following methods are called during a two-phase commit:

- prepare()—called during *phase one* of a two-phase commit. Before returning, this method should write a recovery log to persistent storage. The recovery log should contain whatever data would be necessary to restore the system to a consistent state, in the event that the server crashes before the transaction is finished.

  Note: In some transaction systems, such as OTS Encina, the transaction manager will not call prepare() if it knows that transaction will be rolled back.

The `prepare()` function also votes on whether to commit or roll back the transaction overall, by returning one of the following vote outcomes:

♦ `VoteOutcome.VOTE_COMMIT`—vote to commit the transaction.

♦ `VoteOutcome.VOTE_ROLLBACK`—vote to roll back the transaction. For example, you would return `VOTE_ROLLBACK`, if an error occurred while attempting to write the recovery log.

♦ `VoteOutcome.VOTE_READONLY`—explicitly request *not* to be included in the commit phase of the 2PC protocol.

- `commit()`—called during *phase two* of a two-phase commit, if the transaction outcome was successful overall. The implementation of this method should make permanent any changes associated with the current transaction.

- `rollback()`—called during *phase two* of a two-phase commit, if the transaction must be aborted. The implementation of this method should undo any changes associated with the current transaction, returning the system to the state it was in before.

Getting the transaction manager

After the transaction participant is enlisted by a transaction manager instance, the transaction system calls back to pass a transaction manager to the participant. The following methods are relevant to this callback behavior:

- `preferredTransactionManager()`—called just after the participant is enlisted. The return value is a string that tells the transaction system what type of transaction manager the participant requires. The following return strings are supported:

♦ `DEFAULT_TRANSACTION_TYPE`—no preference; use the current default.

♦ `OTS_TRANSACTION_TYPE`—prefer the `OTSTransactionManager` interface (manager for CORBA OTS transactions).

♦ `WSAT_TRANSACTION_TYPE`—prefer the `WSATTransactionManager` interface (manager for WS-AtomicTransactions).

- `setTransactionManager()`—called after the `preferredTransactionManager()` call. The transaction system calls `setTransactionManager()` to pass a transaction manager of the

preferred type to the participant. If the type of transaction manager requested by the participant differs from the one currently in use, Artix uses *interposition* to simulate the preferred transaction manager type.

For more details about interposition, see "Interposition" on page 77.

Enlisting a transaction participant   Example 9 shows an example of how to enlist a participant instance in a transaction. You must enlist a participant at the start of any transactional WSDL operation. Example 9 shows a sample implementation of an operation, `write()`, which is called in the context of a transaction.

Example 9:   *Example of Enlisting a Transactional Participant*

```
public void write(int value) throws Exception
{
  Bus bus = DispatchLocals.getCurrentBus();

  TransactionSystem txSystem = bus.getTransactionSystem();

  if (txSystem.withinTransaction())
  {
    TxParticipant participant = new TxParticipant(this);

    TransactionManager txManager =
   txSystem.getTransactionManager(TransactionSystem.DEFAULT_TRAN
   SACTION_TYPE);

    txManager.enlist(participant, true);

    m_value = value;
  }
  else
  {
    System.out.println("No transaction");
    throw new BusException("Invocation not in transaction");
  }
}
```

The preceding code example can be explained as follows:

1. `DispatchLocals.getCurrentBus()` is a standard function that returns a reference to the current thread's bus instance.

2. `write()` *requires* a transaction. If it is not called in the context of a transaction, it raises an exception back to the client.

3. The `TXParticipant` class is an implementation of the `TransactionParticipant` interface.

4. The participant is enlisted in the transaction, ensuring that the participant receives callbacks either to commit or rollback any changes.

   The second parameter is a boolean flag that specifies the kind of participant:

   ♦ `true` indicates a *durable participant*, which participates in all phases of the transaction.

   ♦ `false` indicates a *volatile participant*, which is only guaranteed to participate in the prepare phase of the 2PC protocol. There is no guarantee that a volatile participant will participate in the commit phase.

Interposition

What is interposition?

Sometimes, there can be a mismatch between the transaction API used by the application code and the type of the underlying transaction system. For example, imagine that you have a legacy CORBA server that manages transactions with CORBA OTS. If you migrate this server code to a WS-AT-based Artix service, you would obtain a mismatch between the transaction API used by the application code (which is CORBA OTS-based) and the underlying transaction system (which is WS-AT).

To bridge this API mismatch, Artix uses *interposition*. With interposition, the Artix runtime provides the application code with an object of the preferred type (for example, an `OTSTransactionManager` object), but the object is merely a facade, whose calls are ultimately translated into a form suitable for the underlying transaction system (for example, WS-AT).

Interposition matrix

Artix supports interposition between *every* permutation of transaction systems. Internally, Artix converts calls made on a specific transaction API into a technology-neutral API. The calls are then converted from the technology-neutral API into one of the supported transaction APIs.

Notification Handlers

A notification handler is an object that receives callbacks to inform it about the outcome of a transaction.

In this chapter

This chapter discusses the following topics:

Introduction to Notification Handlers

Overview

A *notification handler* is an object that records the outcome of a transaction. It can be used both on the server side and on the client side. For example, you might use a notification handler to log transaction outcomes or to synchronize other events with a transaction.

Implementing a notification handler

To implement a notification handler, implement the com.iona.jbus.transaction.TransactionNotificationHandler interface.

TransactionNotificationHandler interface

Example 10 shows the TransactionNotificationHandler interface. These methods will only be called if an appropriate notification mechanism is available in the underlying transaction system.

Example 10: *The TransactionNotificationHandler Interface*

```Java
// Java
package com.iona.jbus.transaction;

public interface TransactionNotificationHandler
{
  void commitInitiated(TransactionIdentifier transactionId);

  void committed();

  void aborted();
}
```

Notification callback functions

The following notification handler functions receive callbacks from the transaction manager:

- commitInitiated()—informs the handler that a commit has been initiated. This method is called before any participants are prepared.

  Note: WS-AT does not support this notification point.

- committed()—informs the handler that the transaction completed successfully.

- aborted()—informs the handler that the transaction did not complete
  successfully and was aborted.

Enlisting a notification handler

To use a notification handler, you must enlist it with a TransactionManager object while there is a current transaction. You can enlist a notification handler at any time prior to the termination of the transaction.

Example 11 shows how to enlist a sample notification handler, NotificationHandlerImpl.

Example 11: *Example of Enlisting a Notification Handler*

```java
// Java
Bus bus = DispatchLocals.getCurrentBus();
TransactionSystem txSystem = bus.getTransactionSystem();

if (txSystem.withinTransaction())
{
  NotificationHandlerImpl notHandler = new
   NotificationHandlerImpl;

  TransactionManager txManager =
   txSystem.getTransactionManager(TransactionSystem.DEFAULT_TRAN
   SACTION_TYPE);

  txManager.enlistForNotification(notHandler);
}
```

MQ Transactions

This chapter describes how transactions are integrated with the Artix MQ transport, which integrates with the IBM MQ-Series product to provide a reliable message-oriented transport.

In this chapter

This chapter discusses the following topics:

Reliable Messaging with MQ Transactions

Overview

This section describes how to enable reliable messaging with MQ transactions in your Artix applications. MQ transactions differ in several important respects from ordinary Artix transactions, in particular:

- MQ transactions are managed by a transaction manager that is internal to the MQ-Series product.
- MQ transactions are enabled by setting the relevant attributes of a WSDL port in the WSDL contract.
- You can *not* initiate and terminate MQ transactions on the client side using the Artix transaction API (for example, the functions in `IT_Bus::TransactionSystem` are not used for MQ on the client side).

On the client side, MQ transactions follow a completely different model from Artix transactions. On the server side, however, the MQ transaction is integrated with an Artix transaction, so that an incoming message is considered to have been processed, only if the Artix transaction completes successfully on the server side.

Oneway Invocations

Oneway invocation scenario

Figure 22 shows a oneway invocation scenario, where an Artix client invokes oneway operations on an Artix server over the MQ transport with MQ transactions enabled. Because the WSDL operations are *oneway* (that is, consisting only of output messages), the MQ transport does not require a reply queue in this scenario.

Figure 22: *Oneway Operation Invoked Over an MQ Transport with MQ Transactions Enabled*
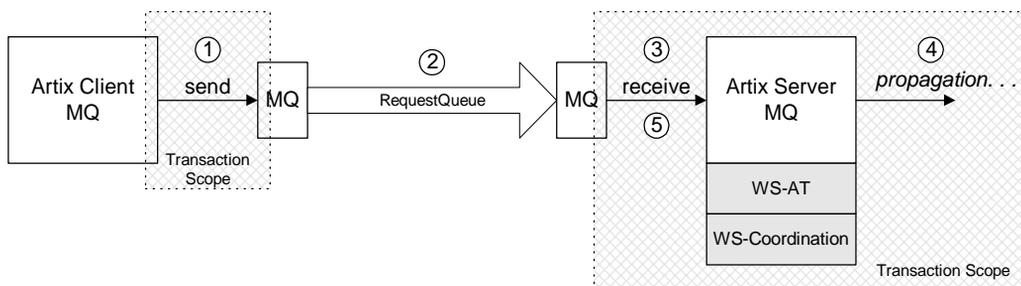
Description of oneway invocation

The oneway operation invocation shown in Figure 22 is executed in the following stages:

Stage	Description
1	When the client invokes a oneway operation over MQ, an MQ transaction is initiated. After the request message is pushed onto the client side of the MQ request queue, the MQ transaction is committed.
	Note: The client MQ transaction is local and does *not* extend beyond the client side.
2	MQ-Series is responsible for reliably transmitting the request message from the client side of the MQ transport to the server side of the MQ transport.

Stage	Description
3	When the server pulls the request message off the incoming queue, an Artix transaction is initiated before dispatching the request to the relevant Artix servant.
4	If the Artix servant now invokes operations on some other Artix servers, these invocations occur within a transaction context. Hence, these follow-on invocations propagate a transaction context (for example, a WS-AT context) and enable the remote servers to participate in the transaction.
5	If the operation completes its work successfully, the transaction is committed and the request message permanently disappears from the queue. On the other hand, if the operation is unsuccessful, the transaction is rolled back and the request message is pushed *back* onto the queue. The request message is immediately reprocessed (the maximum number of times the message can be processed is determined by the queue's backout threshold—see "Configuring the backout threshold" on page 91).

Oneway client configuration

To enable transactional semantics for a client that invokes oneway operations over the MQ transport, you should define a WSDL port as shown in Example 12.

Example 12: *WSDL Port Configuration for Oneway Client Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        <mq:client  QueueManager="MY_DEF_QM"
                    QueueName="HW_REQUEST"


                    AccessMode="send"
                    CorrelationStyle="correlationId"
                    Transactional="internal"
                    Delivery="persistent"
                    UsageStyle="peer"
        />
        ...
    </wsdl:port>
</wsdl:service>
```

Because the invocation is oneway, there is no need to specify a reply queue manager. To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Oneway server configuration

On the server side, you must configure both the WSDL contract and the Artix configuration file appropriately for using MQ transactions.

WSDL Contract Configuration

To enable transactional semantics for a server that receives oneway invocations over the MQ transport, you should define a WSDL port as shown in Example 13.

Example 13: *WSDL Port Configuration for Oneway Server Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        ...
        <mq:server  QueueManager="MY_DEF_QM"
                    QueueName="HW_REQUEST"


                    AccessMode="receive"
                    CorrelationStyle="correlationId"
                    Transactional="internal"
                    Delivery="persistent"
                    UsageStyle="peer"
        />
    </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Artix Configuration File

On the server side, Artix initiates a transaction whenever it receives a request message from the MQ transport. Because this transaction is managed by an Artix transaction manager, you must load and configure one of the Artix transaction systems (for example, OTS or WS-AT).

For details of how to select a transaction system, see "Selecting a Transaction System" on page 19.

Synchronous Invocations

Synchronous invocation scenario

Figure 23 shows a synchronous invocation scenario, where an Artix client invokes normal operations on an Artix server over the MQ transport with MQ transactions enabled. Because the WSDL operations are *synchronous* (that is, consisting of output messages and input messages), the MQ transport requires a reply queue.

Figure 23: *Synchronous Operation Invoked Over the MQ Transport with MQ Transactions Enabled*
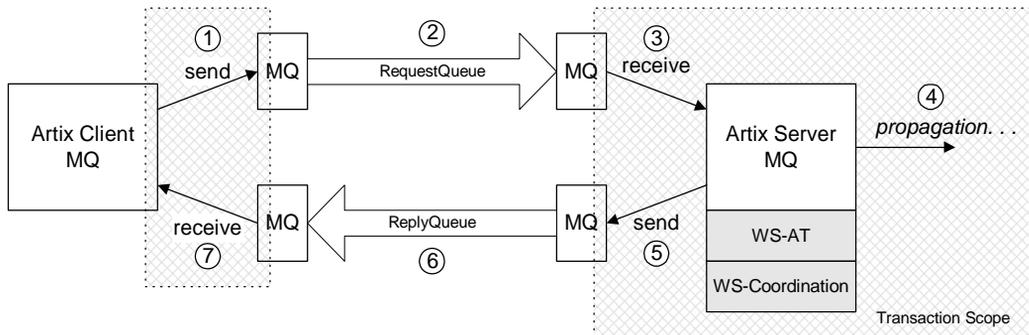
Description of synchronous invocation

The synchronous operation invocation shown in Figure 23 is executed in the following stages:

Stage	Description
1	When the client invokes a synchronous operation over MQ, an MQ transaction is initiated.
2	MQ-Series is responsible for reliably transmitting the request message from the client side of the MQ transport to the server side of the MQ transport.

Stage	Description
3	When the server pulls the request message off the incoming queue, an Artix transaction is initiated before dispatching the request to the relevant Artix servant.
4	If the Artix servant now invokes operations on some other Artix servers, these invocations occur within a transaction context. Hence, these follow-on invocations propagate a transaction context (for example, a WS-AT context) and enable the remote servers to participate in the transaction.
5	If the operation completes its work successfully, the transaction is committed, the request message permanently disappears from the request queue, and a reply message gets pushed onto the reply queue. On the other hand, if the operation is unsuccessful, the transaction is rolled back. No reply message is sent and the request message is pushed *back* onto the request queue. The request message is immediately reprocessed (the maximum number of times the message can be processed is determined by the request queue's backout threshold—see "Configuring the backout threshold" on page 91).
6	MQ-Series is responsible for reliably transmitting the reply message from the server side of the MQ transport to the client side of the MQ transport.
7	When the client receives the reply message, the synchronous operation call returns and the client transaction is committed. Because the client is independent of the server side transaction, however, it is not possible for the client code to receive a rollback exception from the server. It is possible to manage blocked calls by defining the `Timeout` attribute on the `mq:client` element in the WSDL contract. If the timeout is exceeded, an exception will be thrown.

Synchronous client configuration

To enable transactional semantics for a client that invokes synchronous operations over the MQ transport, you should define a WSDL port as shown in Example 14.

Example 14: *WSDL Port Configuration for Synchronous Client Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        <mq:client  QueueManager="MY_DEF_QM"
                    QueueName="HW_REQUEST"
                    ReplyQueueManager="MY_DEF_QM"
                    ReplyQueueName="HW_REPLY"
                    AccessMode="send"
                    CorrelationStyle="correlationId"
                    Transactional="internal"
                    Delivery="persistent"
                    UsageStyle="responder"
        />
        ...
    </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the Transactional attribute to internal and the Delivery attribute to persistent.

Synchronous server configuration

On the server side, you must configure both the WSDL contract and the Artix configuration file appropriately for using MQ transactions.

WSDL Contract Configuration

To enable transactional semantics for a server that receives synchronous invocations over the MQ transport, define a WSDL port as shown in Example 15.

Example 15: *WSDL Port Configuration for Synchronous Server Over MQ*

```
<wsdl:service name="MQService">
    <wsdl:port binding="tns:BindingName" name="PortName">
        ...
        <mq:server  QueueManager="MY_DEF_QM"
                    QueueName="HW_REQUEST"
                    ReplyQueueManager="MY_DEF_QM"
                    ReplyQueueName="HW_REPLY"
                    AccessMode="receive"
```

Example 15: *WSDL Port Configuration for Synchronous Server Over MQ*

```
                    CorrelationStyle="correlationId"
                    Transactional="internal"
                    Delivery="persistent"
                    UsageStyle="responder"
         />
     </wsdl:port>
</wsdl:service>
```

To enable transactions, you must set the `Transactional` attribute to `internal` and the `Delivery` attribute to `persistent`.

Artix Configuration File

On the server side, Artix initiates a transaction whenever it receives a request message from the MQ transport. Because this transaction is managed by an Artix transaction manager, you must load and configure one of the Artix transaction systems (for example, OTS or WS-AT).

For details of how to select a transaction system, see "Selecting a Transaction System" on page 19.

Configuring the backout threshold

You can configure the backout threshold using the `runmqsc` command-line tool, which is provided as part of the *MQ-Series* product. To configure a queue to use backouts, set the following MQ attributes:

- `BOTHRESH`—the backout threshold, which defines the maximum number of times a message can be pushed back onto the queue.
- `BOQNAME`—the backout queue name. If the current backout count equals the backout threshold, Artix puts the message onto the backout queue whose name is given by `BOQNAME`.

Hence, the `BOQNAME` queue would contain all of the messages that have been rolled back more than `BOTHRESH` times. The administrator can then manually examine the messages stored in the `BOQNAME` queue and take appropriate remedial action.

For more details about how to set MQ attributes, see your *MQ-Series* user documentation.

Accessing the backout count

On the server side, you can obtain the backout count for the current message using Artix contexts. To access the current backout count, perform the following steps:

1. Retrieve the server context identified by the `IT_ContextAttributes::MQ_INCOMING_MESSAGE_ATTRIBUTES` QName.

2. Cast the returned context instance to the `IT_ContextAttributes::MQMessageAttributesType` type.

3. Invoke the `getBackoutCount()` function to access the current backout count.

For more details about programming with Artix contexts, see the *JAX-RPC Programmer's Guide*.

Router Propagating MQ Transactions

Router scenario

Figure 24 shows a router scenario, where a message propagates through the router with MQ transactions enabled. In this particular scenario, both the router's source endpoint and the router's destination endpoint are configured to use the MQ transport. It would also be feasible, however, to configure the router's destination endpoint to use a different transport—for example, a transactional SOAP/HTTP transport.

Figure 24: *Router Propagating an MQ Transaction*
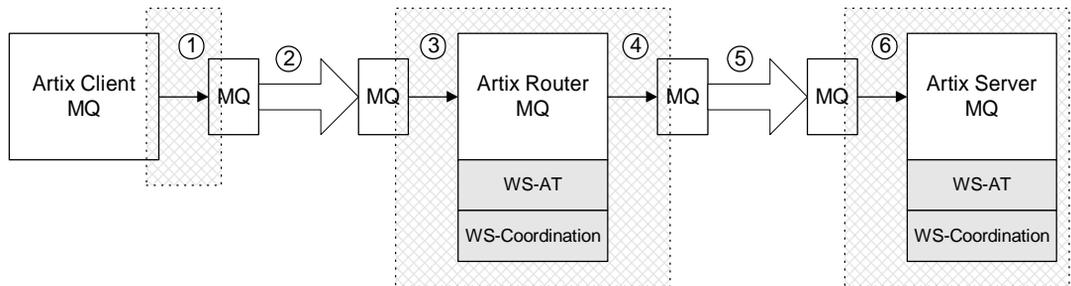
Description of router invocation

The router invocation shown in Figure 24 is executed in the following stages:

Stage	Description
1	When the client invokes a oneway operation over MQ, an MQ transaction is initiated. After the request message is pushed onto the client side of the MQ request queue, the MQ transaction is committed.
2	MQ-Series is responsible for reliably transmitting the request message from the client side of the MQ transport to the router side of the MQ transport.
3	When the router pulls the request message off the incoming queue, an Artix transaction is initiated.

Stage	Description
4	The router routes the request message to the appropriate destination endpoint. In this example, the destination endpoint uses the MQ transport.
5	MQ-Series is responsible for reliably transmitting the request message from the router side of the MQ transport to the target server side of the MQ transport.
6	When the target server pulls the request message off the incoming queue, an Artix transaction is initiated.

Router configuration

The router *must* be configured to load a transaction coordinator, because the router is responsible for initiating Artix transactions whenever it receives an MQ request message. That is, you need to add one of the following plug-ins to the `orb_plugins` list in the Artix configuration (depending on your preferred transaction system): `ws_coordination_service`, `ots_lite`, or `ots_encina`.

For details of how to select a transaction system, see "Selecting a Transaction System" on page 19.

Target server configuration

In this particular scenario (where the destination endpoint is an MQ endpoint), it is also necessary to configure the target server to load a transaction coordinator plug-in.

On the other hand, if the destination endpoint was configured to use a different transport—for example, SOAP/HTTP—it would *not* be necessary to load a transaction coordinator and you could configure the target server in the same way as the server examples described in "Selecting a Transaction System" on page 19. In this case, the target server could participate directly in the transaction initiated in the router and the router's transaction coordinator would be responsible for coordinating the transaction.

Index