PROGRESS
SOFTWARE

Artix™ ESB

Transactions Guide, C++

Version 5.5, December 2008

Updated: December 9, 2008

Contents

List of Tables

List of Figures

Preface

What is Covered in this Book

This book explains how to program and configure Artix transactions in C++.

Who Should Read this Book

This guide is intended for Artix C++ 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

CHAPTER 1

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 functions (`begin_transaction()`, `commit_transaction()` and `rollback_transaction()`) 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

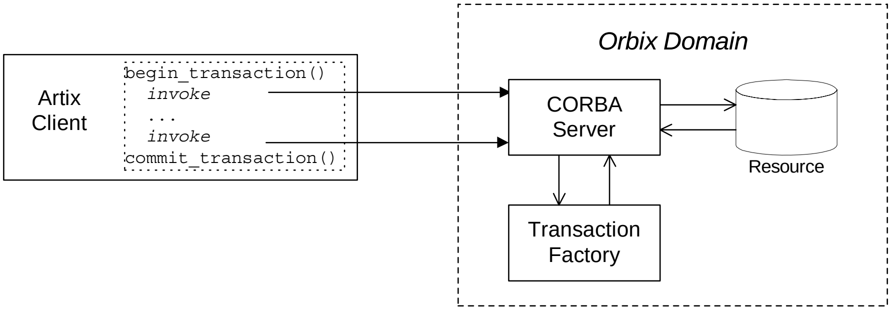
On the server side, an API is provided that enables you to implement *transaction participants* (sometimes referred to as transactional resources). Using transaction participants, you can implement servers that participate in a distributed transaction with the ACID transaction properties (*Atomicity*,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

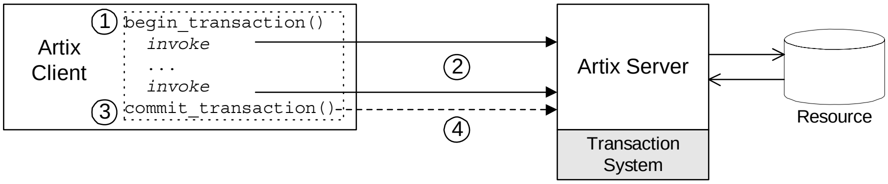
The underlying transaction system used by Artix can be replaced within a pluggable framework. Currently, the following transaction systems are supported by Artix:

- 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 `begin_transaction()` 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 `commit_transaction()` to make permanent any changes caused during the transaction (alternatively, the client could call `rollback_transaction()` 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.

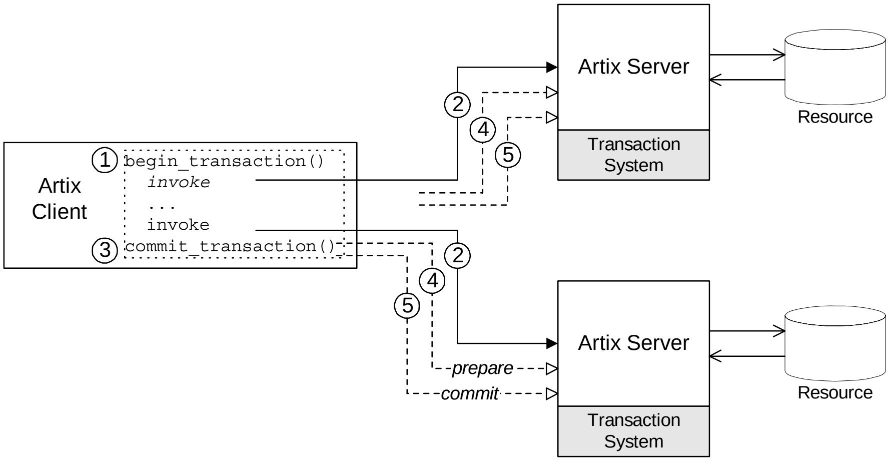
Two-phase commit

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 `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 (alternatively, the client could call `rollback_transaction()` 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.

X/Open Distributed Transaction Processing

Overview

The X/Open Distributed Transaction Processing (DTP) architecture is a technical standard published by the *Open Group*. The X/Open DTP architecture enables you to integrate resources relatively easily into a distributed transaction system.

In this section

This section contains the following subsections:

X/Open DTP Architecture

Overview

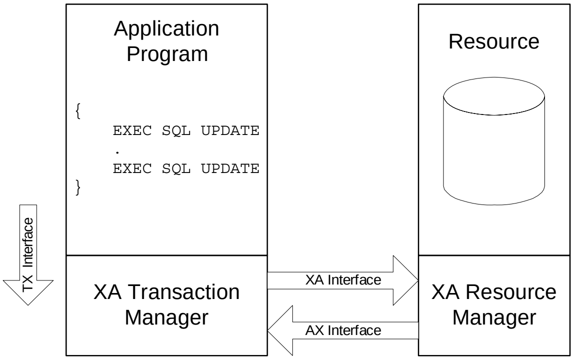
This subsection provides a brief overview of the X/Open Distributed Transaction Processing (DTP) architecture, also known as the *XA specification*. For a complete description of the X/Open DTP standard, you can download the XA specification from the following Web page:

http://www.opengroup.org/bookstore/catalog/c193.htm

DTP model

Figure 4 shows an overview of the X/Open DTP model, showing the basic components and the interfaces between them. The key idea of the X/Open architecture is that responsibility for managing transactions in a distributed system must be divided between two components: a *transaction manager* and a *resource manager*. This division would be unnecessary for local transactions, which could be managed happily by a resource manager alone, but it is essential for distributed transactions, where the mechanisms for coordinating global transactions (that is, starting, committing, and rolling back) are implemented separately from the resource manager.

Figure 4: *The X/Open DTP Architecture*



Resource

A *resource* is any part of the system that could undergo a persistent change. In most cases, a resource represents some form of persistent storage (such as a database), but it could also represent, for example, the mechanism in an Automated Teller Machine that tenders cash to customers.

Resource manager

A *resource manager* manages part of a computer's shared resources. In particular, the resource manager must be capable of grouping resource operations into transactions and either committing or rolling back those transactions in response to calls from the transaction manager (mediated by the XA interface).

For example, the Oracle DB with an XA switch is an XA-compliant resource manager.

Transaction manager

A *transaction manager* is responsible for coordinating transactions across a distributed system. The transaction manager coordinates decisions to commit or roll back a global transaction and is also responsible for coordinating failure recovery.

For example, the OTS Encina transaction manager implements the 2-phase commit protocol for global transactions.

Global transaction

A *global transaction* is a transaction that spans multiple processes and multiple resources in a distributed system. To manage a global transaction properly, it is necessary to ensure that the updates made to different resources in different processes can be committed atomically (or rolled back) at the end of the transaction.

Transaction branch

Because a global transaction is spread over a distributed system, work can be done on the global transaction in different processes. Moreover, within each process, work can be done in different resource managers (for example, you might have an Oracle XA resource manager and an MQ-Series resource manager both registered within the same process). Hence, it is useful to introduce the concept of a *transaction branch*, which identifies the work done on a global transaction by *each* resource manager in *each* process. The total work done on a global transaction is, therefore, equal to the sum of the work done in all of its branches.

XA interfaces

The XA architecture defines a suite of interfaces that mediate the interaction between the various components of the XA DTP model, as follows:

- *XA interface*—a collection of functions that the transaction manager can call on a resource manager in order to coordinate local and distributed transactions. This interface is fully supported by Artix, both in the role of transaction manager (where Artix manages foreign resource managers through the XA interface) and in the role of resource manager (where Artix is controlled by a foreign transaction manager).
- *AX interface*—a collection of functions that the resource manager can call back on the transaction manager. This interface is used internally by Artix to implement the *dynamic registration optimization*. See "Dynamic Registration Optimization" on page 81 for more details.
- *TX interface*—a collection of functions that perform transaction demarcation (beginning, committing and rolling back transactions) by calling on the transaction manager. Artix does *not* implement the TX interface; you use the demarcation functions provided on the `IT_Bus::TransactionSystem` class instead.

X/Open XA Interface

Overview

The X/Open XA interface is the interface that a transaction manager uses to control the committing or rolling back of a transaction branch in a resource manager. The great convenience of the XA interface is that it provides a simple mechanism for integrating a resource into a distributed transaction system. The XA interface effectively enables you to *plug in* a resource manager into a distributed transaction system.

For example, if you want to integrate an Oracle DB into the OTS Encina distributed transaction system (which is one of the transaction systems supported by Artix), you would simply register Oracle's XA switch with Artix. This requires no more than two or three lines of code in your application program. Once you have registered the Oracle XA switch, the Oracle DB is able to partake in distributed transactions managed by OTS Encina.

XA switch type

XA defines a set of C-function pointers, and a C-struct that holds these function pointers, `xa_switch_t` (see `orbix_sys/xa.h`) as shown in Example 1.

Example 1: *The XA Switch Type, xa_switch_t*

```
/* C */
struct xa_switch_t
{
    char name[RMNAMESZ]; /* name of resource manager */
    long flags; /* resource manager specific options */
    long version; /* must be 0 */
    int (*xa_open_entry) /* xa_open function pointer */
    (char *, int, long);
    int (*xa_close_entry) /* xa_close function pointer */
    (char *, int, long);
    int (*xa_start_entry) /* xa_start function pointer */
    (XID *, int, long);
    int (*xa_end_entry) /* xa_end function pointer */
    (XID *, int, long);
    int (*xa_rollback_entry) /* xa_rollback function pointer */
    (XID *, int, long);
    int (*xa_prepare_entry) /* xa_prepare function pointer */
    (XID *, int, long);
    int (*xa_commit_entry) /* xa_commit function pointer */
```

Example 1: *The XA Switch Type, xa_switch_t*

```
    (XID *, int, long);
    int (*xa_recover_entry) /* xa_recover function pointer */
    (XID *, long, int, long);
    int (*xa_forget_entry) /* xa_forget function pointer */
    (XID *, int, long);
    int (*xa_complete_entry) /* xa_complete function pointer */
    (int *, int *, int, long);
};
```

Function pointers

The function pointers provided by the `xa_switch_t` struct point to the following XA functions:

- `xa_open()` and `xa_close()`—the `xa_open()` function opens a connection to the resource. For example, in a single-threaded application, the transaction manager would usually call `xa_open()` as it starts up.

  The `xa_close()` function closes the connection to the resource. For example, the transaction manager would usually call `xa_close()` as it shuts down.

- `xa_start()` and `xa_end()`—the transaction manager calls `xa_start()` before doing any work on a transaction branch. At the end of the work, the transaction manager calls `xa_end()`.

  The `xa_start()` and `xa_end()` functions are closely related to the XA threading model (see "Threading and XA Resources" on page 105). The `xa_start()` function creates an association between the current thread and a transaction branch, and the `xa_end()` function ends the association. By passing in the appropriate flag, it is also possible for `xa_end()` to temporarily *suspend* the association between the current thread and the transaction branch and for `xa_start()` to *resume* the association.

- `xa_prepare()`, `xa_commit()`, and `xa_rollback()`—the transaction manager calls these functions in the course of the 1-phase and 2-phase commit protocols.

- `xa_recover()` and `xa_forget()`—the transaction manager can call these functions to recover after a system crash. Typically, a transaction manager provides a recovery tool to manage the recovery process.

Providing an XA switch instance

Each XA resource manager must provide a global instance of the `xa_switch_t` type. For example, this might be provided either as a global `xa_switch_t` struct or as the return value from a global function. The mechanism for obtaining an `xa_switch_t` instance is *not* standardised and varies from product to product.

For example, Oracle provides a global `xa_switch_t` instance called `xaosw`.

CHAPTER 2

Getting Started with Transactions

This chapter discusses a simple demonstration scenario involving a client and two remote servers. The servers enlist XA resources, which are responsible for integrating the servers' persistent storage with the Artix transaction system.

In this chapter

This chapter discusses the following topics:

Sample Scenario

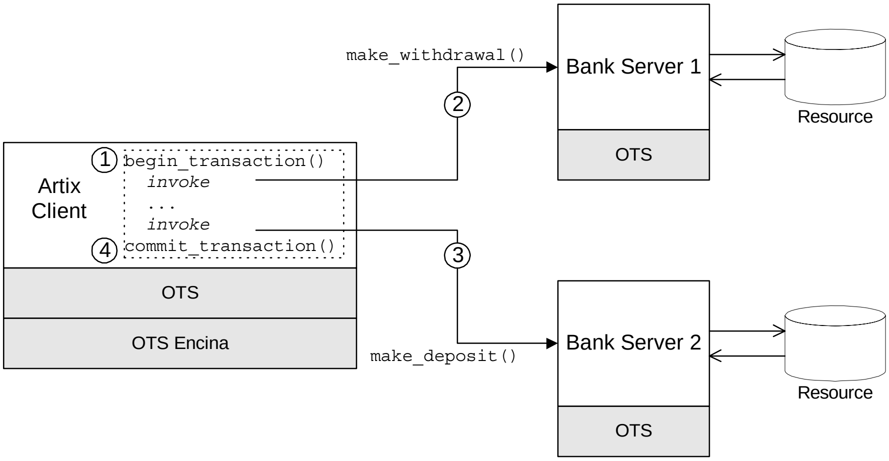
Overview

This section describes a sample scenario involving a funds transfer between two different bank servers, where each bank server is a transactional resource. This scenario is used as the basis for the examples discussed in the rest of this chapter.

Bank example

Figure 5 shows the outline of a scenario involving a funds transfer between two bank accounts, which are located on different servers, Bank Server 1 and Bank Server 2. This scenario assumes that the application is using the OTS transaction system. In particular, the client loads the OTS Encina plug-in, which is responsible for coordinating the global transactions.

Figure 5: *Bank Scenario with Transactions*



Funds transfer

The scenario shown in Figure 5 can be described as follows:

1. The client initiates a transaction by calling the `IT_Bus::TransactionSystem::begin_transaction()` function.
2. Within the scope of the transaction, the client invokes the `make_withdrawal()` operation on an account in Bank Server 1, in order to withdraw a sum of money. The operation request is accompanied by a transaction context.
3. The client invokes the `make_deposit()` operation on another account in Bank Server 2, in order to deposit the sum of money.
4. The client calls the `IT_Bus::TransactionSystem::commit_transaction()` to commit the transaction. The Artix transaction manager then uses a two-phase commit protocol to commit the changes to Bank Server 1 and Bank Server 2.

Bank WSDL contract

Example 2 shows the WSDL contract for the Bank example that is described in this section. There are two port types in this contract, `Bank` and `Account`. For each of the two port types there is a SOAP binding, `BankBinding` and `AccountBinding`.

Example 2: *Bank WSDL Contract*

```
<definitions targetNamespace="http://www.iona.com/demos/transactions/bank"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:bank="http://schemas.iona.com/demos/transactions/bank"
    xmlns:wsa="http://www.w3.org/2005/03/addressing"
    xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
    xmlns:tns="http://www.iona.com/demos/transactions/bank"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema" >
    <types>
        <schema elementFormDefault="qualified"
            targetNamespace="http://schemas.iona.com/demos/transactions/bank"
            xmlns="http://www.w3.org/2001/XMLSchema"
            xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/">
            <import namespace="http://www.w3.org/2005/03/addressing"/>
            <complexType name="AccountIDsType">
                <sequence>
                    <element maxOccurs="unbounded" minOccurs="0" name="name"
                        type="xsd:string"/>
```

Example 2: *Bank WSDL Contract*

```
            </sequence>
        </complexType>
        <complexType name="list_accountsInputData">
            <sequence/>
        </complexType>
        <complexType name="list_accountsOutputData">
            <sequence>
                <element name="return" type="bank:AccountIDsType"/>
            </sequence>
        </complexType>
        <element name="list_accounts" type="bank:list_accountsInputData"/>
        <element name="list_accountsResponse" type="bank:list_accountsOutputData"/>
        <complexType name="create_accountInputData">
            <sequence>
                <element name="account_id" type="xsd:string"/>
            </sequence>
        </complexType>
        <complexType name="create_accountOutputData">
            <sequence>
                <element name="return" type="wsa:EndpointReferenceType"/>
            </sequence>
        </complexType>
        <element name="create_account" type="bank:create_accountInputData"/>
        <element name="create_accountResponse" type="bank:create_accountOutputData"/>
        <complexType name="get_accountInputData">
            <sequence>
                <element name="account_id" type="xsd:string"/>
            </sequence>
        </complexType>
        <complexType name="get_accountOutputData">
            <sequence>
                <element name="return" type="wsa:EndpointReferenceType"/>
            </sequence>
        </complexType>
        <element name="get_account" type="bank:get_accountInputData"/>
        <element name="get_accountResponse" type="bank:get_accountOutputData"/>
        <complexType name="delete_accountInputData">
            <sequence>
                <element name="account_id" type="xsd:string"/>
            </sequence>
        </complexType>
        <complexType name="delete_accountOutputData">
            <sequence/>
        </complexType>
        <element name="delete_account" type="bank:delete_accountInputData"/>
```

Example 2: *Bank WSDL Contract*

```
            <element name="delete_accountResponse" type="bank:delete_accountOutputData"/>
            <complexType name="get_balanceInputData">
                <sequence/>
            </complexType>
            <complexType name="get_balanceOutputData">
                <sequence>
                    <element name="return" type="xsd:double"/>
                </sequence>
            </complexType>
            <element name="get_balance" type="bank:get_balanceInputData"/>
            <element name="get_balanceResponse" type="bank:get_balanceOutputData"/>
            <complexType name="make_depositInputData">
                <sequence>
                    <element name="amount" type="xsd:double"/>
                </sequence>
            </complexType>
            <complexType name="make_depositOutputData">
                <sequence/>
            </complexType>
            <element name="make_deposit" type="bank:make_depositInputData"/>
            <element name="make_depositResponse" type="bank:make_depositOutputData"/>
            <complexType name="make_withdrawlInputData">
                <sequence>
                    <element name="amount" type="xsd:double"/>
                </sequence>
            </complexType>
            <complexType name="make_withdrawlOutputData">
                <sequence/>
            </complexType>
            <element name="make_withdrawl" type="bank:make_withdrawlInputData"/>
            <element name="make_withdrawlResponse" type="bank:make_withdrawlOutputData"/>
        </schema>
    </types>
    <message name="list_accounts">
        <part element="bank:list_accounts" name="parameters"/>
    </message>
    <message name="list_accountsResponse">
        <part element="bank:list_accountsResponse" name="parameters"/>
    </message>
    <message name="create_account">
        <part element="bank:create_account" name="parameters"/>
    </message>
    <message name="create_accountResponse">
        <part element="bank:create_accountResponse" name="parameters"/>
    </message>
```

Example 2: *Bank WSDL Contract*

```
    <message name="get_account">
        <part element="bank:get_account" name="parameters"/>
    </message>
    <message name="get_accountResponse">
        <part element="bank:get_accountResponse" name="parameters"/>
    </message>
    <message name="delete_account">
        <part element="bank:delete_account" name="parameters"/>
    </message>
    <message name="delete_accountResponse">
        <part element="bank:delete_accountResponse" name="parameters"/>
    </message>
    <message name="get_balance">
        <part element="bank:get_balance" name="parameters"/>
    </message>
    <message name="get_balanceResponse">
        <part element="bank:get_balanceResponse" name="parameters"/>
    </message>
    <message name="make_deposit">
        <part element="bank:make_deposit" name="parameters"/>
    </message>
    <message name="make_depositResponse">
        <part element="bank:make_depositResponse" name="parameters"/>
    </message>
    <message name="make_withdrawl">
        <part element="bank:make_withdrawl" name="parameters"/>
    </message>
    <message name="make_withdrawlResponse">
        <part element="bank:make_withdrawlResponse" name="parameters"/>
    </message>
    <portType name="Bank">
        <operation name="list_accounts">
            <input message="tns:list_accounts" name="list_accounts"/>
            <output message="tns:list_accountsResponse" name="list_accountsResponse"/>
        </operation>
        <operation name="create_account">
            <input message="tns:create_account" name="create_account"/>
            <output message="tns:create_accountResponse" name="create_accountResponse"/>
        </operation>
        <operation name="get_account">
            <input message="tns:get_account" name="get_account"/>
            <output message="tns:get_accountResponse" name="get_accountResponse"/>
        </operation>
        <operation name="delete_account">
            <input message="tns:delete_account" name="delete_account"/>
```

Example 2: *Bank WSDL Contract*

```
            <output message="tns:delete_accountResponse" name="delete_accountResponse"/>
        </operation>
    </portType>

    <portType name="Account">
        <operation name="get_balance">
            <input message="tns:get_balance" name="get_balance"/>
            <output message="tns:get_balanceResponse" name="get_balanceResponse"/>
        </operation>
        <operation name="make_deposit">
            <input message="tns:make_deposit" name="make_deposit"/>
            <output message="tns:make_depositResponse" name="make_depositResponse"/>
        </operation>
        <operation name="make_withdrawl">
            <input message="tns:make_withdrawl" name="make_withdrawl"/>
            <output message="tns:make_withdrawlResponse" name="make_withdrawlResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Client Example

Overview

This section describes a transactional Artix client that connects to two remote transactional Artix servers, server A and server B. The client uses the Artix transaction demarcation API to delimit the transaction. The client must also be configured to load a transaction system plug-in (see "Selecting a Transaction System" on page 51).

C++ demonstration code

The bank client demonstration code is located in the following directory:

ArtixInstallDir`/cxx_java/samples/transactions/common/src`
`/clients/cxx_bank_client`

C++ example

Example 3 shows how to use the transaction demarcation functions in an Artix client. Two remote servers, bank server A and bank server B, participate in the transaction. Hence, this example requires a two-phase commit protocol.

Example 3: *C++ Bank Client Example*

```
  // C++
1 BankClient * bank_1_proxy = /* Obtain 1st bank proxy */ ;
  BankClient * bank_2_proxy = /* Obtain 2nd bank proxy */ ;

  AccountClient * acc_1;
  AccountClient * acc_2;

  try {
2     WS_Addressing::EndpointReferenceType acc_1_ref;
      bank_1_proxy->get_account("account_1", acc_1_ref);
      acc_1 = new AccountClient(acc_1_ref, bus);

3     WS_Addressing::EndpointReferenceType acc_2_ref;
      bank_2_proxy->get_account("account_2", acc_2_ref);
      acc_2 = new AccountClient(acc_2_ref, bus);
  }
  catch (const IT_Bus::Exception & access_balance_ex)
  {
      String err_msg("ERROR - account balance access failure! : ");
      err_msg += access_balance_ex.message();
      throw IT_Bus::Exception(err_msg);
```

Example 3: *C++ Bank Client Example*

```
  }

4 try {
5     bus->transactions().begin_transaction();

      acc_1->make_withdrawl(2000.00);
      acc_2->make_deposit(2000.00);

6     bus->transactions().commit_transaction(true);

      display_balances(acc_1, bank_1_id, acc_2, bank_2_id);
  }
7 catch (const IT_Bus::Exception & transfer_ex)
  {
      String err_msg("ERROR - funds transfer failure! : ");
      err_msg += transfer_ex.message();
8     if (bus->transactions().within_transaction())
      {
9         bus->transactions().rollback_transaction();
      }
      throw IT_Bus::Exception(err_msg);
  }
```

The preceding code example can be explained as follows:

1. The bank proxies, `bank_1_proxy` and `bank_2_proxy`, provide the initial connections to bank server A and bank server B, respectively.

   In the demonstration code (not shown here), each bank server writes a reference to a file which is then read by the client (this presupposes that the clients and servers can both access the same file system).
2. Obtain a proxy to an account in bank server A by calling `get_account()` on `bank_1_proxy`. The endpoint reference, `acc_1_ref`, returned from `get_account()` is used to initialize an account proxy object, `acc_1`.
3. Likewise, obtain a proxy to an account in bank server B, `acc_2`.
4. You should always enclose a transaction in a `try` block, because it might be necessary to catch an exception and roll back the transaction.
5. The `IT_Bus::TransactionSystem::begin_transaction()` call initiates the transaction.

6. The `IT_Bus::TransactionSystem::commit_transaction()` call attempts to commit the changes made to server A and server B. The boolean argument is the `report_heuristics` flag, which can take the following values:
    - `true`—specifies that heuristic decisions should be reported during the commit protocol (if supported by the underlying transaction system).
    - false—specifies that heuristic decisions should not be reported.
7. It is essential to catch and handle any exceptions that might be thrown during a transaction.
8. The `within_transaction()` call is needed at this point, because the `rollback_transaction()` function must only be called from within a transaction. If `rollback_transaction()` is called outside a transaction, it raises an exception.
9. If an exception is thrown, the transaction must be aborted by calling `IT_Bus::TransactionSystem::rollback_transaction()`.

Server Example

Overview

This section describes a transactional Artix server that implements a bank service and an unlimited number of account services (each account service representing a single account). The server uses a transactional resource—an Oracle database—to store the account records. This transactional resource is integrated with the Artix transaction manager using an XA interface (which is an X/Open standard, supported both by Artix and by Oracle).

C++ demonstration code

The bank server demonstration code is located in the following directory:

```
ArtixInstallDir/cxx_java/samples/transactions/common/src
    /servers/cxx_xa_http_soap_wsat
```

Servant classes

The bank server implements two servant classes, as follows:

- BankImpl class.
- AccountImpl class.

BankImpl class

The `BankImpl` servant class implements the operations from the `Bank` port type. The `BankImpl` class has the characteristics of a typical account factory class: that is, it provides operations for creating, finding and deleting account objects. Clients that use the bank server would initially connect to the `BankService` service and then call the `Bank` operations to obtain a reference to an account object.

Because the `BankImpl` class does not participate in any transaction (that is, it does not access any transactional resources), it is of no relevance to transactional programming and is not discussed here in detail.

AccountImpl class

The `AccountImpl` servant class implements the operations from the `Account` port type. The `AccountImpl` class is responsible for accessing and updating account details stored in an Oracle database. Because the Oracle XA switch is registered with the Artix transaction manager, any database updates must be coordinated by the Artix transaction manager. When writing the

AccountImpl class, therefore, you should be aware that its operations are participating in a global transaction and that this affects the way you access the database.

Integration with Oracle database

In the bank server demonstration, the Oracle database is treated as a resource whose transactions are to be coordinated by the Artix transaction manager. In order to integrate the Oracle database with the Artix transaction manager, you must do the following:

1. *Register the Oracle XA switch*—to subordinate Oracle transactions to the Artix transaction manager, register an Oracle XA switch object with the Artix transaction manager. See "Registering an XA Resource" on page 75 for a detailed discussion.
2. *Modify code that interacts with the database*—when the XA interface is enabled, you must observe the following programming restrictions:
   - *Do not open or close any database connections*—connections are now managed automatically through the XA interface.
   - *Do not use embedded SQL or native database API to demarcate transactions*—for example, you must not call the embedded SQL commands, `EXEC SQL BEGIN`, `EXEC SQL COMMIT`, or `EXEC SQL ROLLBACK`.
3. *Link the server with the relevant Oracle libraries.*

C++ registering the Oracle XA switch

Example 4 shows how to register an Oracle XA switch with the Artix transaction manager. Registration must occur before the server processes any incoming requests. You would normally register the XA switch during initialization of the server program.

Example 4: *C++ Registering an Oracle XA Switch*

```
   // C++
1  #include <sqlca.h>

2  extern "C" IT_DECLSPEC_IMPORT xa_switch_t xaosw;
   extern "C" IT_DECLSPEC_IMPORT xa_switch_t xaoswd;
   ...
3  xa_switch_t* database_switch = &xaosw;

   IT_Bus::TransactionManager & tx_mgr =
```

Example 4: *C++ Registering an Oracle XA Switch*

```
        bus->transactions().get_transaction_manager(
            IT_Bus::TransactionSystem::XA_TRANSACTION_TYPE
        );

4   IT_Bus::XATransactionManager& xa_tx_mgr =
        dynamic_cast<IT_Bus::XATransactionManager&>(tx_mgr);

    IT_Bus::String db_resource_id("oracle_bank");
    db_resource_id += bank_id;

5   bool succeeded = xa_tx_mgr.register_xa_resource(
        database_switch,
        IT_Bus::String::EMPTY,      // open_string  - ""
        IT_Bus::String::EMPTY,      // close_string - ""
        db_resource_id, // configuration prefix
        false,         // don't use dynamic_registration_optimization
        false          // not single-threaded
    );

    if (!succeeded)
    {
        throw IT_Bus::Exception(
            "Failed to register Oracle database as an XA resource"
        );
    }
```

The preceding code fragment can be explained as follows:

1. The `sqlca.h` header file is an Oracle header file that defines two instances of `xa_switch_t` type: `xaosw`, for a normal XA switch, and `xaoswd`, for a dynamically registering XA switch.
2. Declare `xaosw` to be an external C type (the `xa_switch_t` type is declared in C, not C++).
3. The XA switch used in this example, `database_switch`, is simply a pointer to an ordinary Oracle XA switch object, `xaosw`.
4. The XA transaction manager, `xa_tx_mgr`, is an object that is used to integrate XA resources with the Artix transaction manager.

5. Call `register_xa_resource()` on the `IT_Bus::XATransactionManager` instance to register the Oracle XA switch, `xaosw`, with the Artix XA transaction manager.

   In this example, the open string and the close string are read from an Artix configuration file. This is flagged by passing an empty string, `""`, as the open string. The identifier, `db_resource_id`, is then used as a prefix string to identify the relevant variables in the configuration file. See "Configuration" on page 48 for details.

C++ AccountImpl class

Example 5 shows the implementation of the `AccountImpl` servant class. The operations implemented by this class are all intended to execute in the context of a global transaction. This has an effect on the way you program the database access: in particular, you must avoid starting a local transaction.

Example 5: *C++ AccountImpl Servant Class*

```
   // C++
   ...
   void
1  AccountImpl::get_balance(
       IT_Bus::Double &_return
   ) IT_THROW_DECL((IT_Bus::Exception))
   {
2      IT_Bus::String id = get_instance_id();
       const char * id_str = id.c_str();
       double return_balance = 0;

3      ::get_balance_from_db(id_str, return_balance);

       _return = return_balance;
   }

   void
4  AccountImpl::make_deposit(
       const IT_Bus::Double amount
   ) IT_THROW_DECL((IT_Bus::Exception))
   {
       IT_Bus::String id = get_instance_id();
       const char * id_str = id.c_str();

       IT_Bus::Double balance;
       get_balance(balance);
```

Example 5: *C++ AccountImpl Servant Class*

```
    balance += amount;

    ::set_balance_in_db(id_str, balance);

    cout << "Made deposit of $" << amount << " to account \'" <<
   id << endl;
}

void
AccountImpl::make_withdrawl(
    const IT_Bus::Double amount
) IT_THROW_DECL((IT_Bus::Exception))
{
    IT_Bus::String id = get_instance_id();
    const char * id_str = id.c_str();

    IT_Bus::Double balance;
    get_balance(balance);

    if (balance < amount)
    {
        throw IT_Bus::Exception("Not enough funds to faciliate
   withdrawl");
    }

    balance -= amount;

    ::set_balance_in_db(id_str, balance);

    cout << "Made withdrawl of $" << amount << " from account \'"
   << id << endl;
}

AccountIDsType
AccountImpl::list_all()
{
    AccountIDsType account_ids;
    account_ids = ::list_all_accounts();
    return account_ids;
}
```

The preceding class implementation can be explained as follows:

1. The `get_balance()` function provides the implementation of the account service's `get_balance` WSDL operation.
2. The `get_instance_id()` function returns the identity of the account that is being accessed. The implementation of the `get_instance_id()` function depends on the approach used to implement the account servant class, as follows:
   - *Transient servant*—in this approach, a distinct servant object is created for each account instance. The account identity would be passed to the servant object at creation time and stored in a member variable. The `get_instance_id()` function simply returns the stored identity in this case.
   - *Default servant*—in this approach, a *single* servant object services requests for all account instances. The account identity, therefore, cannot be stored in a member variable. The `get_instance_id()` function obtains the account identity by querying the current *address context* in this case. For details of how this works, see the discussion of default servants in *Developing Artix Application in C++*.
3. The `get_balance_from_db()` function uses embedded SQL calls to retrieve the account balance from an Oracle database. This database access is integrated into the global transaction.

   See Example 6 for a detailed description of this function.
4. The following `make_deposit()`, `make_withdrawl()` and `list_all()` functions are implementations of WSDL operations, which follow a pattern similar to the `get_balance()` function.

C++ database code

Example 6 shows some of the functions that the bank server uses to access the Oracle database (taken from the `oracle_db_fns.pc` file). This file contains embedded SQL statements, which will ultimately be converted into C++ by the Oracle pre-compiler.

Example 6: *C++ Database Code for Accessing Account Data*

```
  // For Pro/C++ compiler (C++ with embedded SQL)

  void
1 get_balance_from_db(
      const char *    the_account_id,
      double&         return_balance
  )
  {
      // local Oracle variables

      EXEC SQL BEGIN DECLARE SECTION;
          VARCHAR acc_id[20];
          double  balance=0.0;
      EXEC SQL END DECLARE SECTION;

      acc_id.len = strlen(the_account_id);
      strncpy((char*)&acc_id.arr[0], the_account_id, 19);
      return_balance = (double)0.0;

      // get the balance from the database table
      bool foundit=false;
      EXEC SQL WHENEVER NOT FOUND DO break;
      for (;;)
      {
          EXEC SQL SELECT CURRENT_BALANCE
              INTO :balance
              FROM ARTIX_ACCOUNTS
              WHERE ACCOUNT_ID = :acc_id;

          foundit = true;
          break;
      }
      if (foundit)
      {
          return_balance = balance;
      }
  }
```

Example 6: *C++ Database Code for Accessing Account Data*

```
  void
2 set_balance_in_db(
      const char *    the_account_id,
      double          new_balance
  )
  {
      EXEC SQL BEGIN DECLARE SECTION;
          VARCHAR acc_id[20];
          double  balance;
      EXEC SQL END DECLARE SECTION;

      acc_id.len = strlen(the_account_id);
      strncpy((char*)&acc_id.arr[0], the_account_id, 19);
      balance = new_balance;

      bool foundit=false;
      EXEC SQL WHENEVER NOT FOUND DO break;
      for (;;)
      {
          EXEC SQL UPDATE ARTIX_ACCOUNTS
              SET CURRENT_BALANCE = :balance
              WHERE ACCOUNT_ID = :acc_id;

          foundit=true;
          break;
      }
  }
```

The preceding database code can be explained as follows:

1. The `get_balance_from_db()` function uses conventional embedded SQL calls to access the `ARTIX_ACCOUNTS` table, selecting the `CURRENT_BALANCE` field from the row indexed by `ACCOUNT_ID`.

   From a transaction viewpoint, it is worth noting that transaction demarcation statements (`EXEC SQL BEGIN`, `EXEC SQL COMMIT`, or `EXEC SQL ROLLBACK`) do *not* appear anywhere in this function. When an XA switch is registered, the Artix transaction manager is responsible for transaction demarcation.

2. The `set_balance_in_db()` function uses conventional embedded SQL calls to update the `ARTIX_ACCOUNTS` table, setting the `CURRENT_BALANCE` field in the row indexed by `ACCOUNT_ID`.

Once again, note the absence of any transaction demarcation statements (`EXEC SQL BEGIN`, `EXEC SQL COMMIT`, or `EXEC SQL ROLLBACK`).

Configuration

Overview

To use Artix transactions, it is necessary to load and configure the relevant transaction system (Artix supports multiple transaction systems). Artix does *not* load a transaction system by default. Hence, you must include transaction plug-ins explicitly in the `orb_plugins` list.

For a more detailed discussion of transaction configuration, see "Selecting a Transaction System" on page 51.

Configuration file location

The `tx_demo.cfg` configuration file is located in the following directory:

ArtixInstallDir`/cxx_java/samples/transactions/common/etc`

Client configuration

Example 7 shows the configuration settings for the bank client, which uses the `artix.demos.tx_demo.wsat_coordinated` Bus ID (which can be specified, for example, by the `-BUSname` command-line switch). In this example, the client is configured to use the WS-AT transaction manager.

Example 7: *Client Configuration Using the WS-AT Transaction Manager*

```
# Artix Configuration File

# Global configuration settings
...

# Transaction demonstrations settings
artix
{
    demos
    {
        tx_demo
        {
            ...
            wsat_coordinated
            {
                orb_plugins = ["local_log_stream", "ws_coordination_service"];
                plugins:bus:default_tx_provider:plugin="wsat_tx_provider";
            };
        };
    };
```

Example 7: *Client Configuration Using the WS-AT Transaction Manager*

```
};
```

The following configuration settings are relevant to transactions in the client:

- `orb_plugins`—the client is configured to load the `ws_coordination_service` plug-in, which implements a transaction manager on the pattern of the WS-Coordination standard. Implicitly, the client also loads the `wsat_protocol` plug-in, which provides the capability to send WS-AtomicTransaction transaction contexts over SOAP.
- `plugins:bus:default_tx_provider:plugin`—because Artix can support several different transaction systems (for example, WS-AT and OTS Encina), you need to specify explicitly which transaction system the client uses when it initiates a transaction. In this example, the client is configured to use the WS-AT transaction system by default.

Server configuration

Example 8: *Server Configuration with Oracle XA Resource*

```
# Artix Configuration File

# Global configuration settings
...

# Transaction demonstrations settings
artix
{
    demos
    {
        tx_demo
        {
            ...
            wsat_server
            {
                orb_plugins = ["local_log_stream", "wsat_protocol", "coordinator_stub_wsdl"];
                plugins:bus:default_tx_provider:plugin="wsat_tx_provider";

                oracle_xa
```

Example 8: *Server Configuration with Oracle XA Resource*

```
                {
                    policies:http:trace_requests:enabled="true";

                    # Configuration settings for the Oracle Databases
                    #
                    oracle_bankA:open_string="Oracle_XA+Acc=P/scott/tiger+SesTm=60+threads=true";
                    oracle_bankA:close_string="";
                    poa:oracle_bankA:direct_persistent="true";
                    poa:oracle_bankA:well_known_address:host="0.0.0.0"; # all network adapters
                    poa:oracle_bankA:well_known_address:port="13003";   # unique port

                    oracle_bankB:open_string="Oracle_XA+Acc=P/scott/tiger+SesTm=60+threads=true";
                    oracle_bankB:close_string="";
                    poa:oracle_bankB:direct_persistent="true";
                    poa:oracle_bankB:well_known_address:host="0.0.0.0"; # all network adapters
                    poa:oracle_bankB:well_known_address:port="13004";   # unique port
                };
            };
        };
    };
};
```

The following configuration settings are relevant to transactions in the server:

- `orb_plugins`—the server is configured to load the `wsat_protocol` plug-in, which provides the capability to send WS-AtomicTransaction transaction contexts over SOAP, and the `coordinator_stub_wsdl` plug-in, which enables the server to call back on the transaction coordinator object in the client.
- `oracle_bankA:open_string`—if the programmer passes a blank open string when registering an XA switch, Artix reads the open string from configuration instead. The prefix, `oracle_bankA`, is set by the programmer at registration time (see "C++ registering the Oracle XA switch" on page 40).
- `oracle_bankA:close_string`—if the programmer passes a blank open string when registering an XA switch, Artix reads the close string from configuration instead. In this example, the close string is a blank, because Oracle does not use the close string.

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

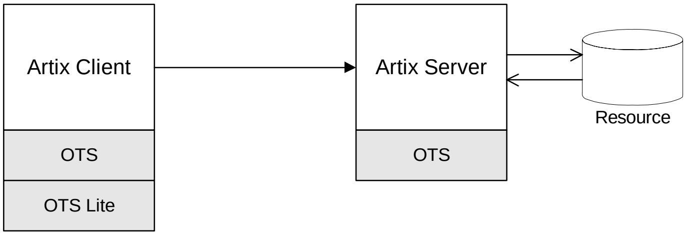
This chapter discusses the following topics:

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 6 shows a client-server deployment that uses the OTS Lite plug-in.

Figure 6: *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 96 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:

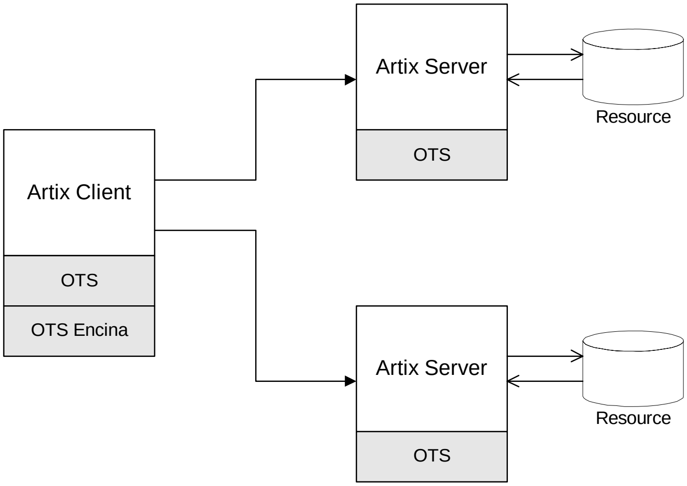
```
# Artix Configuration File

# 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 7 shows a client/server deployment that uses the OTS Encina plug-in.

Figure 7: *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 9 shows a complete configuration for using the OTS Encina transaction manager:

Example 9: *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).

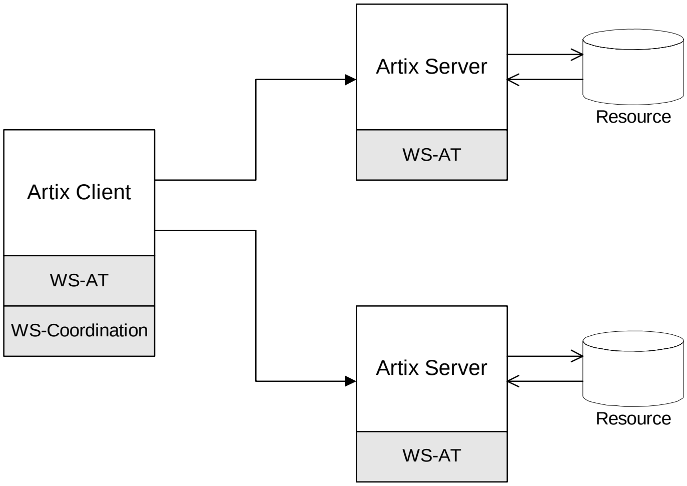
8. The settings in the next few lines specify the basic configuration of the OTS Encina plug-in. It should not be necessary ever to change the values of these configuration settings.

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 8 shows a client/server deployment that uses the lightweight WS-AT transaction system.

Figure 8: *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 10 shows a complete configuration for using the non-recoverable WS-AT transaction manager:

Example 10: *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 10: *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:

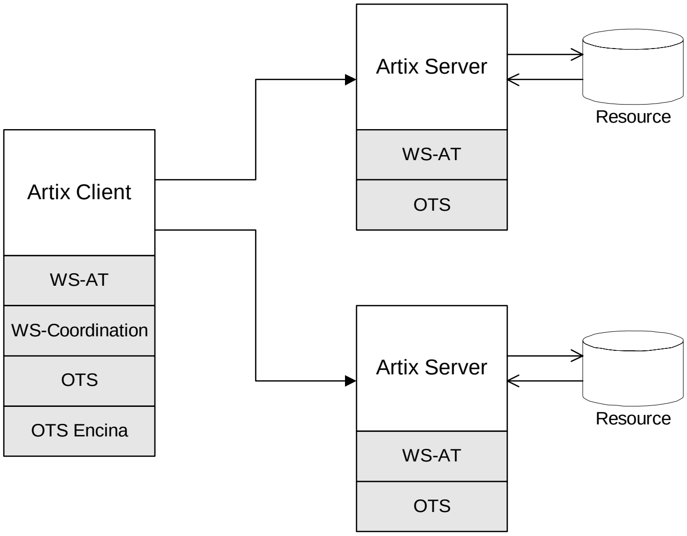
- WS-AtomicTransactions (http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-AtomicTransaction.pdf).
- WS-Coordination (http://msdn.microsoft.com/library/en-us/dnglobspec/html/WS-Coordination.pdf).

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 9 shows a client/server deployment that uses the recoverable WS-AT transaction system.

Figure 9: *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`.

Enabling recovery

Since Artix version 4.0, the WS-AT transaction system is recoverable by default. Hence, to use the recoverable version of WS-AT in your application, you can either omit the `plugins:ws_coordination_service:disable_tx_recovery` variable from your Artix configuration file or set it to false, as follows:

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

Loading WS-AT and OTS Encina plug-ins

The configuration for the recoverable WS-AT transaction system is essentially a combination of the WS-AT configuration and the OTS Encina configuration. It is only necessary to load the WS-AT plug-ins explicitly—if recovery is enabled, Artix implicitly loads the OTS and OTS Encina plug-ins.

Sample configuration

Example 10 shows a complete configuration for using the recoverable WS-AT transaction manager:

Example 11: *Sample Configuration for 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       # OTS Encina Configuration
        initial_references:TransactionFactory:plugin =
     "ots_encina";
        plugins:ots_encina:direct_persistence = "true";
        plugins:ots_encina:iiop:port = "3213";
        plugins:ots_encina:initial_disk = "../../log/encina.log";
        plugins:ots_encina:initial_disk_size = "1";
        plugins:ots_encina:restart_file =
     "../../log/encina_restart";
        plugins:ots_encina:backup_restart_file =
     "../../log/encina_restart.bak";

        # Boilerplate configuration settings for OTS Encina:
        # (you should never need to change these)
        plugins:ots_encina:shlib_name = "it_ots_encina";
```

Example 11: *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 57.

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.

CHAPTER 4

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:

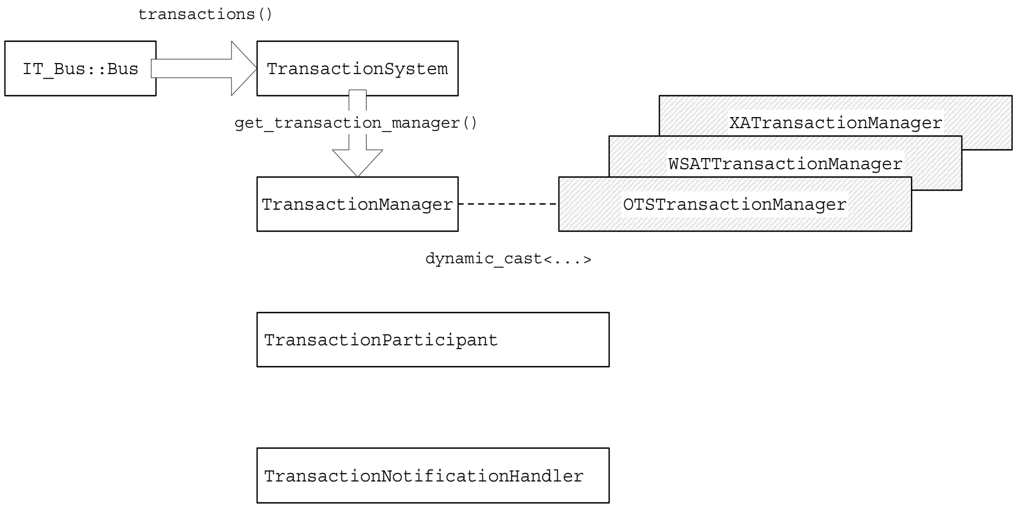
Artix Transaction Interfaces

Overview

Figure 10 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 your code is restricted to using the generic classes, you will be able to switch between any of the transaction systems supported by Artix.

On the server side it is likely that you will need to access advanced functionality, which is available only from technology-specific transaction manager classes, such as `OTSTransactionManager`, `WSATTransactionManager`, or `XATransactionManager`.

Figure 10: *Overview of the Artix Transaction API*



Accessing the transaction system

To access the Artix transaction system, call the transactions() function on the Bus. The returned IT_Bus::TransactionSystem reference provides the starting point for accessing all aspects of Artix transactions.

The IT_Bus::Bus::transactions() function has the following signature:

```
IT_Bus::TransactionSystem&
transactions() IT_THROW_DECL((IT_Bus::Exception));
```

TransactionSystem class

The IT_Bus::TransactionSystem class provides the basic functions needed for transaction demarcation on the client side (begin_transaction(), commit_transaction() and rollback_transaction()). For more details see "Beginning and Ending Transactions" on page 71.

To access server-side functions and advanced client-side functions, you must call IT_Bus::TransactionSystem::get_transaction_manager() to obtain an IT_Bus::TransactionManager instance.

TransactionManager class

The IT_Bus::TransactionManager class provides server-side functions and advanced transaction functionality. For the server side, the most important member function is IT_Bus::TransactionManager::enlist(), which enables you to implement a transactional resource by enlisting a transaction participant object.

In order to support multiple transaction systems, the TransactionManager class is designed as a facade, which is layered above a specific implementation. In some cases, if the functionality provided by the generic TransactionManager is not sufficient, you might need to downcast the TransactionManager reference to one of the following types:

- OTSTransactionManager class.
- WSATTransactionManager class.

OTSTransactionManager class

The IT_Bus::OTSTransactionManager class provides access to an underlying CORBA OTS implementation of the transaction system. Using this class, you can access the CosTransactions::Coordinator and the CosTransactions::Current objects for this transaction.

A discussion of the CORBA OTS is beyond the scope of this guide. For more details, see the *CORBA OTS Guide* (http://www.iona.com/support/docs/orbix/6.2/develop.xml), which is available from the Orbix documentation suite.

WSATTransactionManager class

The IT_Bus::WSATTransactionManager class provides access to an underlying WS-AT implementation of the transaction system. Currently, the WSATTransactionManager class provides access to the WS-AT context, which is included in a SOAP header with every transactional operation call.

TransactionParticipant base class

If you want to implement a transactional resource on the server side, you can define and implement a class that inherits from the IT_Bus::TransactionParticipant base class. The TransactionParticipant class 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 121.

There are alternative ways of implementing a transactional resource, which do not require you to implement a TransactionParticipant class. Some transaction managers (for example, OTSTransactionManager) support alternative approaches.

TransactionNotificationHandler base class

If you want to synchronize certain actions with the committing or rolling back of a transaction, you can define and implement a class that inherits from the IT_Bus::TransactionNotificationHandler base class. The IT_Bus::TransactionNotificationHandler class receives 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* functions. 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 functions are typically the only functions that you need on the client side.

TransactionSystem member functions

Example 12 shows the public member functions of the `IT_Bus::TransactionSystem` class.

Example 12: *The IT_Bus::TransactionSystem Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionSystem
      : public virtual RefCountedBase
    {
      public:
        virtual ~TransactionSystem();

        virtual void
        begin_transaction() IT_THROW_DECL((Exception)) = 0;

        virtual Boolean
        commit_transaction(
            Boolean report_heuristics
        ) IT_THROW_DECL((Exception)) = 0;

        virtual void
        rollback_transaction() IT_THROW_DECL((Exception)) = 0;

        virtual TransactionManager&
        get_transaction_manager(
            const String&
   tx_manager_type=DEFAULT_TRANSACTION_TYPE
        ) IT_THROW_DECL((Exception)) = 0;

        virtual Boolean
```

Example 12: *The IT_Bus::TransactionSystem Class*

```
        within_transaction() = 0;
        ...
        // String constants for transaction manager types
        static const String        DEFAULT_TRANSACTION_TYPE;
        static const String        WSAT_TRANSACTION_TYPE;
        static const String        OTS_TRANSACTION_TYPE;
        static const String        XA_TRANSACTION_TYPE;
        ...
    };

    typedef Var<TransactionSystem> TransactionSystem_var;
    typedef TransactionSystem* TransactionSystem_ptr;
};
```

Client transaction functions

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

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

  This function can throw the following exceptions:

  - `TransactionAlreadyActiveException` is thrown if `begin_transaction()` is called inside an already active transaction.
  - `TransactionSystemUnavailableException` is thrown if the transaction system cannot be loaded. This usually points to a configuration problem.

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

  This function can throw the following exception:

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

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

  This function 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:

- within_transaction()—returns true if the current thread is associated with a transaction; otherwise, false.
- get_transaction_manager()—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 121). For advanced applications, you can also downcast the TransactionManager reference to get a particular implementation of the transaction system (for example, an IT_Bus::OTSTransactionManager object or an IT_Bus::WSATTransactionManager object).

  This function can throw the following exception:

  - ♦ TransactionSystemUnavailableException is thrown if the transaction system cannot be loaded.

Server Programming

Overview

On the server side, the main transactions-related programming task is the integration of resources with the Artix transaction system. The purpose of this integration step is to enable the Artix transaction manager to control the resource's transactions.

By far the simplest and most common method of integrating resources into the Artix transaction system is to use the *XA standard*, which is supported by most modern databases. An XA-compliant resource provides a special data structure, the *XA switch*, which you can then register with Artix in order to integrate the resource with the Artix transaction system.

In this section

This section contains the following subsections:

Registering an XA Resource

Overview

The simplest way to integrate a third-party resource (such as a database) into the Artix transaction system is to use the XA interface. If the third-party resource supports the XA interface, all that you need to do to integrate the resource with the Artix transaction system is to register a particular type of object, an XA switch, with the Artix transaction manager. This puts the Artix transaction manager in charge of beginning, committing and rolling back transactions associated with the XA resource. This also implies that the resource can now participate in distributed transactions, since these are supported by the Artix transaction manager.

When to register an XA resource

You should register an XA resource in the `main()` function as your application program is performing initialization and *before* you attempt to access the resource for the first time.

register_xa_resource() function

The `register_xa_resource()` function, which is a member of the `IT_Bus::XATransactionManager` class, is used to register third-party XA resource managers with the Artix transaction manager. Example 13 gives the signature of the `register_xa_resource()` function.

Example 13: *The register_xa_resource() Function*

```
// C++
// In IT_Bus::XATransactionManager
IT_Bus::Boolean
register_xa_resource(
    xa_switch_t*    xa_switch,
    IT_Bus::String  open_string,
    IT_Bus::String  close_string,
    IT_Bus::String  resource_manager_identifier,
    IT_Bus::Boolean use_dynamic_registration_optimization,
    IT_Bus::Boolean is_single_threaded_resource
)=0;
```

register_xa_resource() arguments

The IT_Bus::XATransactionManager::register_xa_resource() function takes the following arguments:

- xa_switch,
- open_string,
- close_string,
- resource_manager_identifier,
- use_dynamic_registration_optimization,
- is_single_threaded_resource.

xa_switch

The xa_switch argument is a pointer to an xa_switch_t instance, which is provided by the third-party XA resource manager. The xa_switch_t type is declared in the <orbix_sys/xa.h> header, which you need to include in any file that references the xa_switch_t type.

Each XA resource manager defines a specific XA switch instance, which is essentially a global struct variable. Table 1 gives the identifier names for some common XA resource managers.

Table 1: *Sample Mechanisms for Obtaining XA Switches*

XA Resource Manager	XA Switch Instance
Oracle DB	Two XA switches are defined as global instances in the Oracle sqlca.h header file: • xaosw—normal Oracle XA switch. • xaoswd—Oracle XA switch that supports dynamic registration.
Sybase DB	sybase_xa_switch
DB2	db2xa_switch (UNIX), or *db2xa_switch (Windows)

open_string

The open_string argument specifies the string that the Artix XA transaction manager passes to xa_open() when it opens a connection to the XA resource manager. The form of the open string is *not* defined by Artix; it is defined by the particular third-party XA resource manager being registered.

The XA standard intends that the open string be used as a general mechanism for passing initialization parameters to the XA resource manager.

Examples of open strings for some common XA resource managers are provided in Table 2.

Table 2: *Examples of Open Strings for Some XA Resource Managers*

XA Resource Manager	Example Open String
Oracle DB	`Oracle_XA+Acc=P/SCOTT/TIGER+SesTm=60+threads=true`
Sybase DB	`-U<Username> -P<Password> -N<DB_Name> -T<LoggingType> -L<LogFile>`
DB2	`<DB_Name>,<Username>,<Password>`

Note: An empty open string, `""`, is treated as a special case. In this case, Artix assumes that the open string is specified in the Artix configuration file. The name of the configuration variable that specifies the open string is determined by the `resource_manager_identifier` argument.

close_string

The `close_string` argument specifies the string that the Artix XA transaction manager passes to `xa_close()` when it closes a connection to the XA resource manager.

Examples of close strings for some common XA resource managers are provided in Table 3. Some XA resource managers (for example, Oracle DB) ignore the close string, in which case you can pass an empty string, `""`.

Table 3: *Examples of Close Strings for Some XA Resource Managers*

XA Resource Manager	Example Close String
Oracle DB	*None*
Sybase DB	*None*
DB2	*None*

resource_manager_identifier

The resource_manager_identifier argument specifies a string that serves as a name prefix for certain configuration variables in the Artix configuration file. These configuration variables can then be used to configure the resource manager registration.

In particular, if you pass an empty string, "", as the open_string argument, Artix assumes that you want to specify the value of the open string in configuration instead of passing it as an argument. In this case, Artix looks for a configuration variable called *ResourceManagerPrefix*:open_string, where *ResourceManagerPrefix* is the string passed as the resource_manager_identifier argument.

For example, if you specify the open_string argument to be an empty string, "", and the resource_manager_identifier argument to be xa_resource_managers:oracle, you can then specify the open string in the Artix configuration file as follows:

```
# Artix Configuration File
oracle_xa_example {
    xa_resource_managers:oracle:open_string =
        "Oracle_XA+Acc=P/SCOTT/TIGER+SesTm=60";
    xa_resource_managers:oracle:close_string="";

    poa:xa_resource_managers:oracle:direct_persistent="true";
    poa:xa_resource_managers:oracle:well_known_address:host
        ="0.0.0.0"; # all network adapters
    poa:xa_resource_managers:oracle:well_known_address:port
        ="13003";   # unique port
    ...
};
```

Where the Artix Bus has been initialized with the configuration scope, oracle_xa_example.

use_dynamic_registration_optimization

The use_dynamic_registration_optimization argument is a boolean flag that informs the Artix XA transaction manager whether or not the resource manager has enabled the dynamic registration optimization. Consult the documentation for your third-party XA resource manager to discover whether or not this optimization is supported. If the optimization is supported, you can enable it as follows:

1. Follow the instructions in the third-party XA resource manager documentation to enable the dynamic registration optimization.
2. Pass the value, `true`, to the `use_dynamic_registration_optimization` argument.

It is important to ensure that both the transaction manager and the resource manager are aware of the dynamic registration optimization, because this optimization changes the nature of their interaction through the XA interface. For more details, see "Dynamic Registration Optimization" on page 81.

is_single_threaded_resource

The `is_single_threaded_resource` argument is a boolean flag that selects the XA threading model in the transaction manager as follows:

- `false`—the XA threading model is multi-threaded (each thread maps to a resource connection),
- `true`—the XA threading model is single-threaded (a process maps to a single resource connection).

You must also ensure that the third-party XA resource manager is configured to use the *same* threading model as the transaction manager.

For example, if you want to use the multi-threaded model with the Oracle XA switch, you must include the setting, `threads=true`, in the Oracle XA open string.

For more details see "Threading and XA Resources" on page 105.

Example

Example 14 shows an example of how to register an Oracle XA switch with the Artix XA transaction manager.

Example 14: *Example of Registering an Oracle XA Switch*

```
   // C++
   #include <it_bus/bus.h>
   #include <it_bus/transaction_system.h>
   #include <it_bus_pdk/xa_transaction_manager.h>
1  #include <orbix_sys/xa.h>

2  #include <sqlca.h>
```

Example 14: *Example of Registering an Oracle XA Switch*

```
3  extern "C" IT_DECLSPEC_IMPORT xa_switch_t xaosw;

   IT_Bus::Bus_var bus = ...
   ...
4  IT_Bus::XATransactionManager& xa_tx_mgr = dynamic_cast
   <IT_Bus::XATransactionManager&>(
       bus->transactions().get_transaction_manager(
           IT_Bus::TransactionSystem::XA_TRANSACTION_TYPE
       )
   );

5  xa_tx_mgr->register_xa_resource(
       &xaosw,                        // Oracle XA switch
       "Oracle_XA+Acc=P/SCOTT/TIGER+SesTm=60+threads=true",
                                      // Oracle open string
       "",                            // Oracle close string
       "",                            // resource manager identifier
       false,                         // dynamic registration?
       true                           // multi-threaded?
   );
```

The preceding code fragment can be explained as follows:

1. The Artix `orbix_sys/xa.h` header file contains the standard declaration of the `xa_switch_t` struct type, as defined in the *The XA Specification*. Include this header in any file that refers to the `xa_switch_t` type.
2. The `sqlca.h` header file is an Oracle header file that defines two instances of `xa_switch_t` type: `xaosw`, for a normal XA switch, and `xaoswd`, for a dynamically registering XA switch.
3. Declare `xaosw` to be an external C type (the `xa_switch_t` type is declared in C, not C++).
4. From the Bus instance, obtain an `IT_Bus::XATransactionManager` instance.
5. Call `register_xa_resource()` on the `XATransactionManager` instance to register the Oracle XA switch, `xaosw`, with the Artix XA transaction manager. In this example, the open string is provided explicitly in the second parameter; the resource manager identifier is not used (empty string); the dynamic registration optimization is not used; and the threading model is multi-threaded.

Dynamic Registration Optimization

Overview

The dynamic registration optimization is a variation of the usual protocol that governs interactions between an XA transaction manager and an XA resource manager. Typically, it results in more efficient access to the resource. For example, if the resource is a database, this optimization causes the database tables to be locked less often, thereby improving concurrency. Hence, it is usually a good idea to enable this optimization.

If you just want to know how to enable this feature, skip ahead to "Enabling dynamic registration" on page 85 for details. For advanced users, this subsection also provides background information on the dynamic registration optimization, so that you can understand how this protocol works. A key difference between dynamic registration and normal registration is that dynamic registration exploits the AX interface.

AX interface

Example 15 shows the signatures of the two functions, `ax_reg()` and `ax_unreg()`, that constitute the AX interface. These functions enable an XA resource manager to call *back* on an XA transaction manager (that is, reversing the usual direction of control, where the transaction manager calls the resource manager).

Example 15: *Functions in the AX Interface*

```
/* C */
int ax_reg(int rmid, XID *xid, long flags)

int ax_unreg(int rmid, long flags)
```

The AX functions can be explained as follows:

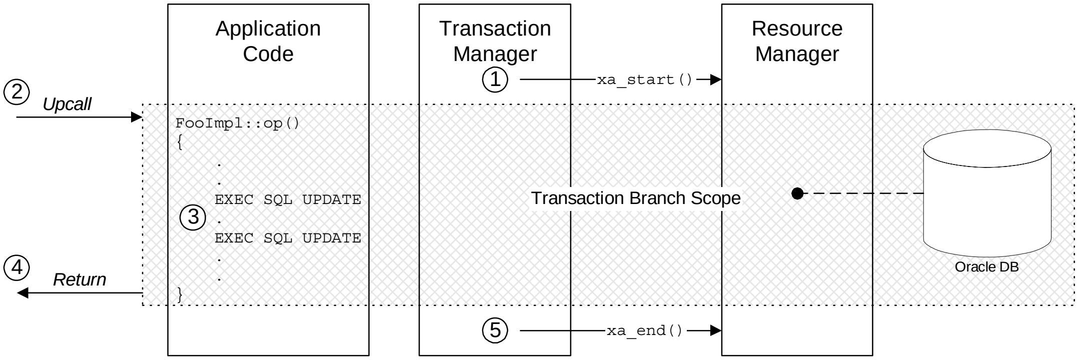
- `ax_reg()` function—is called by the resource manager to inform the transaction manager that work is about to begin on a transaction in the current thread. For example, in the case of a database, the `ax_reg()` call would be triggered, when the application code attempts to perform a database update.
- `ax_unreg()` function—is needed only for the special case where an application makes some database updates *outside* the context of a global transaction. The resource manager then calls `ax_unreg()` to

inform the transaction manager that the work has ended and, therefore, the current thread is free once more to participate in a global transaction.

Normal registration

Figure 11 shows the outline of an Artix transactional server that has a *normally* registered resource manager, where `FooImpl::op()` is the implementation of the WSDL operation, `op()`.

Figure 11: *Invocation Dispatch for a Normally Registered RM*



The server is divided up into the following parts:

- The *Application Code*—showing the implementation of the WSDL operation, `op()`, and
- The *Transaction Manager*—showing the calls made by the Artix transaction manager,
- The *Resource Manager*—showing a database resource and its associated XA resource manager.

The shaded area shows the scope of the association between the current thread and a transaction branch in the resource manager. The association begins with `xa_start()` and ends with `xa_end()`.

Steps in normal registration

In this scenario, the Artix server accesses an XA resource which is registered normally. When the server receives a client request with transactional context, the invocation dispatch proceeds as follows:

1. Before dispatching the invocation, the Artix transaction manager (TM) obtains a list of all the registered XA resource managers (RMs). In this case, there is only one RM, which is registered normally. The TM calls `xa_start()` on the RM, thereby creating an association between the current thread and a transaction branch in the RM.

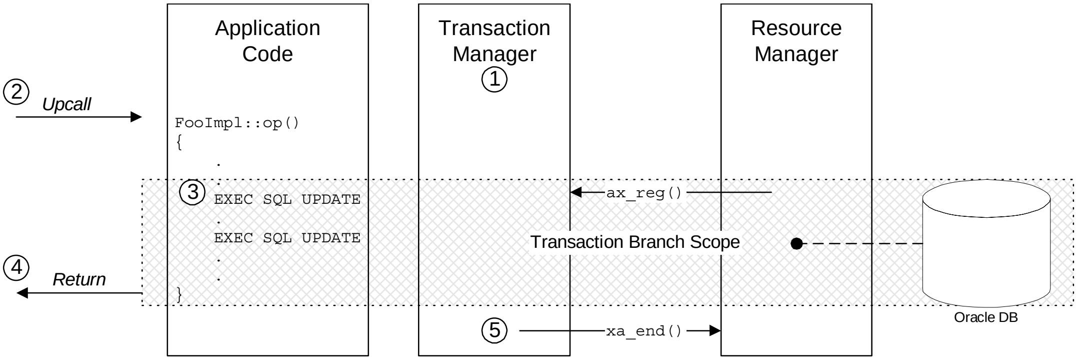
   > **Note:** The `xa_start()` call typically imposes some overheads on the resource. For example, a mutex lock might be set on the database connection.

2. The Artix runtime makes an upcall to the `FooImpl::op()` function, which implements the WSDL operation, `op()`.
3. In the body of the `op()` function, the application code makes updates to the resource—for example, through some embedded SQL calls such as `EXEC SQL UPDATE`. These updates are governed by the current transaction.
4. The `FooImpl::op()` upcall returns.
5. The Artix TM calls `xa_end()` on the RM, thereby ending the association between the current thread and the transaction branch in the RM.

Dynamic registration

Figure 12 shows the outline of an Artix transactional server that has a *dynamically* registered resource manager, where `FooImpl::op()` is the implementation of the WSDL operation, `op()`.

Figure 12: *Invocation Dispatch for a Dynamically Registered RM*



The shaded area shows the scope of the association between the current thread and a transaction branch in the resource manager. The association begins when the RM calls `ax_reg()` and ends when the TM calls `xa_end()`.

Steps in dynamic registration

In this scenario, the Artix server accesses an XA resource which is registered dynamically. When the server receives a client request with transactional context, the invocation dispatch proceeds as follows:

1. Before dispatching the invocation, the Artix TM obtains a list of all the registered XA RMs. In this case, there is one dynamically registered RM. The TM does *not* call `xa_start()` on the dynamically registered RM.
2. The Artix runtime makes an upcall to the `FooImpl::op()` function, which implements the WSDL operation, `op()`.
3. In the body of the `op()` function, the application code makes updates to the resource—for example, through some embedded SQL calls such as `EXEC SQL UPDATE`. The very first update triggers the RM to make an `ax_reg()` callback on the TM. This callback initiates an association between the current thread and a transaction branch in the RM.

4. The `FooImpl::op()` upcall returns.
5. The Artix TM calls `xa_end()` on the dynamically registered RM, thereby ending the association between the current thread and the transaction branch in the RM.

Enabling dynamic registration

To enable dynamic registration for a particular XA resource, perform the following steps:

1. Follow the instructions in the third-party XA resource manager documentation to enable the dynamic registration optimization.
2. In particular, you must ensure that the Artix library containing the implementation of the AX interface (`ax_reg()` and `ax_unreg()` functions) is accessible to the third-party XA resource manager. The Artix library containing the AX interface implementation is, as follows:
    - Windows platforms—`it_xa.lib`.
    - UNIX platforms—`libit_xa.so` or `libit_xa.sl`.
3. Pass the value, `true`, to the `use_dynamic_registration_optimization` argument of the `IT_Bus::XATransactionManager::register_xa_resource()` function when you are registering the resource manager's XA switch.

It is important to ensure that both the transaction manager and the resource manager are aware of the dynamic registration optimization, because this optimization changes the nature of their interaction through the XA interface.

The following examples explain how to enable dynamic registration for certain third-party XA resource managers:

- Enabling dynamic registration for Oracle.
- Enabling dynamic registration for DB2.

Enabling dynamic registration for Oracle

In Oracle, dynamic registration is enabled by registering a special XA switch instance, `xaoswd`, instead of the normal XA switch instance, `xaosw`. You must also set the dynamic registration flag in the `register_xa_resource()` call to `true`. Sample code for registering an Oracle XA switch with dynamic registration enabled is shown in Example 16.

Example 16: *Dynamic Registration for the Oracle XA Resource Manager*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/transaction_system.h>
#include <it_bus_pdk/xa_transaction_manager.h>
#include <orbix_sys/xa.h>

#include <sqlca.h>
extern "C" IT_DECLSPEC_IMPORT xa_switch_t xaoswd;
...
xa_tx_mgr->register_xa_resource(
    &xaoswd,                  // Oracle XA dynamic switch
    "Oracle_XA+Acc=P/SCOTT/TIGER+SesTm=60+threads=true",
                              // Oracle open string
    "",                       // Oracle close string
    "",                       // resource manager identifier
    true,                     // dynamic registration = true
    false                     // single-threaded = false
);
```

To make the Artix implementation of the AX interface available to Oracle, you must also ensure that the `it_xa.lib` (Windows) or `libit_xa[.so][.sl]` (UNIX) library is placed in the link line *before* the Oracle client library.

Enabling dynamic registration for DB2

In DB2, dynamic registration is enabled by updating the DB2 configuration with the name of the Artix library that implements the AX interface. Enter the following db2 command:

```
db2 update dbm cfg using TP_MON_NAME <AX_LibNameRoot>
```

Where *`<AX_LibNameRoot>`* is the name of the relevant Artix library less the filename suffix—that is, `it_xa` (Windows) or `libit_xa.so`, `libit_xa.sl` (UNIX). The Artix library must also be made accessible to DB2 (by including it in the library path, or whatever is appropriate for your platform). You need to restart DB2 after issuing this command.

You must also set the dynamic registration flag in the `register_xa_resource()` call to `true`. Sample code for registering a DB2 XA switch with dynamic registration enabled is shown in Example 17.

Example 17: *Dynamic Registration for the DB2 XA Resource Manager*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/transaction_system.h>
#include <it_bus_pdk/xa_transaction_manager.h>
#include <orbix_sys/xa.h>

#ifdef WIN32
#define db2xa_switch (*db2xa_switch)
#endif
extern "C" IT_DECLSPEC_IMPORT xa_switch_t db2xa_switch;
...
xa_tx_mgr->register_xa_resource(
    &db2xa_switch,             // DB2 XA switch
    "<DB_Name>,<Username>,<Password>",
                               // DB2 open string
    "",                        // DB2 close string
    "",                        // resource manager identifier
    true,                      // dynamic registration = true
    false                      // single-threaded = false
);
```

Writing a Custom Resource

When do you need a custom resource?

Occasionally, it might be necessary to integrate a resource with the Artix transaction manager, where that resource does *not* support the XA standard. That is, the resource does *not* provide an XA switch that can be registered with a transaction manager.

Implementing a custom resource

In this case, you would have to write a *custom resource* by implementing a class that derives from the Artix `IT_Bus::TransactionParticipant` base class. This custom resource would implement the same functionality as a resource manager. Writing the custom resource is a fairly complex task that requires a good understanding of transaction systems.

Reference

For an introduction to some of the programming issues involved in writing a custom resource, see "Recoverable Resources" on page 121.

Server-Side Programming Model

Overview

When you register an XA resource with Artix, this typically has an impact on the way you program the XA resource itself. You should consult the documentation for the third-party resource in order to get a detailed overview of the resource's programming model under XA.

Although the programming model under XA is specific to a particular resource implementation, it is possible to make a few general observations on the programming model, as follows:

- Restrictions on connecting to and disconnecting from a resource.
- Transaction demarcation restrictions.
- Demarcation models under XA.

Restrictions on connecting to and disconnecting from a resource

Typically, an XA switch is implemented in such a way that `xa_open()` is responsible for opening a connection to the XA resource and `xa_close()` is responsible for closing the connection to the XA resource. In this case the Artix transaction manager, through calls to `xa_open()` and `xa_close()`, is responsible for opening and closing connections to the resource. Typically, this implies that you must avoid making any explicit calls (using the resource API) to open or close connections to the resource.

For example, when you register an XA switch for the Oracle database, the `xa_open()` and `xa_close()` calls are responsible for opening and closing connections to the database. When an XA switch is registered, Oracle forbids you from opening or closing a database connection explicitly.

Transaction demarcation restrictions

If your third-party resource has a native demarcation API—that is, a native API for beginning, committing and rolling back transactions—you must *not* use this native demarcation API when you have registered the resource's XA switch.

For example, if the resource is a database supporting embedded SQL, you must avoid using any embedded SQL statements that demarcate a transaction (whether explicitly or implicitly). At a minimum, you must avoid using the `EXEC SQL BEGIN`, `EXEC SQL COMMIT`, and `EXEC SQL ROLLBACK` commands.

Demarcation models under XA

When a resource's transactions are under the control of the Artix XA transaction manager, the programming model for transaction demarcation changes fundamentally. When implementing a WSDL operation in Artix, there are essentially three different cases to consider:

- Operation participating in a global transaction.
- Operation not participating in a global transaction.
- Operation sometimes participating in a global transaction.

Operation participating in a global transaction

If you are writing database code in the body of an operation which *always* participates in a global transaction (that is, incoming requests always include a transaction context), you should observe the following coding guidelines when accessing the database:

- Do not open or close any database connections—that is the responsibility of the transaction manager.
- Do not use any embedded SQL commands that demaracate transactions. For example, avoid using `EXEC SQL BEGIN`, `EXEC SQL COMMIT`, and `EXEC SQL ROLLBACK`.
- Do not use any native database APIs that demarcate transactions.
- Do not use the Artix `begin_transaction()`, `commit_transaction()`, and `rollback_transaction()` functions (defined on the `IT_Bus::TransactionSystem` object). A thread can only associate with one transaction at a time and the operation's thread is already associated with a global transaction.

Operation not participating in a global transaction

If you are writing database code in the body of an operation which *never* participates in a global transaction (that is, incoming requests never include a transaction context), you should observe the following coding guidelines when accessing the database:

- Do not open or close any database connections—that is the responsibility of the transaction manager.
- You can demarcate transactions, but you must not do so using embedded SQL commands or the native database API. Instead, use the demarcation functions provided by the Artix `IT_Bus::TransactionSystem` class—that is, `begin_transaction()`, `commit_transaction()`, and `rollback_transaction()`.

Operation sometimes participating in a global transaction

If you are writing database code in the body of an operation which sometimes participates in a global transaction (that is, incoming requests may include a transaction context), you should observe the following coding guidelines when accessing the database:

- Do not open or close any database connections—that is the responsibility of the transaction manager.
- Use the `TransactionSystem::within_transaction()` function to determine whether the operation is being called in the context of a global transaction or not.
- If `within_transaction()` returns `true`, do not attempt to demarcate a new transaction, as any database operations would be executed in the context of the global transaction.
- If you wish to demarcate a new transaction, separate to the global transaction, you must first disassociate the global transaction from the current thread using the `TransactionManager::detach_thread()` function. Once the global transaction has been detached, you can demarcate a new transaction using the demarcation functions provided by the Artix `IT_Bus::TransactionSystem` class—that is, `begin_transaction()`, `commit_transaction()`, and `rollback_transaction()`.
- If you have detached a transaction from the current thread it is imperative that it be re-attached before the operation exits, using the `TransactionManager::attach_thread()` operation.

CHAPTER 5

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.

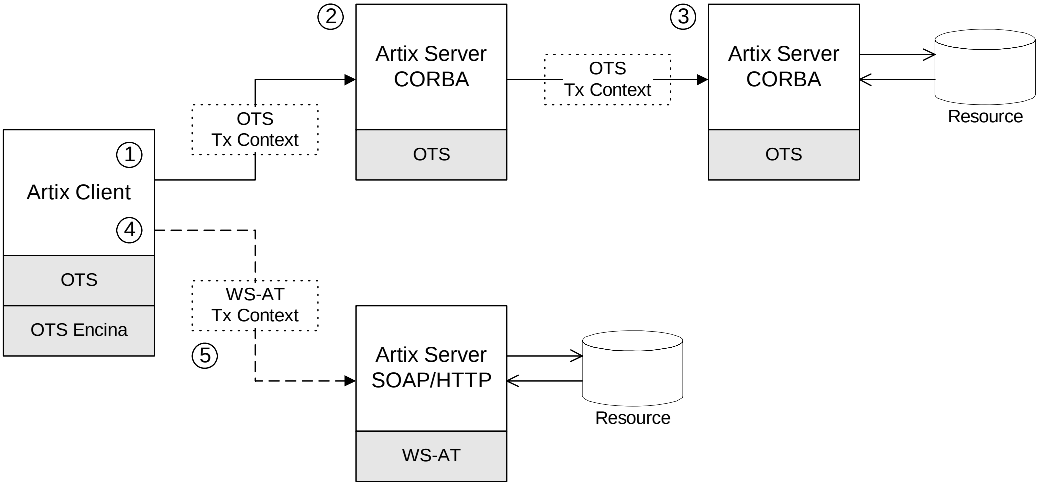
Propagation scenario

The propagation scenario shown in Figure 13 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 13, 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 13, 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 13: *Overview of Different Kinds of Transaction Propagation*



Scenario steps

The propagation scenario shown in Figure 13 can be described as follows:

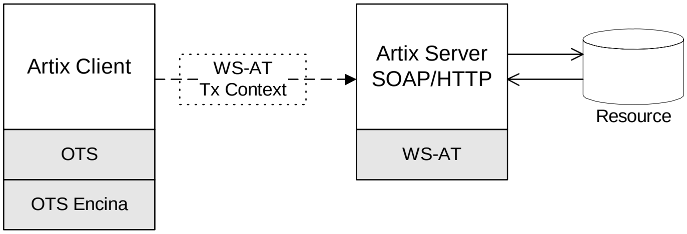
Stage	Description
1	The Artix client (which is configured to use the OTS Encina transaction system) initiates a transaction by calling the `begin_transaction()` function. The client then invokes a remote operation, which results in a request message being sent over an IIOP connection.
2	The request received 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 14 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 14: *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 `detach_thread()` and `attach_thread()` functions. After calling `detach_thread()`, 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 `attach_thread()`.

For more details on these functions, see "Threading" on page 99.

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

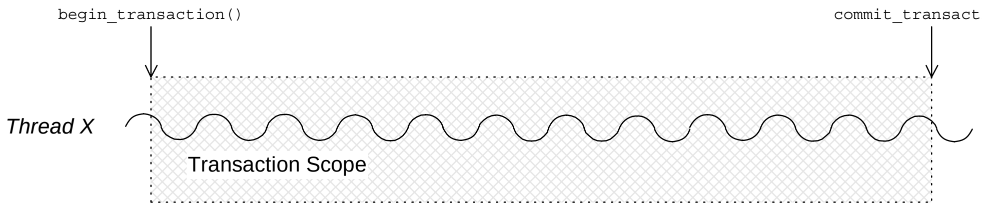
Overview

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

Default client threading model

Figure 15 shows the default threading model for transaction on the client side. When you call begin_transaction(), 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 commit_transaction() (or rollback_transaction(), if the transaction must be aborted), the transaction is deleted.

Figure 15: *Default Client Threading Model*



Transaction identifiers

A *transaction identifier* is an opaque identifier of type IT_Bus::TransactionIdentifier that identifies a transaction uniquely. Depending on the underlying transaction system, a transaction identifier can be downcast (using dynamic_cast<...>) to an implementation-specific transaction identifier.

For example, if OTS is the underlying transaction system, the transaction identifier can be downcast to an instance of an OTSTransactionIdentifier. The OTS transaction identifier provides access to implementation-specific features, such as the CosTransaction::Control class.

Controlling thread affinity

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

Example 18: *Functions for Controlling Thread Affinity*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionManager
      : public virtual RefCountedBase
    {
      public:
        virtual TransactionIdentifier* detach_thread()=0;

        virtual Boolean                attach_thread(
            TransactionIdentifier* tx_identifier
        ) = 0;

        virtual TransactionIdentifier* get_tx_identifier()=0;
        ...
};
```

These functions can be explained as follows:

- `detach_thread()`

  Detach the transaction from the current thread. After the call to `detach_thread()`, 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.

- `attach_thread()`

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

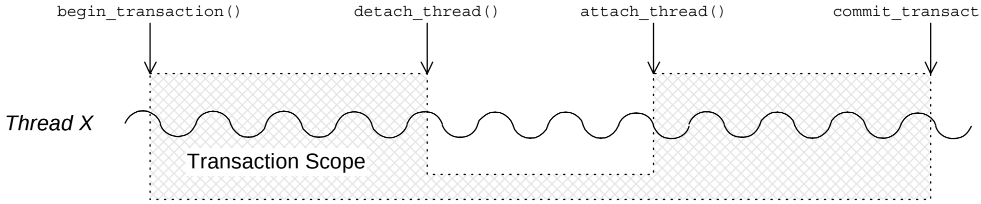
- `get_tx_identifier()`

  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 16 shows how to use the `detach_thread()` and `attach_thread()` functions 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.

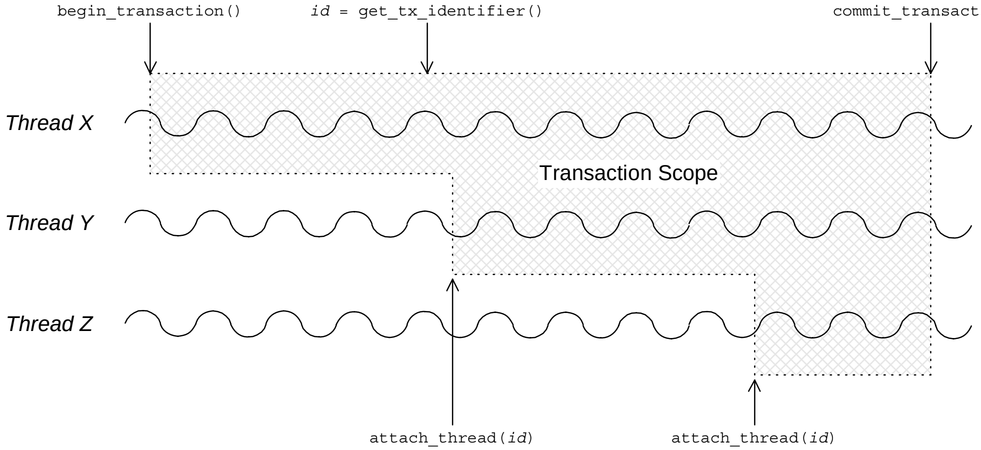
Figure 16: *Detaching and Re-Attaching a Transaction to a Thread*



Attaching a transaction to multiple threads

Figure 17 shows how to use the `get_tx_identifier()` and `attach_thread()` functions to associate a transaction with multiple threads. The `get_tx_identifier()` function 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 17: *Attaching a Transaction to Multiple Threads*

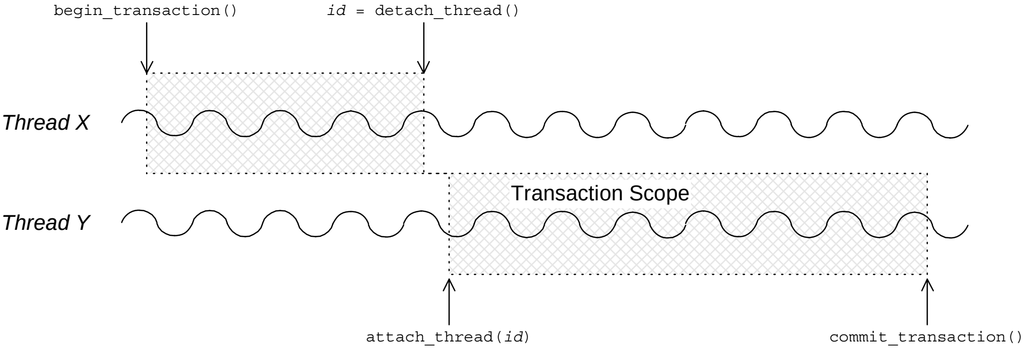


Note: Some transaction systems do not allow you to associate multiple threads with a transaction. In this case, an `attach_thread()` call fails (returning `false`), if you attempt to attach a second thread to the transaction.

Transferring a transaction from one thread to another

Figure 18 shows how to use the `detach_thread()` and `attach_thread()` functions to transfer a transaction from thread X to thread Y. The transaction ID returned from the `detach_thread()` call must be passed to thread Y, enabling it to attach the transaction.

Figure 18: *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 `attach_thread()` 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:

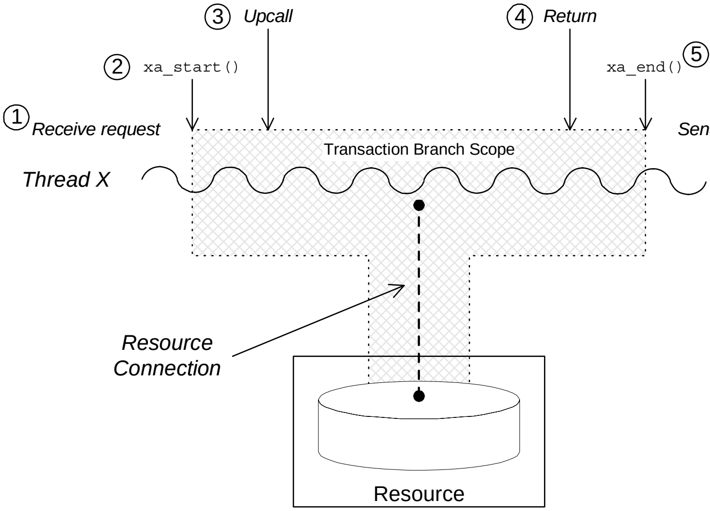
- Auto-association.
- Multiple registered resources.
- Multi-threaded resource connections.
- Dynamic registration.

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

Figure 19: *Auto-Association with a Single Registered Resource*



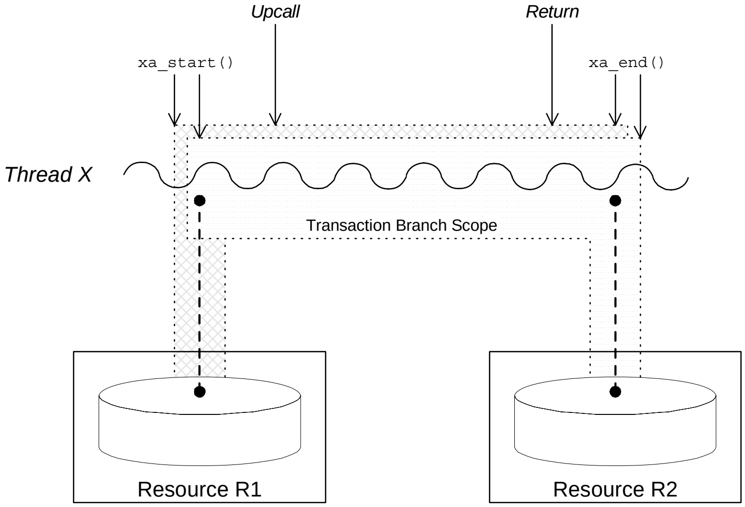
The sequence of events shown in Figure 19 on page 105 can be explained as follows:

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 20 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 20: *Auto-Association with Multiple Registered Resources*



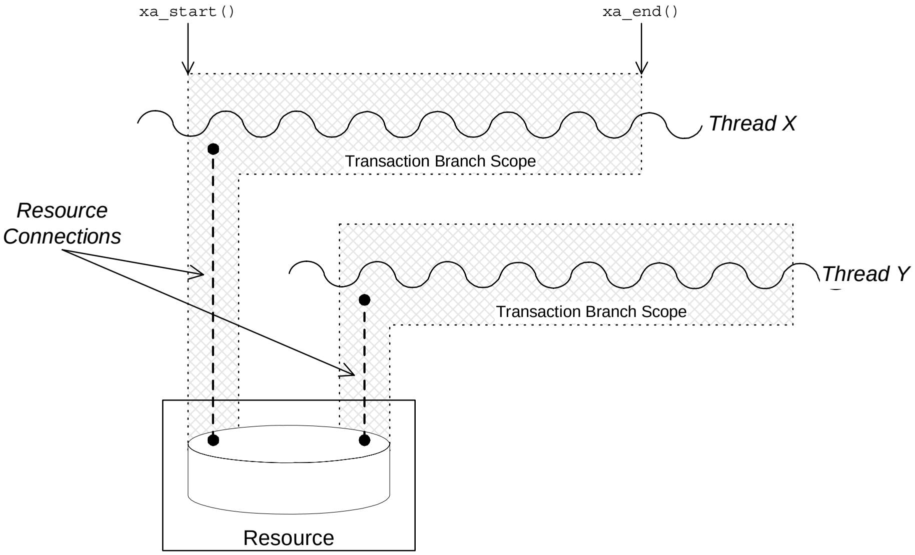
After the upcall, any application code in the servant function that interacts with one of the resources (either resource R1 or resource R2) is implicitly governed by a global transaction, where the global transaction ID has been obtained from the received transaction context.

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 21 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 21: *Database Resource Operating in Multi-Threaded Mode*

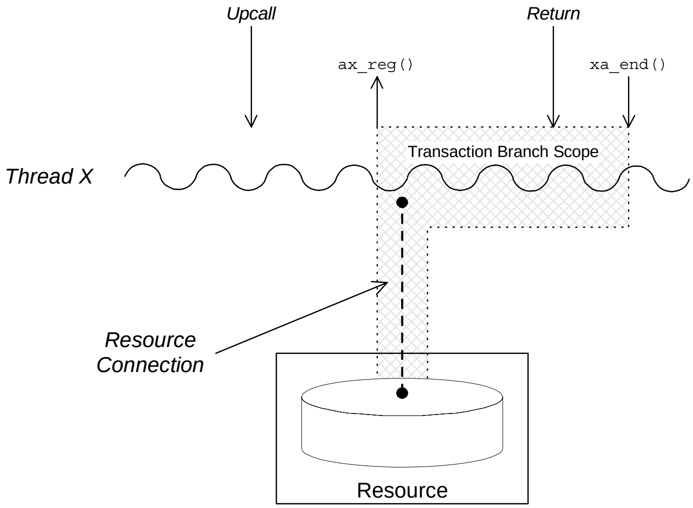


To use the multi-threaded resource mode, both the resource manager and the Artix transaction manager must be configured appropriately. For details of how to configure the Artix transaction manager in this case, see "is_single_threaded_resource" on page 79.

Dynamic registration

As shown in Figure 22, some XA resources support an alternative algorithm, *dynamic registration*, for associating a global transaction with a locally registered resource.

Figure 22: *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.

For details of how to configure dynamic registration, see "Dynamic Registration Optimization" on page 81.

CHAPTER 7

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 4 summarizes the characteristics of the various Artix transaction systems.

Table 4: *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:

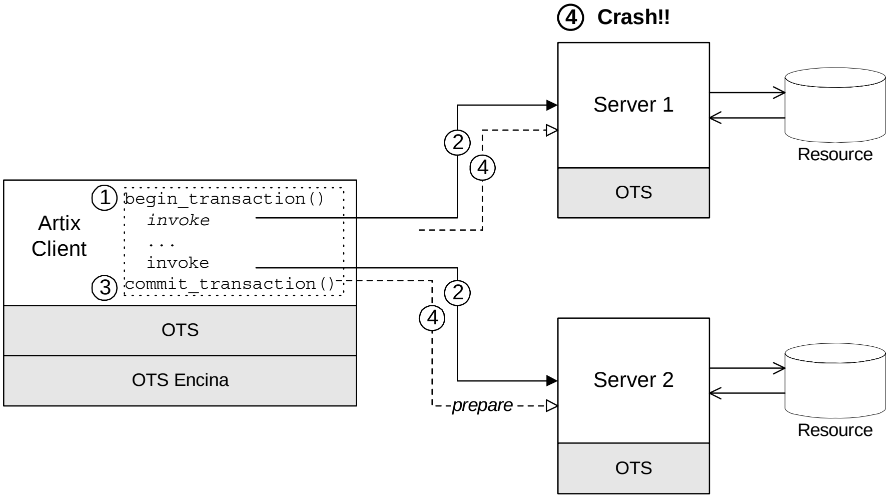
Server Crash before or during Prepare Phase

Overview

Figure 23 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 23: *Server Crash before or during the Prepare Phase*



Steps leading to crash

As shown in Figure 23, 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 23, it will be a rollback).

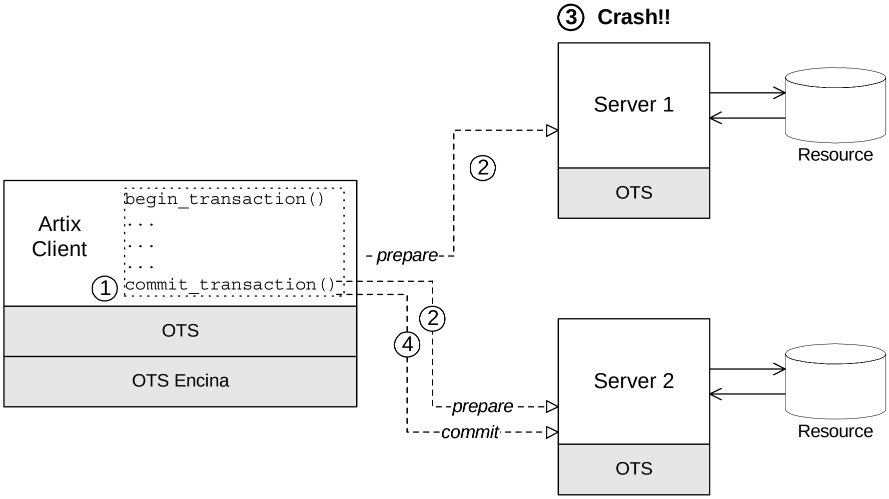
Server Crash after Prepare Phase

Overview

Figure 24 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 24: *Server Crash after the Prepare Phase*



Steps leading to crash

As shown in Figure 24, 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 24 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.

CHAPTER 8

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

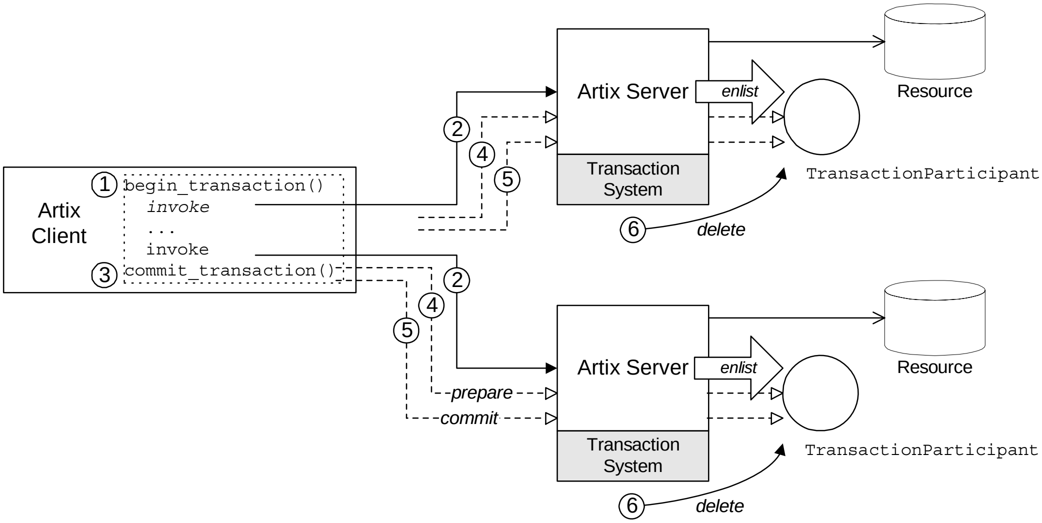
Overview

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 25 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 25: *Transaction Participants in a 2-Phase Commit Protocol*



Participants in a 2-phase commit

As shown in Figure 25, the transaction participants participate in a two-phase commit as follows:

Stage	Description
1	The client calls `begin_transaction()` 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 `commit_transaction()` 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 implement a transaction participant, define a class that inherits from the `IT_Bus::TransactionParticipant` base class and implement all of its member functions.

TransactionParticipant member functions

Example 19 shows the public member functions of the `IT_Bus::TransactionParticipant` class.

Example 19: *The IT_Bus::TransactionParticipant Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionParticipant
      : public virtual RefCountedBase
    {
      public:
        virtual ~TransactionParticipant();

        enum VoteOutcome {
            VoteCommit,
            VoteRollback,
            VoteReadOnly
        };

        // 1PC Functions.
```

Example 19: *The IT_Bus::TransactionParticipant Class*

```
        virtual void commit_one_phase()=0;

        // 2PC Functions.
        virtual VoteOutcome prepare()=0;
        virtual void        commit()=0;
        virtual void        rollback()=0;

        // Getting the transaction manager.
        virtual String
        preferred_transaction_manager()=0;

        virtual void
        set_manager(
            TransactionManager*    tx_manager
        )=0;
        ...
    };
    typedef Var<TransactionParticipant>
        TransactionParticipant_var;
    typedef TransactionParticipant* TransactionParticipant_ptr;
};
```

1PC callback function

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

- `commit_one_phase()`—this function should make permanent any changes associated with the current transaction.

2PC callback functions

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

- `prepare()`—called during *phase one* of a two-phase commit. Before returning, this function 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:

    - `IT_Bus::TransactionParticipant::VoteCommit`—vote to commit the transaction.
    - `IT_Bus::TransactionParticipant::VoteRollback`—vote to roll back the transaction. For example, you would return `VoteRollback`, if an error occurred while attempting to write the recovery log.
    - `IT_Bus::TransactionParticipant::VoteReadOnly`—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 function 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 function 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 functions are relevant to this callback behavior:

- `preferred_transaction_manager()`—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).

- set_manager()—called after the preferred_transaction_manager() call. The transaction system calls set_manager() 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 130.

Enlisting a transaction participant

Example 20 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 20 shows a sample implementation of a WSDL operation, transactional_op(), which is called in the context of a transaction.

Example 20: *Example of Enlisting a Transactional Participant*

```
   // C++
   void
   HelloWorldServantImpl::transactional_op(
       const IT_Bus::String value
   ) IT_THROW_DECL((IT_Bus::Exception))
   {
       cout << "HelloWorld transactional_op() called" << endl;

1      IT_Bus::Bus_var bus = this->get_bus();
2      if (bus->transactions().within_transaction())
       {
           cout << "This is a transaction" << endl;

3          TXParticipant * participant = new TXParticipant(this);
4          bus->transactions().get_transaction_manager().enlist(
               participant,
               true
           );

5          // Implementation of 'transactional_op()' comes here.
           // Includes writing to DB or other persistent resources.
           // (not shown)
           ...
       }
       else
       {
           cout << "No transaction" << endl;
```

Example 20: *Example of Enlisting a Transactional Participant*

```
        IT_Bus::Exception ex("Invocation not in transaction");
        throw ex;
    }
}
```

The preceding code example can be explained as follows:

1. The `get_bus()` function is a standard servant function that returns a stored reference to the Bus instance.
2. In this example, the `transactional_op()` operation *requires* a transaction. If it is not called in the context of a transaction, it raises an exception back to the client.

   It is an implementation decision whether or not an operation should require a transaction. In some cases, it may be appropriate for the operation to proceed with or without a transaction.
3. The `TXParticipant` class is a sample participant class, which is implemented by inheriting from `IT_Bus::TransactionParticipant`.

   In this example, a new `TXParticipant` instance is created every time `transactional_op()` is called.
4. This line enlists the participant 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.
5. The implementation of `transactional_op()` involves writing to a persistent resource. The committing or rolling back of any changes to this persistent resource is controlled by the enlisted `TXPersistent` instance.

Alternatives to the Artix transaction participant

Implementing and enlisting an Artix `TransactionParticipant` class is not the only way to make a WSDL operation transactional. By drilling down to the underlying transaction manager type (for example, `IT_Bus::OTSTransactionManager`) it is sometimes possible to use an alternative API supported by a specific transaction system.

For example, the following demonstration shows how to use the OTS transaction system:

ArtixInstallDir`/artix/`*Version*`/demos/transactions/legacy_ots_integration`

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.

Using interposition

As an example of interposition, consider a service that loads the WS-AT transaction system (for example, see “Configuring Non-Recoverable WS-AT” on page 59), but actually implements the transaction functionality using the CORBA OTS programming interface. In this case, it is necessary for the `TransactionParticipant` implementation to request explicitly an OTS transaction manager, instead of the default WS-AT transaction manager.

Example 21 shows the implementation of the `preferred_transaction_manager()` function and the `set_manager()` function for the transaction participant implementation, `TxParticipant`.

Example 21: *Example of a TransactionParticipant that Uses Interposition*

```
// C++
...
IT_Bus::String
TXParticipant::preferred_transaction_manager()
```

Example 21: *Example of a TransactionParticipant that Uses Interposition*

```
{
    return IT_Bus::TransactionSystem::OTS_TRANSACTION_TYPE;
}

void
TXParticipant::set_manager(
    IT_Bus::TransactionManager*    tx_manager
)
{
    m_ots_tx_manager =
       dynamic_cast<IT_Bus::OTSTransactionManager*>(tx_manager);
}
```

When Artix calls back on `set_manager()`, it passes a transaction manager object, `tx_manager`, of `OTSTransactionManager` type. There is no need to query the type of the `tx_manager` object before downcasting it, because its type is already specified by the `preferred_transaction_manager()` callback.

CHAPTER 9

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, define a class that inherits from the `IT_Bus::TransactionNotificationHandler` base class and implement all of its member functions.

TransactionNotificationHandler base class

Example 22 shows the `TransactionNotificationHandler` base class. These functions will only be called if an appropriate notification mechanism is available in the underlying transaction system.

Example 22: *The IT_Bus::TransactionNotificationHandler Class*

```
// C++
namespace IT_Bus
{
    class IT_BUS_API TransactionNotificationHandler
      : public virtual RefCountedBase
    {
      public:
        ...
        virtual void commit_initiated(
            TransactionIdentifier_ptr   tx_identifier
        )=0;
        virtual void committed()=0;
        virtual void aborted()=0;
        ...
    };

    typedef Var<TransactionNotificationHandler>
        TransactionNotificationHandler_var;
    typedef TransactionNotificationHandler*
        TransactionNotificationHandler_ptr;
};
```

Notification callback functions

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

- commit_initiated()—informs the handler that a commit has been initiated. This function 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 23 shows how to enlist a sample notification handler, NotificationHandlerImpl.

Example 23: *Example of Enlisting a Notification Handler*

```
// C++
IT_Bus::Bus_var bus = ... // Get reference to Bus object
if (bus->transactions().within_transaction())
{
    // Enlist notification handler
    NotificationHandlerImpl * handler
         = new NotificationHandlerImpl();
    TransactionManager& tx_manager
        = bus->transactions().get_transaction_manager()
   tx_manager.enlist_for_notification(handler);
}
else
{
    IT_Bus::Exception ex("Invocation not in transaction");
    throw ex;
}
```

CHAPTER 10

Exposing Artix as an XA Resource

You can expose Artix as an XA resource manager by registering the Artix XA switch with a third-party XA transaction manager.

In this chapter

This chapter discusses the following topics:

Introduction to the Artix XA Resource Manager

Overview

The most common use case for XA in Artix is where you register a third-party resource manager (such as an Oracle DB) with Artix and Artix is responsible for coordinating the transactions.

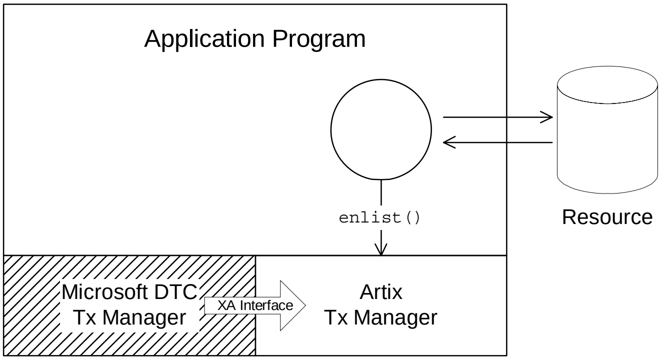
It is possible, however, to reverse these roles, so that Artix assumes the role of an XA resource manager and a *foreign* transaction manager is responsible for coordinating the transactions in Artix. To support this use case, Artix provides an XA switch, which can be registered with the foreign transaction manager. Although this use case is much less common than the former, there are two possible scenarios where you might want to expose Artix as an XA resource manager, as follows:

- Scenario 1 - local resource.
- Scenario 2 - remote resource.

Scenario 1 - local resource

In the scenario shown in Figure 26, the Artix XA resource manager is registered with the Microsoft DTC transaction manager and has responsibility for managing a local resource. This scenario could arise, for example, if you have already implemented a recoverable resource using the Artix transaction API and you now want to integrate the resource with a third party transaction manager (such as Microsoft DTC).

Figure 26: *Artix XA Resource Manager Manages a Local Resource*

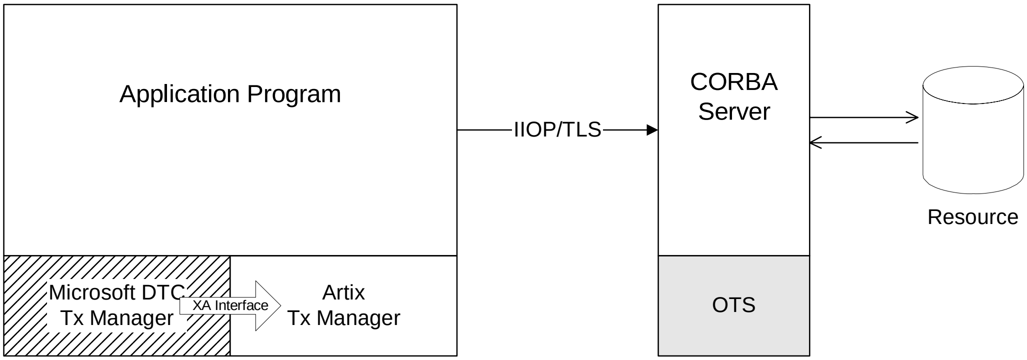


Of course, it is unlikely that you would implement an Artix recoverable resource just for this purpose. But if you already have such an implementation, the Artix XA switch enables you to integrate it rapidly with a third-party transaction manager.

Scenario 2 - remote resource

In the scenario shown in Figure 27, the Artix XA resource manager is registered with the Microsoft DTC transaction manager, but the managed resource (or resources) belongs to a remote server. In this case, the Artix Bus is effectively being used as a transport stack to facilitate interoperability with a remote server that manages a transactional resource. Artix uses the IIOP protocol to communicate with the CORBA server and the OTS standard is used to coordinate the distributed CORBA transactions.

Figure 27: *Artix XA Resource Manager Manages a Remote Resource*



To program this example, you would demarcate the transactions using the relevant API from Microsoft DTC. To access the operations supported by the remote CORBA server, use the Artix programming API (the relevant function signatures for the operations are provided in the Artix stub code).

How to use the Artix XA switch

To use the Artix XA switch with a third-party transaction manager, perform the following steps:

1. *Obtain the Artix XA switch*—you need to obtain a pointer to a struct of `xa_switch_t` type (as specified by the XA standard). Artix provides a number of ways of obtaining the Artix XA switch instance. See "Obtaining an Artix XA Resource Manager" on page 141 for details.
2. *Register the Artix XA switch*—after obtaining a pointer to the Artix XA switch, you must register the switch instance with your third-party transaction manager. Typically, the registration step consists of a single function call that requires you to provide an open string and a close string (for details of the Artix-specific open and close strings, see "Artix XA Open and Close Strings" on page 146).

   For details of how to register the XA switch, consult the documentation for your third-party transaction manager.
3. *Configure the Artix XA resource manager*—the Artix XA resource manager needs to be configured as described in "Configuring the Artix XA Resource Manager" on page 148.
4. *Observe the usual XA programming conventions*—according to the usual XA programming conventions, once you have registered the Artix XA switch, the third-party transaction manager, and *not* the Artix transaction system, is responsible for transaction demarcation. This implies that you should not use the `begin_transaction()`, `commit_transaction()`, and `rollback_transaction()` functions from the `TransactionSystem` class to demarcate transactions.

Obtaining an Artix XA Resource Manager

Overview

Artix supports several different ways of obtaining an XA resource manager. Essentially, this involves providing a pointer to the xa_switch_t struct. The different approaches to obtaining the XA switch are described in the following subsections.

In this section

This section contains the following subsections:

Obtaining the XA Switch from a Global Function

Overview

In this scenario, you obtain a pointer to the Artix `xa_switch_t` instance by calling a global function. Use this approach when the external transaction manager provides an API function to enlist the XA switch and you do *not* have an instance of an Artix Bus.

GetXaSwitch() function

To obtain a pointer to the Artix XA switch, call the `GetXaSwitch()` function, which is a C function defined in the global scope. The `GetXaSwitch()` function takes no arguments and has a return type of `xa_switch_t *`.

Example

Example 24 shows how to obtain an Artix XA switch using the `GetXaSwitch()` function. Remember to include the `it_bus/xa_switch.h` header file.

Example 24: *Obtaining the Artix XA Switch Using GetXaSwitch()*

```
// C++
#include <it_bus/xa_switch.h>
....
xa_switch_t* artix_xa_switch = ::GetXaSwitch();
....
```

Required library

You need to link your code with the Artix `it_xa_switch` library.

Obtaining the XA Switch from a Bus Instance

Overview

In this scenario, you obtain a pointer to the Artix `xa_switch_t` instance through an `IT_Bus::XATransactionManager` object, which you can obtain from the Artix Bus. Use this approach when the external transaction manager provides an API function to enlist the XA switch and you *do* have an instance of an Artix Bus.

get_xa_switch() function

To obtain a pointer to the Artix XA switch, call the `get_xa_switch()` function, which is a member of the `IT_Bus::XATransactionManager` class. The `get_xa_switch()` function takes no arguments and has a return type of `xa_switch_t *`.

Example

Example 25 shows how to obtain an Artix XA switch from the Bus instance, by calling the `IT_Bus::XATransactionManager::get_xa_switch()` function.

Example 25: *Obtaining the Artix XA Switch from a Bus Instance*

```
// C++
#include <it_bus/bus.h>
#include <it_bus/transaction_system.h>
#include <it_bus_pdk/xa_transaction_manager.h>

IT_Bus::Bus_var bus = ...
...
IT_Bus::XATransactionManager& xa_tx_mgr = dynamic_cast
<
    IT_Bus::XATransactionManager,
    bus->transactions().get_transaction_manager(
        IT_Bus::TransactionSystem::XA_TRANSACTION_TYPE
    )
>;
xa_switch_t* artix_xa_switch = xa_tx_mgr->get_xa_switch();
```

Required library

You need to link your code with the Artix `it_bus` library.

Obtaining the XA Switch from a Switch Load File

Overview

In this scenario, the third-party transaction manager obtains the Artix XA switch by loading a shared library file (the *switch load file*). Use this approach when the external transaction manager does *not* provide an API function to enlist the XA switch, but does support switch load files.

Using a switch load file

To use a switch load file, you supply the third-party transaction manager (TM) with the name and location of the relevant shared library or DLL. When the TM loads the switch load library file, it calls a particular function to obtain the XA switch instance. The mechanisms that are used to load the switch file and obtain the XA switch instance are specific to the particular TM. Refer to your third-party TM documentation for details.

Default switch load file

Artix provides a default switch load file: the `it_xa_switch` library. The precise name of the default switch load file depends on the platform, as shown in Table 5.

Table 5: *Default Switch Load File for Artix on Various Platforms*

Platform	Link Library	Shared Library or DLL
Windows VC++ 6.0	`it_xa_switch.lib`	`it_xa_switch5_vc60.dll`
Windows VC++ 7.1	`it_xa_switch.lib`	`it_xa_switch5_vc71.dll`
Solaris	`libit_xa_switch.so`	`libit_xa_switch_sc53.so.5`
HP-UX	`libit_xa_switch.sl`	`libit_xa_switch_acca0331.5`
AIX	`libit_xa_switch.a`	`libit_xa_switch5_xlc60.so`

The default switch load file exposes the C functions shown in Example 26.

Example 26: *Functions in the Default Artix Switch Load File*

```
/* C */
xa_switch_t* GetXaSwitch()  /* for use by Microsoft DTC */
xa_switch_t* MQStart()      /* for use by MQSeries      */
```

Example of using a switch load file with Microsoft DTC

For example, if you are writing a COM+ application on the Windows platform, you can use Microsoft DTC to load a switch load file. Microsoft DTC provides the following function to load a switch load file:

```
// In IDtcToXaMapper

HRESULT RequestNewResourceManager(
  CHAR * pszDSN,
  CHAR * pszClientDllName,
  DWORD * pdwRMCookie
);
```

The argument, `pszDSN`, is used as the open string for the XA switch; the argument, `pszClientDllName`, is the name of the switch load file; and the argument, `pdwRMCookie`, is a cookie used to identify the resource manager loaded by this call. See Opening an XA Connection in the Microsoft documentation for more details.

Creating a custom switch load file

You can create your own custom switch load file, as follows. Implement the global function required by your third-party TM (usually a simple wrapper function around the Artix `GetXaSwitch()` function). Then compile this code as a shared library or DLL, as appropriate for the platform you are working on.

For example, the following code shows the implementation of a load switch file for use with MQ-Series:

```
// C++
#include <cmqc.h>
#include<it_bus/xa_switch.h>

struct xa_switch_t * MQENTRY MQStart(void)
{
    return ::GetXaSwitch();
}
```

The header, `cmqc.h`, is an MQ-Series header file that defines the signature of the `MQStart()` function. The `MQSeries()` function is called automatically by MQ-Series after it loads the switch file.

Note: You do not actually have to implement the `MQStart()` function, because it is already defined in the default switch load file.

Artix XA Open and Close Strings

Overview

When registering the Artix XA switch with a third-party transaction manager (TM), the TM usually requires you to supply an *open string* and a *close string*. These strings are used as follows:

- The TM passes the open string to the `xa_open()` function, when it opens a connection to the Artix resource manager,
- The TM passes the close string to the `xa_close()` function, when it closes the connection to the Artix resource manager.

The format of the open and close strings is specific to an XA switch implementation. Therefore, just as Oracle and Sybase have their own proprietary formats for their open and close strings, the Artix XA switch defines proprietary open string and close string formats, as described here.

Specifying open and close strings

The mechanism for specifying the open and close strings is defined by the third-party TM implementation. See your TM documentation for details.

Open string

For the Artix XA switch, the open string must be an *Artix Bus ID*. In practice, the Bus ID is equivalent to the name of an Artix configuration scope.

For example, if you choose a Bus ID equal to `xa_bus.ots_lite_coordinated`, Artix will initialize a Bus object that takes its configuration from the `xa_bus.ots_lite_coordinated` scope in the Artix configuration file (for example, see the configuration scope in Example 27).

Close string

For the Artix XA switch, there are two cases to consider for the close string:

- If the Artix XA switch is obtained either from a global function (see “Obtaining the XA Switch from a Global Function” on page 142) or from a switch load file (see “Obtaining the XA Switch from a Switch Load File” on page 144), the close string should usually be `shutdown=true`. This close string tells the Bus to call `IT_Bus::Bus::shutdown(true)` when `xa_close()` is called by the TM.
- If the Artix XA switch is obtained from a Bus instance (see “Obtaining the XA Switch from a Bus Instance” on page 143), the close string should be empty, `""`, implying that the caller will take care of calling `bus->shutdown()`.

Configuring the Artix XA Resource Manager

Overview

When Artix is exposed as an XA resource manager, it has the same configuration requirements as an Artix application that uses the OTS transaction coordinator. Two alternative configurations can be used:

- Configuration for a single resource.
- Configuration for multiple resources.

Configuration for a single resource

Example 27 shows the configuration, xa_bus.ots_lite_coordinated, which is suitable for an Artix XA resource manager that manages a single resource. This type of configuration is suitable for the scenario shown in Figure 26 on page 138.

Example 27: *Resource Manager Configuration for a Single Resource*

```
# Artix Configuration File
xa_bus
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop", "ots"];
    plugins:ots:default_ots_policy="adapts";
    plugins:bus:default_tx_provider:plugin=
   "xa_transaction_provider";

    ots_lite_coordinated
    {
        initial_references:TransactionFactory:plugin ="ots_lite";
    };
};
```

The presence of the ots plug-in is required in the list of ORB plug-ins. The default_tx_provider setting ensures that the xa_transaction_provider plug-in is loaded by default. Strictly speaking, the latter setting is unnecessary. Whenever a third-party transaction manager attempts to obtain a reference to the Artix XA switch, the xa_transaction_provider plug-in is loaded automatically.

To use this configuration with the Artix XA switch, pass xa_bus.ots_lite_coordinated as the open string.

Configuration for multiple resources

Example 28 shows the configuration, `xa_bus.ots_encina_coordinated`, which is suitable for an Artix XA resource manager that manages multiple resources. This type of configuration is suitable for the scenario shown in Figure 27 on page 139.

Example 28: *Resource Manager Configuration for Multiple Resources*

```
# Artix Configuration File
xa_bus
{
    orb_plugins = ["local_log_stream", "iiop_profile", "giop",
   "iiop", "ots"];
    plugins:ots:default_ots_policy="adapts";
    plugins:bus:default_tx_provider:plugin=
   "xa_transaction_provider";

    ots_encina_coordinated
    {
        plugins:ots_encina:direct_persistence = "true";
        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";
        initial_references:TransactionFactory:plugin =
   "ots_encina";
        plugins:ots_encina:initial_disk = "encina.log";
        plugins:ots_encina:initial_disk_size = "1";
        plugins:ots_encina:restart_file = "encina_restart";
        plugins:ots_encina:backup_restart_file =
   "encina_restart.bak";
    };
};
```

The presence of the `ots` plug-in is required in the list of ORB plug-ins.

To use this configuration with the Artix XA switch, pass `xa_bus.ots_encina_coordinated` as the open string.

Note: There might be more resources registered than you think. In certain cases, Artix automatically registers extra resources to support interposition. See “Limitation of using OTS Lite with propagation” on page 96.

Interoperating with WS-AT transactions

The Artix XA resource manager can also interoperate over SOAP with applications that require WS-AT transactions. This requires no special configuration. Artix automatically loads the required WS-AT plug-ins, if they are needed.

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).

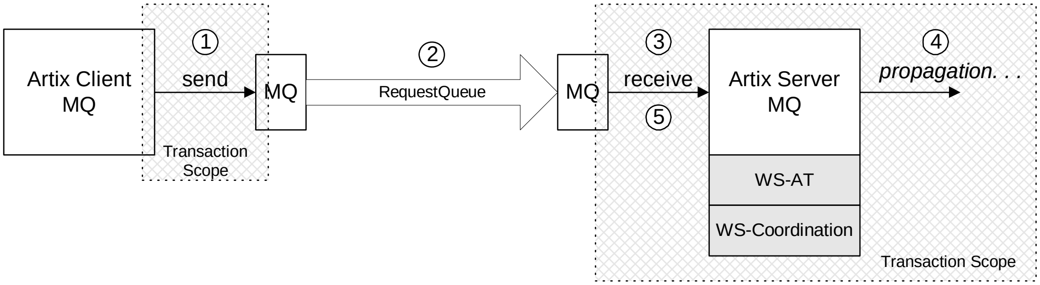
On the client side, MQ transactions follow a completely different model from Artix transactions. On the server side, however, the MQ transaction is integrated with an Artix transaction, so that an incoming message is considered to have been processed, only if the Artix transaction completes successfully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

Oneway Invocations

Oneway invocation scenario

Figure 28 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 28: *Oneway Operation Invoked Over an MQ Transport with MQ Transactions Enabled*



Description of oneway invocation

The oneway operation invocation shown in Figure 28 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 159).

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 29.

Example 29: *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 30.

Example 30: *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).

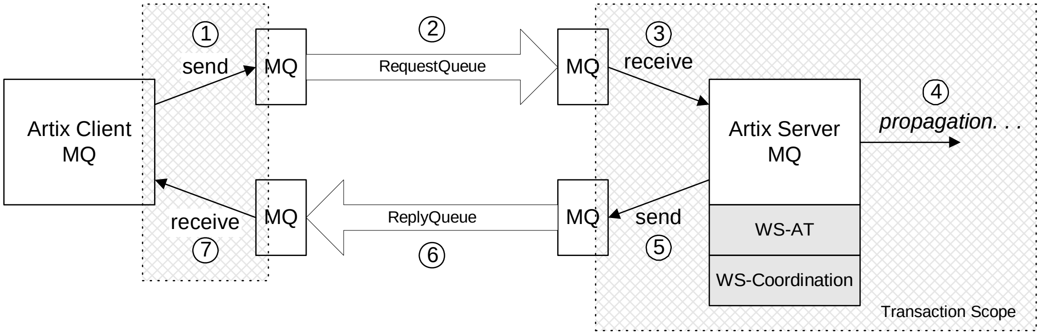
For details of how to select a transaction system, see “Selecting a Transaction System” on page 51.

Synchronous Invocations

Synchronous invocation scenario

Figure 29 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 29: *Synchronous Operation Invoked Over the MQ Transport with MQ Transactions Enabled*



Description of synchronous invocation

The synchronous operation invocation shown in Figure 29 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 159).
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 31.

Example 31: *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 32.

Example 32: *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 32: *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 51.

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.

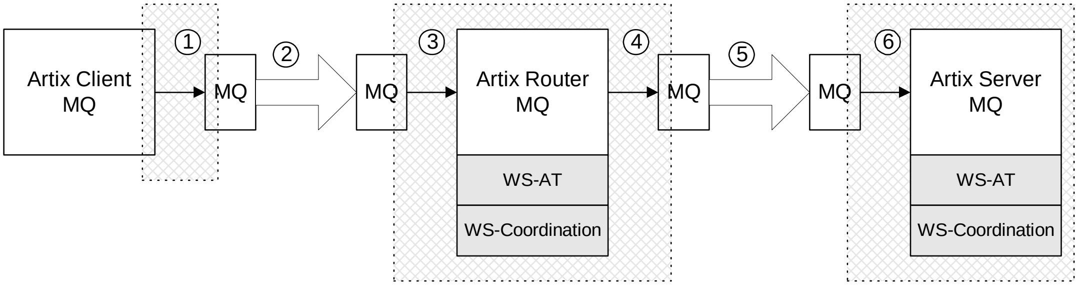
For more details about programming with Artix contexts, see the *C++ Programmer's Guide*.

Router Propagating MQ Transactions

Router scenario

Figure 30 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 30: *Router Propagating an MQ Transaction*



Description of router invocation

The router invocation shown in Figure 30 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 51.

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 51. 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