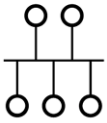

Orbix®

CORBA OTS Guide Java

Version 6.3, December 2005

Making Software Work Together™

Updated: 09-Dec-2005

M 3 2 1 6

Contents

List of Figures

List of Tables

Preface

Orbix OTS is a full implementation from IONA Technologies of the interoperable transaction service as specified by the Object Management Group. Orbix OTS complies with the following specifications:

- CORBA 2.3
- OTS 1.2
- GIOP 1.2 (default), 1.1, and 1.0

If you need help with this or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `docs-support@iona.com`.

Audience

This guide is intended to help you become familiar with the transaction service, and shows how to develop applications with it. This guide assumes that you are familiar with CORBA concepts, and with Java.

This guide does not discuss every interface and its operations in detail, but gives a general overview of the capabilities of the transaction service and how various components fit together. For detailed information about individual operations, refer to the *CORBA Programmer's Reference*.

Related Documentation

For the latest version of all IONA product documentation, see the IONA web site:

`http://www.iona.com/docs/`

Organization of this Guide

This guide is divided as follows:

Chapter 1 provides a brief overview of the basic concepts involved in using the transactions service.

Chapter 2 provides an overview of the transaction service's interfaces. It also provides information on the X/Open XA interfaces and how to use them to interact with compliant resources.

Chapter 3 is a simple example of the steps involved in developing a client that uses the transaction service. It discusses the basic steps required to use transactions and the concepts behind them.

Chapter 4 covers transaction demarcation. It covers both using the transactions `Current` object, which is convenient but limited, and using the `TransactionFactory` and the `Terminator` interfaces to directly manipulate demarcation.

Chapter 5 covers how to control how the transaction is propagated to its target object through the use of POA policies.

Chapter 6 provides a detailed discussion on how to use the Java Transaction API.

Chapter 7 covers some additional areas of transaction management. This includes synchronization objects, transaction identity and status operations, relationships between transactions and recreating transactions.

Chapter 8 describes the `CosTransactions::Resource` interface; how resource objects participate in the transaction protocols and the requirements for implementing resource objects.

Chapter 9 describes how the Orbix OTS interoperates with older releases of Orbix and with other OTS implementations including the Orbix 3 OTS.

Chapter 10 discusses the plugins that implement the transaction service and options for deploying them.

Additional Related Resources

The IONA knowledge base contains helpful articles, written by IONA experts, about the Orbix and other products. You can access the knowledge base at the following location:

`http://www.iona.com/support/knowledge_base/`

The IONA update center contains the latest releases and patches for IONA products:

`http://www.iona.com/support/updates/`

Typographical Conventions

This guide uses the following typographical conventions:

`Constant width`	Constant width (courier font) in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class. Constant width paragraphs represent code examples or information a system displays on the screen. For example: `#include <stdio.h>`
Italic	Italic words in normal text represent *emphasis* and *new terms*. Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example: `% cd /users/`*`your_name`* **Note:** Some command examples may use angle brackets to represent variable values you must supply. This is an older convention that is replaced with ***italic*** words or characters.

Keying Conventions

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, a prompt is not used.
%	A percent sign represents the UNIX command shell prompt for a command that does not require root privileges.
#	A number sign represents the UNIX command shell prompt for a command that requires root privileges.
>	The notation > represents the DOS, Windows NT, Windows 95, or Windows 98 command prompt.
. . . . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.

[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

CHAPTER 1

Transaction Service

This chapter describes the transaction processing capabilities of Orbix, showing how to use the Object Transaction Service (OTS) for transaction demarcation, propagation and integration with resource managers. The Java Transaction API (JTA) interfaces and integration with JTA compliant resource managers is also discussed.

In this chapter

This chapter discusses the following topics:

About Transactions

What is a transaction?

Orbix gives separate software objects the power to interact freely even if they are on different platforms or written in different languages. Orbix 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.

Transaction support in Orbix

To support the development of object-oriented, distributed, transaction-processing applications, Orbix offers:

- An implementation of the Object Management Group's Object Transaction Service (OMG OTS).
- An implementation of the J2EE Java Transaction API (JTA) providing integration with resource managers.
- A pluggable architecture that supports both a lightweight OTS implementation and a full recoverable two-phase-commit (2PC) implementation.

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.

Transaction Managers

Purpose of a Transaction Manager

Most resource managers, for example databases and message queues, provide support for native transactions. However, when an application wants two or more resource managers to be part of the same transaction some third party must provide the necessary coordination to ensure the ACID properties are guaranteed for the distributed transaction. This is where the concept of an transaction manager that is independent of the individual resource manager comes in.

The application uses the transaction manager to create the transaction. Each resource manager accessed during the transaction becomes a participant in the transaction. Then when the application completes the transaction, either with a commit or rollback request, the transaction manager communicates with each resource manager.

Two-phase commit protocol

When there are two of more participants involved in a transaction the transaction manager uses a two-phase-commit (2PC) protocol to ensure that all participants agree on the final outcome of the transaction despite any failures that may occur. Briefly the 2PC protocol works as follows:

- In the first phase, the transaction manager sends a “prepare” message to each participant. Each participant responds to this message with a vote indicating whether the transaction should be committed or rolled back.
- The transaction manager collects all the prepare votes and makes a decision on the outcome of the transaction. If all participants voted to commit the transaction may commit. However if a least one participant voted to rollback the transaction is rolled back. This completes the first phase.
- In the second phase the transaction manager sends either commit or rollback messages to each participant.

The 2PC protocol guarantees the ACID properties despite any failures that may occur. Usually the transaction manager uses a log to record the progress of the 2PC protocol so that messages can be replayed during recovery.

One-phase-commit protocol

If there is only one participant in the transaction the transaction manager can use a one-phase-commit (1PC) protocol instead of the 2PC protocol which can be expensive in terms or the number of messages sent and the data that must be logged. The 1PC protocol essentially delegates the transaction completion to the single resource manager. Orbix supports this 1PC protocol which allows developers to make use of the Orbix transaction manager without suffering the overheads associated with the 2PC protocol. By making use of the OTS and JTA interfaces an application can be easily extended to support multiple resource managers within a transaction easily.

CHAPTER 2

OMG OTS and J2EE JTA Interfaces

The OMG OTS provides interfaces to manage the demarcation of transactions and the propagation of transaction contexts. The J2EE JTA interfaces provide an alternative means of transaction demarcation and integration with compliant resource managers such as databases and message queues.

In this chapter

This chapter discusses the following topics:

Transaction Interfaces

Purpose

The OMG OTS provides interfaces to manage the demarcation of transactions (creation and completion), the propagation of transaction contexts to the participants of the transaction and interfaces to allow applications to participate in the transaction.

The J2EE JTA interfaces provide an alternative means of transaction demarcation and integration with compliant resource managers such as databases and message queues.

Illustration of transaction interfaces

Figure 1 shows these areas of transaction management.

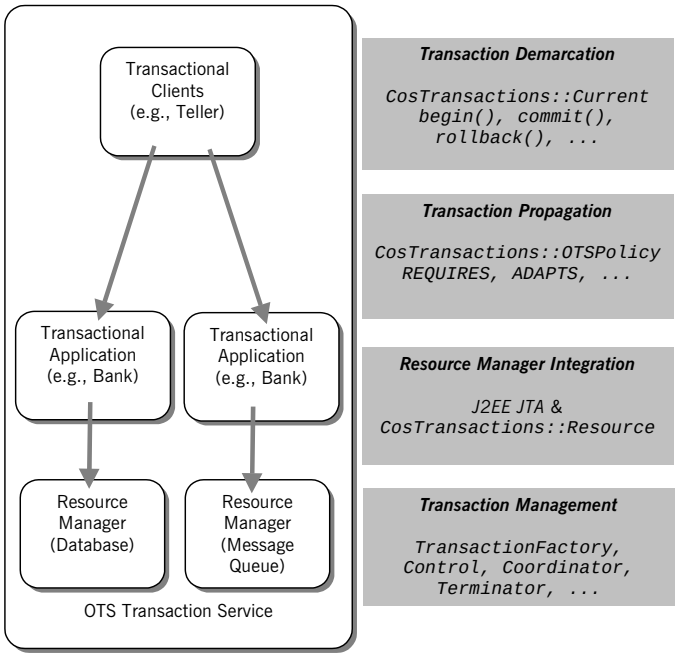

Figure 1: *OTS and JTA*

Transaction Demarcation

Transaction demarcation is where the application sets the boundaries of the transaction. Typically this is done using the OTS `Current` interface; invoking the `begin()` operation at the start of the transaction and either `commit()` or `rollback()` at the end of the transaction. The JTA interfaces may also be used to demarcate the transaction using either the UserTransaction or TransactionManager interfaces. An alternative to using the `Current` interface is to create transactions directly using the `TransactionFactory` interface and commit or rollback the transactions using the `Terminator` interface.

Transaction Propagation

Propagation refers to the passing of information related to the transaction to the application objects that are participants in the transaction. When the `Current` interface is used for transaction demarcation this propagation takes place transparently and is controlled by a number of POA policies. If the transaction is created using the JTA interfaces propagation is also done transparently. Transactions created using the `TransactionFactory` interface must be propagated by adding an extra parameter to the operation.

Resource Manager Integration

Integration with resource managers such as databases is done using the JTA interface. Alternatively an application may use the OTS `Resource` interface to provide integration with proprietary resource managers.

Transaction Management

The OTS interfaces also provide operations for general transaction management. These include, setting timeouts, registering resource objects and synchronization objects, comparing transactions and getting transaction names

OTS Interfaces

Supported OTS Interfaces

The following is a list of the main interfaces supported by the OTS. All interfaces are part of the IDL module CosTransactions. For more details on these interfaces, refer to the *CORBA Programmer's Reference*.

Table 1: *OTS Interfaces*

Interface	Purpose
Control	The return type of `TransactionFactory::create()`. It provides access to the two controllers of the transactions, the `Coordinator` and the `Terminator`.
Coordinator	Provides operations to register objects that participate in the transaction.
Current	A local interface that provides the concept of a transaction to the current thread of control. The `Current` interface supports a subset of the operations provided by the `Coordinator` and `Terminator` interfaces.
RecoveryCoordinator	Used in certain failure cases to complete the transaction completion protocol for a registered resource object.
Resource	Represents a recoverable participant in a transaction. Objects supporting this interface are registered with a transaction's coordinator, and are then invoked at key points in the transaction's completion.
SubtransactionAwareResource	Represents a participant that is aware of nested transactions. Nested transactions are not supported in this release.

Table 1: *OTS Interfaces*

Interface	Purpose
Synchronization	Represents a non-recoverable object allowing application specific operations to occur both before and after transaction completion.
Terminator	Provides a means of directly committing or rolling back a transaction.
TransactionalObject	This interface has been deprecated and replaced with transaction policies (see Chapter 5).
TransactionFactory	Provides a means of directly creating top-level transactions.

OTS Transaction Modes

When using the OTS interfaces for transaction demarcation and propagation, there are two modes of use:

Indirect/Implicit — In the indirect/implicit mode transaction are created, committed and rolled back using the `Current` interface. Propagation takes place automatically depending on the policies in the target object's POA.

Direct/Explicit — In the direct/explicit mode transactions are created using the `TransactionFactory` and committed or rolled back using the `Terminator` object. Propagation is done by adding a parameter (for example, the transaction's control object) to each IDL operation.

The preferred mode for most applications is the indirect/implicit mode. The direct/explicit provides more flexibility but is more difficult to manage (see "Direct Transaction Demarcation" on page 40 and "Explicit Propagation" on page 61) for more details.

J2EE JTA Interfaces

JTA Interfaces

The Java Transaction API (JTA) is a set of high-level interfaces for transaction management for J2EE based applications. It provides interfaces both for transaction demarcation and control and interfaces for integrating resource managers such as JDBC databases and message queues. The JTA interfaces are provided for use in CORBA applications.

Interfaces for transaction demarcation and control are provided in the javax.transaction package. The interfaces are:

Table 2: *JTA javax.transaction package interfaces*

Interface	Purpose
Status	Definitions of transaction status codes.
Synchronization	Interface to allow applications to be notified before and after a transaction completes.
Transaction	Represents a transaction and allows operations to be performed on the transaction.
TransactionManager	Provides transaction demarcation and control (intended for use in J2EE application servers).
UserTransaction	Provides transaction demarcation and control (intended for use by the client application).

Integration with resource managers

Integration with resource managers is provided by interfaces in the javax.transaction.xa package. Most applications do not deal directly with these interfaces; rather JTA compliant resource managers provide implementations of the interfaces that are called by the OTS during transaction completion. These interfaces are:

Table 3: *JTA javax.transaction.xa package interfaces*

Interface	Purpose
XAResource	A Java mapping of the X/Open XA interface.
Xid	A Java mapping of the X/Open XID structure.

Integration with JDBC

The JDBC 2.0 specification supports JTA though the use of the interface javax.sql.XADataSource. Integration between a JTA compliant JDBC driver and the OTS is supported by two classes in the package com.iona.datasource:

Table 4: *JTA com.iona.datasource package classes*

Interface	Purpose
IT_XADataSource	A wrapper around a resource manager's javax.sql.XADataSource object.
IT_NonXADataSource	A wrapper around a resource manager's javax.sql.DataSource object.

Applications must use wrap an instance of either the IT_XADataSource or IT_NonXADataSource around the resource managers's equivalent objects.

CHAPTER 3

Getting Started with Transactions

This chapter illustrates the Object Transaction Service (OTS) by way of an example application. It includes the basic steps needed to develop an application with the OTS.

In this chapter

This chapter discusses the following topics:

Application Overview

Funds transfer application

The example application is that of funds transfer between two bank accounts. Figure 2 shows the application. The client has a reference to two objects representing two accounts. The account objects are implemented directly on top of an JTA-compliant database and use JDBC to access the database. This example shows the source and destination accounts using different databases, however they could both be using the same database.

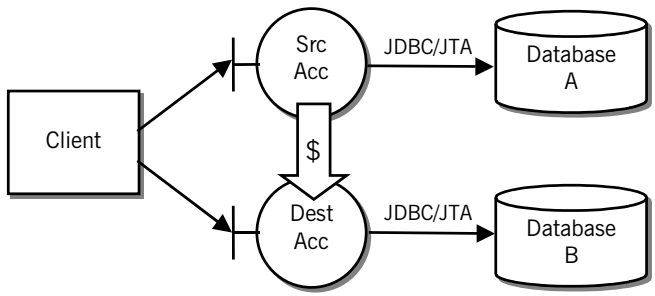

Figure 2: *Example OTS Application – Funds Transfer*

Interface definition

The interface for the account objects is defined in IDL as follows:

```
// IDL
module Bank
{
  typedef float CashAmount;
  interface Account
  {
    exception InsufficientFunds {};
    void deposit(in CashAmount amt);
    void withdraw(in CashAmount amt)
      raises (InsufficientFunds);
  };
  ...
};
```

TransactionalObject interface deprecated

Readers familiar with version 1.1 of the OTS specification (used by OrbixOTM and Orbix 3) will notice that the `Account` interface does not inherit from the `CosTransactions::TransactionalObject` interface. The use of that interface to mark objects as transactional has been deprecated in favor of using POA policies in version 1.2 of the specification. The `TransactionalObject` interface is still supported for backward compatibility with the OTS in OrbixOTM and Orbix 3. See "Use of the TransactionalObject Interface" on page 115 for more details.

Since the `TransactionalObject` interface is deprecated, application developers no longer have to change the IDL used by their applications when adding transactional capabilities.

Transferring funds

Given a source and destination account, the funds transfer is performed by invoking the `withdraw()` operation on the source account followed by invoking the `deposit()` operation on the destination account. The application will look something like the following:

```
// Java
Bank.Account src_acc = ...
Bank.Account dest_acc = ...
float amount = 100.0;
src_acc.withdraw(amount);
dest_acc.deposit(amount);
```

Completing the application

To make this a transactional application we need three more steps:

1. The funds transfer application needs to be wrapped in a transaction to ensure the ACID properties. This is covered in "Transaction Demarcation" on page 18.
2. The application must make sure the transaction is propagated to the two account objects during the invocations of the `deposit()` and `withdraw()` operations. This is covered in "Transaction Propagation and POA Policies" on page 21
3. The implementation of the account objects must be integrated with a JTA compliant database. This is covered in "JTA Resource Manager Integration" on page 23.

Transaction Demarcation

Demarcation using OTS current object

Transaction demarcation refers to setting the boundaries of the transaction. The simplest way to do this is to use the OTS current object. The following are the steps involved:

1. Obtain a reference to the OTS current object from the ORB.
2. Create a new transaction.
3. Perform the funds transfer.
4. Complete the transaction by either committing it or rolling it back.

More information on transaction demarcation including other ways of creating, committing and rolling back transactions is covered in Chapter 4.

Obtain a reference to the OTS current object from the ORB

The OTS current object supports the `CosTransactions::Current` interface and a reference to the object is obtained by calling the ORB operation `resolve_initial_references("TransactionCurrent")`.

The interfaces in the CosTransactions module are in the package org.omg.CosTransactions. Error handling has been omitted for clarity:

```
// Java
...
import org.omg.*;
import org.omg.CosTransactions.*;
...
public static void main(String[] args)
{
  ORB orb = ORB.init(args, null);

  Object obj =
    orb.resolve_initial_references("TransactionCurrent");
  Current tx_current = CurrentHelper.narrow(obj);
  ...
}
```

Create a new transaction

The next step is the creation of a new top-level transaction. This is done by invoking `begin()` on the OTS current object:

```
// Java
tx_current.begin();
```

If the `begin()` succeeds, a new transaction is associated with the current thread of control.

Perform the funds transfer

The funds transfer is the same as shown in the application overview. There are no changes for transaction management. The code is reproduced here for completeness:

```
// Java
Bank.Account src_acc = ...
Bank.Account dest_acc = ...
float amount = 100.0;
src_acc.withdraw(amount);
dest_acc.deposit(amount);
```

Complete the transaction by either committing it or rolling it back

Once the work has been done, we need to complete the transaction. Most of the time the application simply wants to attempt to commit the changes made: this is done by invoking the `commit()` operation on the OTS current object:

```
// Java
try {
  tx_current.commit(false)
} catch (TRANSACTION_ROLLEDBACK) {
  // Transaction has been rolled back.
}
```

The `commit()` operation only attempts to commit the transaction. It may happen that due to system failures or other reasons the transaction cannot be committed; in this case the `TRANSACTION_ROLLEDBACK` system exception is raised.

The parameter passed to `commit()` is a boolean specifying whether heuristics outcomes should be reported to the client (see "Heuristic Outcomes" on page 102 for details on heuristic outcomes). In this example we do not wait for heuristic outcomes.

If instead of attempting a commit the application wants to roll back the changes made, the operation `rollback()` is invoked on the OTS current object:

```
// Java
tx_current.rollback()
```

Transaction Propagation and POA Policies

Propagating the transaction

The funds transfer application invokes the `withdraw()` and `deposit()` operations within the context of a transaction associated with the current thread of control. However the transaction needs to be propagated to the target objects to ensure that any updates they make are done in the context of the application's transaction.

POA Policies

To ensure propagation of transaction contexts the target objects must be placed in a POA with specific OTS POA policies. In particular the POA must use one of the OTSPolicy values `REQUIRES` or `ADAPTS`. The following code shows the creation of a POA with the `REQUIRES` OTSPolicy and the activation of an account object in the POA.

```
// Java

ORB orb = ...

// Create a policy object for the REQURIES OTS Policy.
Any policy_val = orb.create_any();
OTSPolicyValueHelper.insert(policy_val, REQUIRES.value);
Policy tx_policy =
    orb.create_policy(OTS_POLICY_TYPE.value,
                      policy_val);

// Add OTS policy to policy list (just 1 policy in this case).
Policy[] policies = new Policy[1];
policies[0] = tx_policy;

// Get a reference to the root POA.
Object obj = orb.resolve_initial_references("RootPOA");
POA root_poa = POAHelper.narrow(obj);

// Create a new POA with the OTS Policy.
POA tx_poa = root_poa.create_POA("REQUIRES TX",
                                 root_poa.the_POAManager(),
                                           policies);
```

```
// Create object using the transactional POA. This example
// uses servant_to_reference() to create the object
//
// AccountImpl is the servant class implementing the
// IDL interface Account.
AccountImpl servant = new AccountImpl(...);
byte[] id = tx_poa.activate_object(servant);
obj = tx_poa.servant_to_reference(servant);
Account account = AccountHelper.narrow(obj);
```

OTSPolicy values

There are three OTSPolicy values: `REQURIES`, `ADAPTS` and `FORBIDS`. `REQUIRES` specifies that the object must be invoked within a transaction; `ADAPTS` allows the object to be invoked both within and without a transaction; `FORBIDS` specifies that the object must not be invoked within a transaction. See Chapter 5 for a full discussion of POA and client policies relating to transaction propagation. Support for the deprecated `TransactionalObject` interface is discussed in "Use of the TransactionalObject Interface" on page 115.

The `create_resource_manager()` operation is passed the resource manager's name, XA switch (xaosw is Oracle's XA switch), open-string and close string as well as flags that affect the behavior of the resource manager. It returns a reference to the ResourceManager object and a reference to the CurrentConnection object (as an out parameter).

JTA Resource Manager Integration

Process of using a JTA resource manager

Integrating an OTS or JTA created transaction with a JTA compliant resource manager involves two steps:

1. First, the datasource object provided by the resource manager (in this case a database) must be wrapped by an Orbix 2000 datasource object.
2. The normal JDBC code must take account of the wrapped datasource object and the OTS/JTA transaction.

Wrapping the DataSource

The JDBC drivers provide XA compliant datasource objects for use within a distributed transaction. However these object cannot be used directly; instead they must be wrapped by an instance of the com.iona.datasource.IT_XADataSource class. This ensures that database connections created through the wrapper datasource are fully integrated into the current JTA or OTS transaction.

For example, the following code shows how an database's datasource object is wrapped by the IT_XADataSource object.

```
// Java
DataSource db_ds = ...
ORB orb = ...
IT_XADataSource ds = new IT_XADataSource(orb);
ds.setXADataSource(db_ds, "");
```

Refer to your JDBC driver documentation for information on obtaining their XA compliant datasource object.

Using JDBC within a OTS/JTA transaction

The JDBC code used to read and write to the database is the same as for a normal application with the following exceptions:

1. Before each access to the database a connection must be obtained using the getConnection() operation on the "wrapper" DataSource object.

2. After the database access the connection must be closed.

3. The JDBC java.sql.Connection operations that control transaction such as commit(), setAutoCommit() and

rollback() cannot be used. Instead the equivalent OTS or JTA operations must be used.

The following shows how integration with a JTA compliant JDBC 2.0 database is achieved:

```
// Java (in class AccountImpl)

public void deposit(float amt)
{
  // The “wrapper” datasource object.
  DataSource ds = ...

  try {
    Connection con = ds.getConnection();
    Statement stmt = con.createStatement();

    // Get current balance.
    String sql = “SELECT BALANCE FROM ACCOUNTS” +
                 “ WHERE ACC_ID = ” + m_accId;
    ResultSet rs = stmt.executeQuery(sql);

    float balance = results.getFloat(“BALANCE”);
    // Update balance.
    balance += amt;

    sql = “UPDATE ACCOUNTS SET BALANCE = ” + balance +
          “ WHERE ACC_ID = ” + m_accId;

    stmt.executeUpdate(sql);

    stml.close();
    con.close();
  } catch (java.lang.Exception ex) {
    ...
  }
}
```

Application-Specific Resources

Resource interface operations

The `CosTransactions::Resource` interface provides a mechanism for applications to become involved in the commit and rollback protocol of a transaction. The `Resource` interface provides five operations that are called at key points during the commit or rollback protocols:

- `prepare()`
- `commit()`
- `rollback()`
- `commit_one_phase()`
- `forget()`

Implementing resource objects

An application implements a resource object that supports the `Resource` interface and registers an instance of the object with a transaction using the `register_resource()` operation provided by the `Coordinator` interface. Resource object implementations are responsible for cooperating with the OTS to ensure the ACID properties for the whole transaction. In particular resource objects must be able to recover from failures.

The implementation of resource objects is discussed in detail in Chapter 8.

Configuration Issues

Issues

Before an application using OTS can run there are a number of configuration issues. These are concerned with loading the appropriate plug-ins and setting up the client and server bindings to enable implicit propagation of transactions.

Loading the OTS plug-in

For server applications, the OTS plug-in must be loaded explicitly by including it in the `orb_plugins` configuration variable. For example:

```
orb_plugins = [ ..., “ots”];
```

The client and server bindings are controlled with the configuration variables `binding:client_binding_list` and `binding:server_binding_list` respectively. The settings for both variables need to take account of the OTS for potential bindings. For example, to be considered for the IIOP/GIOP and collocated-POA bindings the variables must be set as follows:

```
binding:client_binding_list = [“OTS+POA_Coloc”,
                               “OTS+GIOP+IIOP”,
                               “POA_Coloc”,
                               “GIOP+IIOP”];

binding:server_binding_list = [“OTS”, “”];
```

Other configuration variables can be used to alter the characteristics of your application. These are covered in Chapter 11.

CHAPTER 4

Transaction Demarcation and Control

The most convenient means of demarcating transactions is to use the OTS Current object. The JTA UserTransaction and TransactionManager interfaces provide similar functionality. Direct transaction demarcation using the TransactionFactory and Terminator interfaces provide more flexibility but is more difficult to manage.

In this chapter

This chapter discusses the following topics:

The OTS Current Object

Current Interface

The OTS `Current` object maintains associations between the current thread of control and transactions. The `Current` interface is defined as follows:

```
// IDL (in module CosTransactions)
local interface Current : CORBA::Current {

  void begin()
    raises(SubtransactionsUnavailable);

  void commit(in boolean report_heuristics)
    raises(NoTransaction, HeuristicMixed,
           HeuristicHazard);

  void rollback()
    raises(NoTransaction);

  void rollback_only()
    raises(NoTransaction);

  Status get_status();

  string get_transaction_name();

  void set_timeout(in unsigned long seconds);
  unsigned long get_timeout();

  Control get_control();

  Control suspend();

  void resume(in Control which)
    raises(InvalidControl);
};
```

Threads and transactions

The OTS `Current` object maintains the association between threads and transactions. This means the same OTS `Current` object can be used by several threads. Figure 3 shows the relationship between threads, the OTS `Current` object, and the three objects that represent a transaction (`Control`, `Coordinator` and `Terminator`).

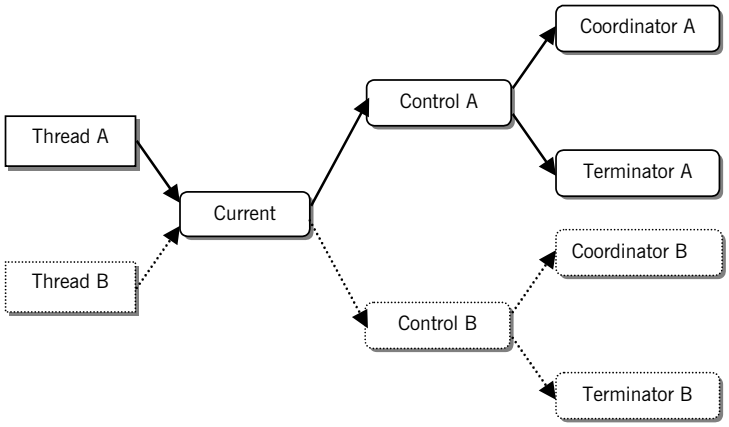

Figure 3: *Thread and Transaction Associations*

Getting a Reference to the OTS Current Object

A reference to the OTS `Current` object is obtained by calling `resolve_initial_references()` passing `"TransactionCurrent"` as the parameter and narrowing the result to `CosTransactions::Current`. For example:

```
// Java
Current tx_current;
try {
  ORB orb = ...
  Object obj =
    orb.resolve_initial_references("TransactionCurrent");

  tx_current = CurrentHelper.narrow(obj);
}
catch (SystemException ex)
{
  // Error handling.
  ...
}
```

The `Current` interface is declared as local which means references to the `Current` object cannot be passed as parameters to IDL operations or passed to operations such as `object_to_string()`.

Creating Transactions

The `begin()` operation is used to create a new transaction and associate the new transaction with the current thread of control. If there is no current transaction a top-level transaction is created; otherwise a nested transaction is created (see "Nested Transactions" on page 33).

The following code creates a new transaction:

```
// Java
Current tx_current = ...
try
{
  tx_current.begin();
}
catch (SubtransactionsUnavailable ex)
{
  // Already in a transaction and nested transaction are not
  // supported.
}
catch (SystemException ex)
{
  // Error handling...
}
```

Committing the Current Transaction

The `commit()` operation attempts to commit the current transaction, if any, and removes the current thread/transaction association. If the `commit()` operation returns normally the transaction was successfully committed. However if the `TRANSACTION_ROLLEDBACK` system exception is raised the transaction has been rolled back. In both cases the transaction is disassociated with the current thread of control.

For example, the following code attempts to commit the current transaction:

```
// Java
Current tx_current = ...
try
{
  // Attempt to commit the current transaction.
  tx_current.commit(false);
}
catch (TRANSACTION_ROLLEDBACK)
{
  // The transaction was rolled back.
}
catch (SystemException ex)
{
  // Error handling...
}
catch (NoTransaction)
{
  // There was no transaction to commit.
}
```

If there is no current transaction the `CosTransactions::NoTransaction` exception is raised.

The `commit()` operation takes a boolean parameter that indicates whether reporting of heuristic exceptions is permitted. Heuristic exceptions occur when a there is a conflict or potential conflict between the outcome decided by the transaction coordinator and the outcome performed by one or more resource managers (see "Heuristic Outcomes" on page 102 for more details). If a value of true is passed, the application must be prepared to catch the `HeuristicMixed` and `HeuristicHazard` exceptions; if a value of false is passed these exceptions are never raised.

Rolling Back the Current Transaction

The `rollback()` operation rolls back the current transaction, if any, and removes the current thread/transaction association. For example, the following code rolls back the current transaction:

```
// Java
Current tx_current = ...
try
{
  tx_current.rollback();
}
catch (SystemException ex)
{
  // Error handling...
}
catch (NoTransaction)
{
  // There was no transaction to commit.
}
```

If there is no current transaction the `CosTransactions::NoTransaction` exception is raised.

The `rollback_only()` operation may also be used to mark a transaction to be rolled back. This operation does not actively rollback the transaction, but instead prevents it from ever being committed. This can be useful, for example, to ensure the current transaction will be rolled back during a remote operation. Again, the `NoTransaction` exception is raised if there is no current transaction.

Nested Transactions

Nested transactions, also known as sub-transactions, provide a way of composing applications from a set of transactions each of which can fail independently of each other. Nested transactions form a hierarchy known as a transaction family. No updates are made permanent until the top-level transaction commits.

When using the `Current` object, a nested transaction is created by calling `begin()` when there is already a transaction associated with the current thread of control. When nested transaction is committed or rolled back, the thread transaction association reverts back to the parent transaction.

Note: Nested transactions are not supported in this release of Orbix.

Timeouts

The `set_timeout()` operation sets the timeout in seconds for subsequent top-level transactions. It does not set the timeout for the current transaction. Passing a value of 0 means subsequent top-level transactions will never timeout.

If `set_timeout()` is not called the default timeout is taken from the `plugins:ots:default_transaction_timeout` configuration variable.

The `get_timeout()` operation returns the current timeout in seconds for subsequent top-level transactions. It does not return the timeout for the current transaction.

For example, the following code sets the timeout for subsequent top level transactions to 30 seconds:

```
// Java
Current tx_current = ...
tx_current.set_timeout(30);
```

Suspending and Resuming Transactions

The `suspend()` operation temporarily removes the association between the current thread of control and the current transaction if any. Calling `suspend()` returns a reference to a control object for the transaction. The transaction can be resumed later by calling the `resume()` operation passing in the reference to the control object.

Suspending a transaction is useful if it is necessary to perform work outside of the current transaction. For example:

```
// Java
Current tx_current = ...
tx_current.begin();
account.deposit(100.0);

// Suspend the current transaction.
Control control = tx_current.suspend();

// Do some non-transactional work.
...

// Resume the transaction.
tx_current.resume(control);

tx_current.commit(true);
```

The `resume()` operation raises the `CosTransactions::InvalidControl` exception if the transaction represented by the control object cannot be resumed.

Sometimes the work done during the transaction's suspend state can be work on a different transaction. Thus, `suspend()` and `resume()` give you a way to work on multiple transactions within the same thread of control.

Miscellaneous Operations

The `get_status()` and `get_transaction_name()` operations provide information on the current transaction. The `get_control()` operations returns the `Control` object for the current transaction or nil if there is no current transaction. This is used to provide access to the `Coordinator` and `Terminator` objects for more advanced control. See Chapter 7 for more details

The JTA Interfaces

Use of UserTransaction and TransactionManager

The JTA interfaces UserTransaction and TransactionManager can be used as an alternative to the OTS Current object for transaction demarcation. The UserTransaction interface is for use within client applications while the TransactionManager interface provides some additional operations for use within server applications. This section deals only with the UserTransaction interface; full details on the remainder of the JTA interfaces are available in Chapter 6.

UserTransaction Interface

The UserTransaction interface is part of the javax.transaction package an is defined as follows:

```
public interface UserTransaction
{
  public abstract void begin()
    throws NotSupportedException, SystemException;

  public abstract void commit()
    throws RollbackException, HeuristicMixedException,
           HeuristicRollbackException, SecurityException,
           IllegalStateException, SystemException;

  public abstract int getStatus();
    throws SystemException;

  public abstract void rollback();
    throws IllegalStateException, SecurityException,
           SystemException;

  public abstract void setRollbackOnly();
    throws IllegalStateException, SystemException;

  public abstract void setTransactionTimeout(int seconds);
    throws SystemException;
}
```

All of the UserTransaction operations are supported in the TransactionManager interface.

Getting a Reference to the UserTransaction Object

A reference to the UserTransaction object is obtained by passing "UserTransaction" to resolve_initial_references() and casting the result to UserTransaction. For example:

```
// Java
...
import javax.transaction.*;
import org.omg.*;
...
public class BankTeller {
  public static void main(String[] args) {

    try {
      ORB orb = ORB.init(args, null)
      Object obj =
        orb.resolve_initial_reference("UserTransaction");
      UserTransaction utx =
        (UserTransaction) obj;
      ...
    } catch (Exception ex) {
      ...
    }
  }
}
```

Creating a Transaction

The operation begin() is used to create a new transaction and associate the transaction with the current thread of control. For example:

```
// Java
UserTransaction utx = ...
try {
  utx.begin();
} catch (NotSupportedException ex) {
  // Nested transaction not supported.
} catch (Exception ex) {
  ...
}
```

The NotSupportedException exception is raised if there is already a transaction associated with the current thread of control and nested transaction are not supported.

Committing the current Transaction

To attempt to commit the current transaction the operation commit() is used. If this operation returns successfully the transaction was committed. However if the RollbackException is raised if the transaction was rolled back. For example:

```
// Java
UserTransaction utx = ...
try {
  utx.commit();
} catch (RollbackException) {
  // Transaction has been rolled back.
} catch (Exception) {
  ...
}
```

The IllegalStateException exception is raised if there is no transaction associated with the current thread of control. The exception HeuristicMixedException and HeuristicRollbackException are raised if heuristic outcomes occurred.

Rolling Back the Current Transaction

To rollback the current transaction the operation rollback() is used. For example:

```
// Java
UserTransaction utx = ...
try {
  utx.rollback();
} catch (Exception) {
  ...
}
```

The IllegalStateException exception is raised if there is no transaction associated with the current thread of control.

Alternatively the opertation setRollbackOnly() may be used to mark the transaction to be rolled back without actively rolling back the transaction. Once this operation has been called the transaction cannot be committed.

The IllegalStateException exception is raised if there is no transaction associated with the current thread of control for both of these operations.

Timeouts

The operation setTransactionTimeout() can be used to set the timeout, in seconds, for subsequence transactions created using the begin() operation. For example the following code creates a new transaction with a timeout of 30 seconds:

```
// Java
UserTransaction utx = ...
try {
  utx.setTransactionTimeout(30);
  utx.begin();
} catch (Exception) {
  ...
}
```

Direct Transaction Demarcation

Using the transaction factory to create transactions

The alternative to using the OTS `Current` object or the JTA UserTransaction and TransactionManager interfaces is to use the transaction factory directly to create transactions.

Example

The following code shows the creation of a new top-level transaction:

```
// Java
//
// Get a reference to the transaction factory.
ORB orb = ...
Object obj =
  orb.resolve_initial_references("TransactionFactory");
TransactionFactory tx_factory =
  TransactionFactoryHelper.narrow(obj);

// Create a transaction with a timeout of 60 seconds.
Control control = tx_factory.create(60);
```

The first step is to obtain a reference to the transaction factory object. This is done by calling `resolve_initial_references()` passing a value of `"TransactionFactory"` and narrowing the result to `CosTransactions::TransactionFactory`.

The `create()` operation creates a new top-level transaction and returns a control object representing the new transaction. `create()` is passed the timeout in seconds for the transaction. A value of `0` means there is no timeout.

To complete a transaction created using the transaction factory, the terminator object is used. The terminator object is obtained by calling `get_terminator()` on the control object. The `Terminator` interface provides the `commit()` and `rollback()` operations. These are the same as the ones provided by the `Current` interface except they do not raise the `NoTransaction` exception.

Example of a commit

The following shows the attempted commit of a transaction using the direct approach:

```
// Java
//
try {
  Terminator term = control.get_terminator();
  term.commit(true);
} catch (TRANSACTION_ROLLEDBACK){
  // Transaction has been rolled back.
}
```

CHAPTER 5

Propagation and Transaction Policies

This chapter describes how to control transfer of the transaction to the target object using POA policies or explicitly.

In this chapter

This chapter discusses the following topics:

Implicit Propagation Policies

Implicit and Explicit Propagation

Propagation refers to the transfer of the transaction to the target object during an invocation.

For transactions created using the OTS Current object or the JTA UserTransaction and TransactionManager interfaces, propagation is implicit. That is, the application does not have to change the way the object is invoked in order for the transaction to be propagated. Implicit propagation is controlled using POA policies.

For transactions created directly via the TransactionFactory reference, explicit propagation must be used.

Policies for implicit propagation

For implicit propagation, there are two POA policies and one client policy that affect the behavior of invocations with respect to transactions.

The POA policies are:

- OTSPolicy
- InvocationPolicy

Both policies allow an object to set requirements on whether the object is invoked in the context of a transaction and transaction model being used.

The client OTS policy is:

- `NonTxTargetPolicy`

This alters the client's behavior when invoking on objects that do not permit transactions.

Note: These three policies replace the deprecated `TransactionPolicy` and the use of the deprecated `TransactionalObject` interface both of which are still supported in this release. See "Migrating from TransactionPolicies" on page 59 and "Use of the TransactionalObject Interface" on page 115 for more details.

Shared and Unshared Transactions

InvocationPolicy transaction models

The InvocationPolicy deals with the transaction model supported by the target object. There are two transaction models:

- shared
- unshared

Shared model

The shared model is the familiar end-to-end transaction where the client and the target object both share the same transaction. That is, an invocation on an object within a shared transaction is performed within the context of the transaction associated with the client.

Unshared model

An unshared transaction is used for asynchronous messaging where different transactions are used along the invocation path between the client and the target object. Here, the target object invocation is performed within the context of a different transaction than the transaction associated with the client. Hence, the client and target object does not share the same transaction. This model is required since with asynchronous messaging it is not guaranteed that the client and server are active at the same time.

Orbix does not support unshared transactions in this release. They are included in the following discussion for completeness only.

Policy Meanings

The three standard OTSPolicy values

The OTSPolicy has three possible standard values plus additional two values specific to Orbix. The Orbix-specific values are discussed in "Orbix-Specific OTSPolicies" on page 55; the standard values and their meanings are:

REQUIRES	This policy is used when the target object always expects to be invoked within the context of a transaction. If there is no transaction the `TRANSACTION_REQUIRED` system exception is raised. This policy guarantees that the target object is always invoked within a transaction.
FORBIDS	This policy is used when the target object does not permit invocations performed within the context of a transaction. If a transaction is present the `INVALID_TRANSACTION` system exception is raised. This policy guarantees that the target object is never invoked within a transaction. This is the default policy.
ADAPTS	This policy is used when the target object can accept both the presence and absence of a transaction. If the client is associated with a transaction, the target object is invoked in the context of the transaction; otherwise the target object is invoked without a transaction. This policy guarantees that the target object is invoked regardless of whether there is a transaction or not. Here, the target object adapts to the presence or not of a transaction.

Objects with the `REQUIRES` or `ADAPTS` OTSPolicy are also known as transactional objects since they support invocations within transactions; objects with the `FORBIDS` OTSPolicy or no OTSPolicy at all are known as non-transactional objects since they do not support invocations within transactions.

For an example of using an OTSPolicy see "Example Use of an OTSPolicy" on page 49 below.

The two NonTxTargetPolicy values

The default behavior for a client that invokes on an object within a transaction where the target object has the `FORBIDS` OTSPolicy (or where the object does not have any OTSPolicy, since `FORBIDS` is the default) is for the `INVALID_TRANSACTION` exception to be raised. This behavior can be altered with the `NonTxTargetPolicy`. The policy values and their meanings are:

PREVENT	The invocation is prevented from proceeding and the `INVALID_TRANSACTION` system exception is raised. This is the default behavior
PERMIT	The invocation proceeds but the target object is not invoked within the context of the transaction. This satisfies the target object's requirements and allows the client to make invocations on non-transactional objects within a transaction.

Setting the policies

As with all client policies, there are four ways in which they may be set:

1. Using configuration. For the `NonTxTargetPolicy` the variable to set is `policies:non_tx_target_policy`.
2. Set the policy on the ORB using the `CORBA::PolicyManager` interface.
3. Set the policy for the current invocation using the `CORBA::PolicyCurrent` interface.
4. Set the policy on the target object using the `CORBA::Object::_set_policy_overrides()` operation.

For more information on client policies see the chapter "Using Policies" in the *CORBA Programmer's Guide*. For an example of using a `NonTxTargetPolicy` see "Example Use of a NonTxTargetPolicy" on page 51 below.

Note that since the default OTSPolicy is `FORBIDS`, using the `PREVENT` `NonTxTargetPolicy` could result in previously working code becoming unworkable due to invocations been denied. The `PREVENT` policy should be used with care.

The three InvocationPolicy values

Finally, the choice of which transaction model (shared or unshared) that an object supports is done using the InvocationPolicy. This has three values:

SHARED	The target object supports only shared transactions. This is the default. An asynchronous invocation results in the `TRANSACTION_MODE` system exception being raised.
UNSHARED	The target object supports only unshared transactions. A synchronous invocation results in the `TRANSACTION_MODE` system exception begin raised.
EITHER	The target object supports both shared and unshared transactions.

Note that the `UNSHARED` and `EITHER` InvocationPolicies cannot be used in combination with the `FORBIDS` and `ADAPTS` OTSPolicies. Attempting to create a POA with these policy combinations results in the `PortableServer::InvalidPolicy` exception being raised.

Example Use of an OTSPolicy

Steps to create an object with an OTSPolicy

The following are the steps to create an object with a particular OTS policy:

1. Create a CORBA `Policy` object that represents the desired OTS policy. This is done by calling the ORB operation `create_policy()` passing in the value `CosTransactions::OTS_POLICY_VALUE` as the first parameter and the policy value (encoded as an `any`) as the second parameter.
2. Create a POA that includes the `OTSPolicy` in its policy list. This is done by calling `create_POA()`.
3. Create an object using the new POA.

Example

The following code sample shows an object being created in a POA that uses the `ADAPTS` OTSPolicy. For clarity, the POA is created off the root POA and only one new policy is added.

```
// Java
//
// Create CORBA policy object for ADAPTS OTSPolicy.

ORB orb = ...
Any tx_policy_value = orb.create_any();
OTSPolicyHelper.insert(tx_policy_value, ADAPTS.value);

// Create a POA using the transactional policy.
Policy[] policies = new Policy[1];
policies[0] = orb.create_policy(OTS_POLICY_TYPE.value,
                                tx_policy_value);

// Get a reference to the root POA.

Object obj = orb.resolve_initial_references("RootPOA");
POA root_poa = POAHelper.narrow(obj);

POA tx_poa = root_poa.create_POA("TX ADAPTS", null, policies);
```

```
// Create object using the transactional POA. This example
// uses servant_to_reference() to create the object

// AccountImpl is the servant class implementing the
// IDL interface Account.
AccountImpl servant = new AccountImpl(...);

ObjectId id = tx_poa.activate_object(servant);

obj = tx_poa.servant_to_reference(servant);
Account account = AccountHelper.narrow(obj);
```

Example Use of a NonTxTargetPolicy

Steps to use a NonTxTargetPolicy

The following are the steps for a client to use a `NonTxTargetPolicy` when invoking on a non-transactional object:

1. Get a reference to the `PolicyCurrent` or `PolicyManager` object.
2. Create a CORBA `Policy` object that represents the desired `NonTxTargetPolicy`. This is done by calling the `CORBA::ORB::create_policy()` operation passing in the value `CosTransactions::NON_TX_TARGET_POLICY_TYPE` as the first parameter and the policy value (encoded as an `any`) as the second parameter.
3. Call the `set_policy_overrides()` operation on the `PolicyCurrent` or `PolicyManager` object passing in a policy list containing the `NonTxTargetPolicy`. Alternatively call the `_set_policy_overrides()` operation on the target object itself.
4. Invoke on the non-transaction object (from within a transaction).

Example

The following code shows a client using the PERMIT NonTxTargetPolicy to invoke on a non-transactional object within a transaction. The client uses the PolicyCurrent object to set the policy. Assume that the Account object is using the REQUIRES or ADAPTS OTSPolicy and the AuditLog object is using the FORBIDS OTSPolicy or no OTSPolicy at all:

```
// Java
//
// Get reference to PolicyCurrent object.
ORB orb = ...
Object obj = orb.resolve_initial_references("PolicyCurrent");
PolicyCurrent policy_current =
  PolicyCurrentHelper.narrow(obj);

// Create PERMIT NonTxTarget policy.
Any tx_policy_value = orb.create_any();
NonTxTargetPolicyHelper.insert(tx_policy_value, PERMIT);
Policy[] policy_list = new Policy[1];
policy_list[0] = orb.create_policy(NON_TX_TARGET_POLICY_TYPE,
                                   tx_policy_value);
// Set policy overrides.
policy_current.set_policy_overrides(policy_list,
                                    SetOverrideType.ADD_OVERRIDE);
// Invoke on target object
Current tx_current = ...
Account account = ...
AuditLog log = ...

tx_current.begin();
account.deposit(100.00);
log.append("User ... deposited 100 to account ...");
tx_current.commit(true);
```

Specifying the default NonTxTargetPolicy

The default NonTxTargetPolicy value is taken from the policies:non_tx_target_policy configuration variable, which can be set to "prevent" and "permit" to represent the PREVENT and PERMIT policy values. If this configuration variable is not set, the default is PREVENT.

Use of the ADAPTS OTSPolicy

Using the ADAPTS OTSPolicy

The ADAPTS OTSPolicy is useful for implementing services that must work whether or not the client is using OTS transactions. If the client is using transactions, the target object simply executes in the same transaction context and its work will be either committed or rolled back when the client completes the transaction.

However, if there is no transaction the target object can choose to create a local transaction for the duration of the invocation.

Example

The following code shows how a servant might be implemented to take advantage of the ADAPTS OTSPolicy (error handling has been omitted):

```
// Java (in class AccountImpl)
public static void deposit(float amount)
{
  Current tx_current = ...

  // Test if a transaction was propagated from the client.
  Control control = tx_current.get_control();

  if (control == null)
  {
    // No current transaction, so create one.
    tx_current.begin();
  }

  // Do the transactional work
  ...

  // If a local transaction was created, commit it.
  if (control == null)
  {
    tx_current.commit(true);
  }
}
```

This approach allows clients to selectively bracket operations with transactions based on how much work is done. For example, if only a single server operation is performed then no client transaction needs to be created. However, if more than one operation is performed the client creates a transaction to ensure ACID properties for all of the operations.

For example (error handling omitted):

```
// Java
// Deposit money into a single account (no transaction
// needed).
Account acc = ...
acc.deposit(100.00);

// Transfer money between two account (this requires a
// transaction)
Account src_acc = ...
Account dest_acc = ...
Current tx_current = ...

tx_current.begin();
src_acc.withdraw(200.00);
dest_acc.deposit(200.00);
tx_current.commit(true);
```

For this example the servant created an OTS transaction. However, it could just create a local database transaction instead or not create any transaction at all.

Orbix-Specific OTSPolicies

The two proprietary OTSPolicy values

Orbix extends the set of OTSPolicies with two proprietary values to support automatically created transactions and optimizations. The values and their meanings are:

AUTOMATIC	This policy is used when the target object always expects to be invoked within the context of a transaction. If there is no transaction a transaction is created for the duration of the invocation. This policy guarantees that the target object is always invoked within a transaction. See "Automatic Transactions" on page 55below.
SERVER_SIDE	This policy is used in conjunction with just-in-time transaction creation to optimize the number of network messages in special cases. See "Just-In-Time Transaction Creation" on page 56 below.

Automatic Transactions

The `ADAPTS` OTSPolicy (see "Use of the ADAPTS OTSPolicy" on page 53) is useful for implementing servants that can be invoked both with and without transactions. A useful pattern to use is for the servant to check for the existence of a transaction and create one for the duration of the invocation if there is none. The `AUTOMATIC` OTSPolicy provides this functionality without having to code it into the servant implementation.

From the target object's point of view the `AUTOMATIC` OTSPolicy is the same as `REQUIRES` since the target object is always invoked in the context of a transaction. However, from the clients point of view, the `AUTOMATIC` policy is the same as `ADAPTS` since the client can choose whether to invoke on the object within a transaction or not. In fact, object references created in a POA with the `AUTOMATIC` OTSPolicy contain the `ADAPTS` policy so they can be used by other OTS implementations that do not support the `AUTOMATIC` OTSPolicy.

For the case were the client does not use a transaction and the automatically created transaction fails to commit, the standard `TRANSACTION_ROLLEDBACK` system exception is raised. Reporting of heuristic exceptions is not supported.

Just-In-Time Transaction Creation

Orbix provides three extensions to support the concept of just-in-time (JIT) transaction creation to eliminate network messages in special conditions. These extensions are:

1. A configuration option to enable JIT transaction creation, which allows the creation of a transaction to be delayed until it is really needed.
2. The `SERVER_SIDE OTSPolicy` which allows a transaction to be created just before a target object is invoked.
3. A additional operation `commit_on_completion_of_next_call()` that allows the next invocation on an object to also commit the transaction.

The use of JIT transaction creation is useful when invocations between a client and an object involve using a network connection. This is because it can reduce the number of network messages that are exchanged to create, propagate and commit a transaction.

Enabling JIT Transaction Creation

JIT transaction creation is enabled by setting the `plugins:ots:jit_transactions` configuration variable to `"true"`. When enabled a call to `Current::begin()` does not create a transaction; instead, it remembers that the client requested to create one. The client is said to be in the context of an empty transaction. At this stage a call to `Current::get_status()` would return `StatusActive` event though a real transaction has not been created. Likewise, calls to `Current::commit()` and `Current::rollback()` would succeed. A real transaction is only created at the following points:

1. When any of the following `CosTransactions::Current` operations are invoked: `rollback_only()`, `get_control()`, `get_transaction_name()` or `suspend()`.
2. When an object with any of the standard `OTSPolicies` is invoked.

If the target object's OTSPolicy is `SERVER_SIDE`, a real transaction is not created until the invocation has reached the object's POA. Note that unlike the `AUTOMATIC` OTSPolicy, this transaction it not terminated when the invocation has completed. Instead, the client adopts the newly created transaction.

When JIT transactions are not enabled, the SERVER_SIDE OTSPolicy behaves the same as the ADAPTS OTSPolicy, except that unlike the AUTOMATIC policy, other OTS implementations will not recognize the new policy.

A final optimization is possible when JIT transaction creation and the SERVER_SIDE OTSPolicy are used. The OTS current object in Orbix provides an additional operation that allows a transaction to be committed within the context of the target object rather than by the client:

```
// IDL
module IT_CosTransactions
{
  interface Current : CosTransactions::Current
  {
    void
    commit_on_completion_of_next_call()
      raises (CosTransactions::NoTransaction)
  };
};
```

The commit_on_completion_of_next_call() operation causes the current transaction to be committed after the completion of the next object invocation (so long as the target object is using the SERVER_SIDE OTSPolicy). The transaction commit is performed by the target object's POA, which means that the transaction will have been created and committed in the context of the target object rather than by the client.

To use the operation, the client must narrow the OTS current object to the IT_CosTransactions::Current interface (located in the com.iona.corba package).

```
// Java
Current tx_current = ...

IT_CosTransactions.Current it_tx_current =
  IT_CosTransactions.CurrentHelper.narrow(tx_current);

Account account = ...
it_tx_current.begin();

account.deposit(100.00);

it_tx_current.commit_on_completion_of_next_call();
account.deposit(50.00);

it_tx_current.commit(true);
```

Note that the client still must call the `commit()` operation, though this will not result in any network messages.

Migrating from TransactionPolicies

Mapping from TransactionPolicy values

Previous releases of Orbix used the deprecated `CosTransaction::TransactionPolicy` which provided seven standard policy values and two Orbix extensions. Below is a table that provides the mapping from `TransactionPolicy` values to their OTSPolicy and InvocationPolicy equivalent.

Table 5: *Mapping from TransactionPolicy values*

TransactionPolicy Value	OTSPolicy Value	InvocationPolicy Value
Allows_shared	ADAPTS	SHARED
Allows_none	FORBIDS	SHARED
Requires_shared	REQUIRES	SHARED
Allows_unshared	ADAPTS	Not supported
Allows_either	ADAPTS	Not supported
Requires_unshared	REQUIRES	UNSHARED
Requires_either	REQUIRES	EITHER or none
Automatic_shared	AUTOMATIC	SHARED
Server_side_shared	SERVER_SIDE	SHARED

Combining Policy Types

It is possible to create a POA that combines all three policy types to support interoperability with earlier versions of Orbix. However, invalid combinations result in the `PortableServer::InvalidPolicy` exception being raised when `PortableServer::POA::create_POA()` is called. An invalid combination is any combination not in Table 5; for example combining `Requires_shared` with `ADAPTS` and `SHARED`.

The mappings for the `Allows_unshared` and `Allows_either` `TransactionPolicies` are not supported since this would lead to an invalid combination of `OTSPolicies` and `InvocationPolicies`.

Note: Support for the TransactionPolicy type may be discontinued in a future Orbix release. It is recommended that only OTSPolicies and InvocationPolicies be used.

Explicit Propagation

Altering the IDL to propagate explicitly

When a transaction is created directly using the `TransactionFactory` interface the transaction must be propagated explicitly to target objects. This means altering the IDL for the application to add an extra parameter for the transaction's Control object.

Example

The following is the `Account` IDL interface modified to support explicit propagation:

```
// IDL (in module Bank)
#include <CosTransactions.idl>
...
interface Account
{
  exception InsufficientFunds {};

  void deposit(in CashAmount amt.
               in CosTransactions::Control ctrl);

  void withdraw(in CashAmount amt,
                in CosTransactions::Control ctrl)
    raises (InsufficientFunds);
};
```

Each invocation on the account object must now take a reference to a transaction control as its last parameter:

```
// Java
TransactionFactory tx_factory = ...
Control control = tx_factory.create(60);

Bank.Account src_acc = ...
Bank.Account dest_acc = ...
float amount = 100.0;
src_acc.withdraw(amount, control);
dest_acc.deposit(amount, control);

Terminator term = control.get_terminator();
term.commit(true);
```

It is also possible to pass a reference to the transaction's coordinator object instead of its control object.

CHAPTER 6

Using the Java Transaction API

This chapter describes the local Java interfaces that constitute the JTA package. These interfaces sit between a transaction manager on one hand and the application, application server and resource manager on the other.

In this chapter

This chapter contains the following sections:

JTA Features

What is JTA

The Java2 Platform, Enterprise Edition (J2EE) includes support for distributed transactions through the Java Transaction API (JTA) specification. The Java Transaction API (JTA) is a high level, implementation-independent, protocol-independent API that allows applications and application servers to manage transactions.

Features of the JTA

The JTA provides:

- An application level interface that allows for transaction boundary demarcation.
- An application server level interface that allows the application server to provide transaction demarcation, propagation, and resource management on behalf of an application.
- A Java mapping of the X/Open XA protocol to allow a resource manager to participate in a global transaction. It must do this by implementing a transactional resource interface, which will be used by the transaction manager to indicate transaction association, completion and recovery. A JDBC XADataSource is a typical transactional resource manager.

Figure 4 illustrates interaction between the Application, Application Server,Resource Manager and Transaction Manager components via JTA.

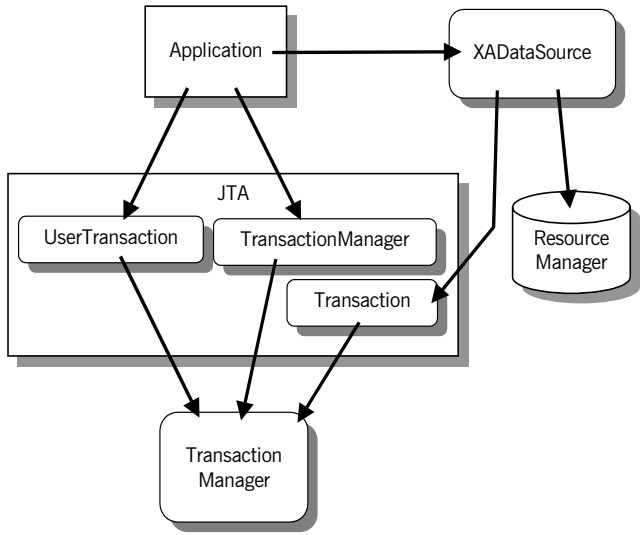

Figure 4: *Application and Resource Manager Interaction using JTA*

JTA API Overview

UserTransaction

The `javax.transaction.UserTransaction` interface is a client-side application-level interface that allows for transaction boundary demarcation. All global transactions created using this interface are associated with the calling thread. Transaction context propagation and thread-transaction association are managed by the transaction manager implementation underlying JTA and is transparent to the application. The underlying transaction manager for the Orbix implementation of JTA is OTS.

TransactionManager

The `javax.transaction.TransactionManager` interface is an application server-level interface that allows the application server to provide transaction demarcation, propagation, and resource management on behalf of an application. Each global transaction created via this interface will be associated with the calling thread and represented by a `javax.transaction.Transaction` object. By obtaining a reference to a particular transaction, the application can perform operations upon the represented transaction without regard to the calling thread. The `javax.transaction.TransactionManager` interface also provides a mechanism to disassociate a transaction from a calling thread and thereafter resume the association.

Transaction

Every global transaction that is created is associated with a `javax.transaction.Transaction` object. The `Transaction` interface provides functionality for enlisting resources into the global transaction, registering `Synchronization` objects and ending the transaction.

XAResource

The `javax.transaction.xa.XAResource` interface is a Java mapping of the X/Open XA protocol. It exists to allow interaction between a resource manager and transaction manager by associating a global transaction with a transactional resource.

Synchronization

Transaction synchronization allows the application to be notified by the transaction manager prior to and after completion of the global transaction. Specifically the `Synchronization` object is notified prior to the start of and after the 2PC protocol. The first notification will be within the context of the transaction being committed.

Managing Transactions and Resources

Overview

For a transaction manager to be able to coordinate work performed on behalf of a global transaction by a resource manager, a transactional resource must be enlisted into the global transaction and delisted prior to the end of the transaction.

Enlisting transactional resources

For each resource that will be used within the context of a transaction the `Transaction.enlistResource()` method must be invoked specifying the particular `XAResource` object. This allows the transaction manager to inform the resource manager to associate all work performed through that resource with the associated transaction. The transaction manager does this by invoking the `XAResource.start()` method.

It is the responsibility of the transaction manager to ensure that all resources representing the same resource manager are grouped accordingly. This must be done to ensure that the same resource manager is not asked by the transaction manager to commit on behalf of the same transaction more that once. The transaction manager can determine whether two `XAResource` objects represent the same resource manager by invoking the `XAResource.isSameRM()` method.

Delisting transactional resources

Transactional resources must be delisted prior to the end of a transaction; that is, before either commit or rollback is invoked. This is done via the `XAResource.delistResource()` method thus informing the resource manager to end the transaction–resource association.

Transaction manager interactions

The following sequence diagram shows the interaction between the transaction manager, resource manager and application.

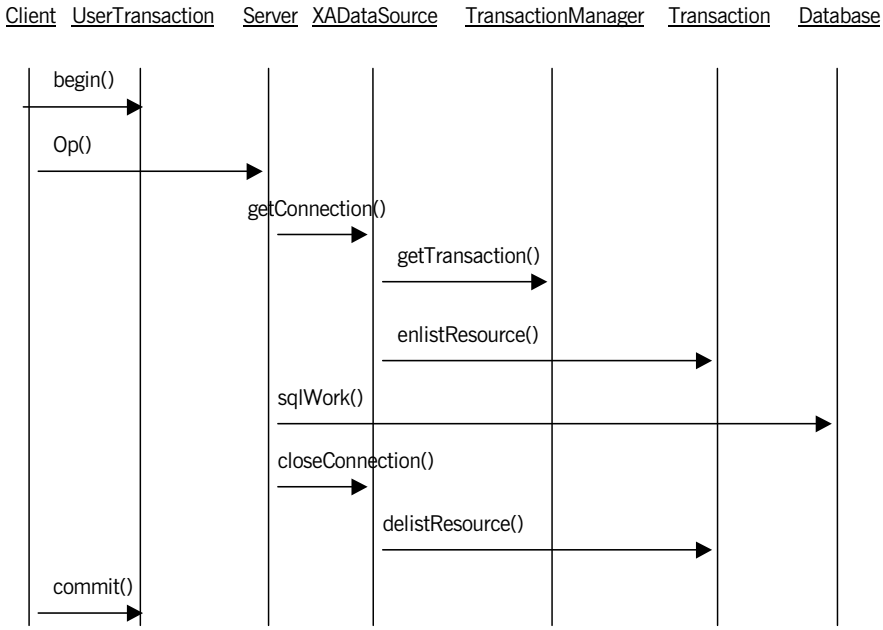

Figure 5: *A JTA Transaction*

DataSources

Overview

JDBC provides native database transactions through the JDBC Connection API. These are independent of JTA. If you want to use JTA to manage your own 2PC transactions, you must ensure that the datasources you use understand distributed transactions.

Managing transactional resources in Orbix

The Orbix solution for managing transactional resources; that is, an XAResource, within a global transaction is to provide two proprietary `java.sql.DataSource` implementations:

- `com.iona.datasource.IT_XADataSource` allows developers to manage javax.sql.XAConnection's within a global transaction.
- `com.iona.datasource.IT_NonXADataSource` allows standard java.sql.Connection's to become involved in a global transaction.

Both of these datasources are essentially wrappers around real datasources; for example, Cloudscape, Merant or Oracle datasources. Requesting a connection via either of these datasources in the `com.iona.datasource` package results in the return of a connection from the underlying datasource, which has been transparently associated with a JTA transaction.

The JDBC 2.0 XA specification dictates how this works for distributed transactions. In the non-distributed case a normal datasource can be used in the same way with one exception. That is that commit and rollback scenarios are initiated by the JTA rather than the application or, in other words, through the transaction and not the connection.

Example

The following example shows a business method on the server-side that accesses a database within the context of a propagated transaction:

Example 1: *Accessing a database through a propagated transaction*

```
// Java
//
public class Client
{
  public void updateDatabase(...)
  {
    javax.transaction.UserTransaction ut;
    DataAccessExampleImpl dataAccess;

1  // Lookup the UserTransaction reference
    org.omg.CORBA.Object obj =
      orb.resolve_initial_references("UserTransaction");
    ut = (javax.transaction.UserTransaction) obj;

2  // Lookup the DataAccessExample IOR
    dataAccess = ....;

3  // Begin a transaction.
    ut.begin();

4  // Perform some work on the server-side.
    dataAccess.accessDataSource(...);

5  // Commit the Transaction.
    ut.commit();
  }
}
```

The steps are:

1. Look up the `UserTransaction` reference.
2. Look up the IOR for the server-side IDL implementation object. The IOR must have been created from a POA with a transactional policy to allow the transaction to be propagated.
3. Begin a transaction.
4. Do some work on the server side. The transaction will be propagated with the request.
5. If no exceptions have been raised, commit the transaction.

Implementing a client with Orbix JTA and OTS

The Orbix JTA and OTS implementations are fully interoperable. Users can build a client that uses the `CosTransactions::Current` interface to control the creation of transactions, and a server that performs work on a database via connections from one of the datasources in the `com.iona.datasource` package.

Or users can build a client that uses the `javax.transaction.UserTransaction` interface to manage transactions, and a server that uses only interfaces from the `CosTransaction` module. The latter case will not be able to support implicitly enlisting database operations into the global transaction.

```
// Java
//
public class DataAccessExample
{
  public void accessDatasource(...)
  {
    javax.sql.DataSource ds;
    java.sql.Connection con;
    java.sql.Statement stmt;
    InitialContext initCtx = ...;

1 // Set up the IONA DataSource (see below)
    ...

2  ((com.iona.datasource.IT_XADataSource)ds).setOnePhase(true)

3 // Get connection from datasource.
    con = ds.getConnection();
    stmt = con.createStatement();

4 // Perform some work on the Database.
    stmt.executeQuery(...);
    stmt.executeUpdate(...);

5 // Close the connection.
    stmt.close();
    con.close();
  }
}
```

The steps are:

1. Setup the Orbix datasource wrapper.

2. Tell the datasource not to use XA if only a single resource is involved in the transaction.
3. Get a database connection from the datasource. The datasource will obtain a reference to the propagated transaction and transparently enlist the connection into the transaction. You can access the propagated transaction by calling `getTransaction()` on a `TransactionManager` reference.
4. Do some work on the database.
5. Close the connection. When the request returns, the client will commit the transaction and the changes will be committed to the database.

DataSource Configuration

Overview

The JTA plug-in and datasource implementations require some configuration information concerning your use of JNDI. (You should familiarize yourself with the concepts behind JNDI before continuing.)

Configuring JNDI

Within your application code you must set all system properties that are required by the specific JNDI implementation that you are using. For example, to use the Sun J2EE reference implementation you need two pieces of information:

- The location of the `InitialContext` factory that you want to use; and
- The location where the JNDI stores its binding information.

Thus your application code should contain the following lines of code:

```
// Java
String initialContextFactory =
  "com.sun.jndi.fscontext.RefFSContextFactory";
String providerURL =
  new File(System.getProperty("user.home")).toURL().toString();
System.setProperty(Context.INITIAL_CONTEXT_FACTORY,
                   initialContextFactory);
System.setProperty(Context.PROVIDER_URL,
                   providerURL);
```

Retrieving a reference to TransactionManager

The `com.iona.datasource` datasources must be able to retrieve a reference to the `TransactionManager` interface, which it will need to enlist its connections to any active transactions. It can do this in two ways:

- via JNDI; or
- `resolve_initial_references()`.

Retrieving a reference via JNDI

If the datasource is constructed using the null parameter constructor it will attempt to obtain the `TransactionManager` reference from JNDI. Therefore you must obtain a reference to the `TransactionManager` interface and register this with JNDI as shown below:

```
// Java
org.omg.CORBA.Object obj =
  orb.resolve_initial_references("TransactionManager");
TransactionManager tx_manager = (TransactionManager) obj;

Context jta_jndi_context = new InitialContext();
jta_jndi_context.bind("TransactionManager", tx_manager);
```

Note that the reference must be bound under the name `"TransactionManager"`.

Retrieving a reference using resolve_initial_references()

If you create the datasource using the constructor that takes an ORB reference as a parameter, it will obtain a `TransactionManager` reference via `resolve_initial_references()`.

The following code example demonstrates the preparatory work required to include your `XADataSource` in a JTA transaction:

```
// Java

// Set the JNDI System Properties...
// See above...

// Create a JNDI Initial Context ...
InitialContext initCtx = new InitialContext();

// Lookup the underlying datasource i.e. Merant, Oracle
javax.sql.XADataSource underlying_ds =
  (javax.sql.XADataSource) initCtx.lookup("jdbcXADataSource");

// To enable the underlying datasource to become
// involved in a transaction it must be wrapped in the
// supplied Orbix datasource via the setXADataSource()
// method. Also to enable recovery you must also supply
// the full path name with which the underlying datasource
// is registered through JNDI.
```

```
com.iona.datasource.IT_XADataSource xa_ds =
   new com.iona.datasource.IT_XADataSource();

// Note that you need only specify a jndi name
// for you datasource for 2PC scenarios, otherwise
// a null string will suffice. Binding the Orbix
// datasource wrapper with jndi also becomes
// irrelevant in a 1PC scenario.
xa_ds.setXADataSource(underlying_ds, "jdbcXADataSource");
initCtx.bind("ionaDataSource", xa_ds);

// Should the application require only a single resource
// to be registered for any given transaction or should
// recovery not be required, we can optimize performance
// of the Orbix datasource by informing it that XA
// is not required.
xa_ds.setOnePhase(true);
```

Using a standard DataSource

If you do not want to use an `XADataSource`, it is also possible to include a standard `DataSource` in a JTA transaction by using the Orbix supplied `com.iona.datasource.IT_NonXADataSource` class. As before you must set up a JNDI context, but then the code becomes:

```
// Java
//
// Lookup the underlying non xa datasource, e.g., Merant,
// Oracle.

javax.sql.DataSource underlying_ds =  (javax.sql.DataSource)
initCtx.lookup("jdbcDataSource");

com.iona.datasource.IT_NonXADataSource non_xa_ds =
   new com.iona.datasource.IT_NonXADataSource();
non_xa_ds.setDataSource(underlying_ds, "jdbcDataSource");

initCtx.bind("ionaDataSource", non_xa_ds);
```

Now that you have a data source registered with JNDI under the name `"ionaDataSource"` or `"ionaXADataSource"`, it is this data source that your application must retrieve. It is from this data source that it will acquire connections to the database, upon which it will perform work. Each connection will be associated with the transaction that associated with the current thread, making it transparent to the user.

JTA Configuration

Overview

Some configuration issues that must be addressed before an application can use either one of the JTA plug-ins. Before proceeding you should read Chapter 11, because the JTA plug-in uses the OTS plug-in.

Specifying plug-ins

The `orb_plugins` configuration variable should specify the JTA plug-ins. This is required for either plug-in to successfully register itself with JNDI. For example:

```
orb_plugins = [..., "jta_manager"]
```

or

```
orb_plugins = [..., "jta_user"]
```

Registering a persistent POA

The `com.iona.transaction.manager` plug-in can support recovery when using an `XADataSource` and when configured to use a 2PC transaction service; for example, the Encina OTS transaction manager. To allow recovery it uses a persistent POA, the name of which must be registered in conjunction with the ORB name in the locator. The default behavior is that a transient POA named `"iJTA"` is created by the plug-in (simply as a namespace) and a persistent POA named `"resource"` is created from the namespace POA. For example, using the default names with an ORB called `JTA_Manager`, you should execute the following commands:

```
itadmin orbname create JTA_Manager
itadmin poa create -transient iJTA
itadmin poa create -orbname JTA_Manager iJTA/resource
```

The POA names are configurable through the following variables:

```
plugins:jta:poa_namespace     = "iJTA";
plugins:jta:resource_poa_name = "resource";
```

This persistent POA is only required if you have configured the JTA to enable recovery, which is done by setting the following configuration variable:

```
plugins:jta:enable_recovery = "true";
```

CHAPTER 7

Transaction Management

This chapter covers some additional areas of transaction management. This includes Synchronization objects, transaction identity and status operations, relationships between transactions and recreating transactions.

In this chapter

This chapter discusses the following topics:

Synchronization Objects

Synchronization interface

The transaction service provides a `Synchronization` interface to allow an object to be notified before the start of a transaction's completion and after it is finished. This is useful, for example, for applications integrated with a JTA compliant resource manager where the data is cached inside the application. By registering a synchronization object with the transaction the cache can be flushed to the resource manager before the transaction starts to commit. Without the synchronization object any updates made by the application could not be moved from the cache to the resource manager. The `Synchronization` interface is as follows:

```
// IDL (in module CosTransactions)
interface Synchronization : CosTransactions::TransactionalObject
   {
  void before_completion();

  void (in Status s);
};
```

before_completion()

This operation is invoked during the commit protocol before any 2PC or 1PC operations have been called, that is before any JTA or Resource prepare operations.

An implementation may flush all modified data to the resource manager to ensure that when the commit protocol begins, the data in the resource is up to date.

Raising a system exception causes the transaction to be rolled back as does invoking the `rollback_only()` operation on the `Current` or `Coordinator` interfaces.

The `before_completion()` operation is only called if the transaction is to be committed. If the transaction is being rolled back for any reason this operation is not called.

after_completion()

This operation is invoked after the transaction has completed, that is after all JTA or Resource commit or rollback operations have been called. The operation is passed the status of the transaction so it is possible to

determine the outcome. It is possible that `before_completion()` has not been called, so the implementation must be able to deal with this possibility.

An implementation can use this operation to release locks that were held on behalf of the transaction or to clean up caches. Raising an exception in this operation has no effect on the outcome of the transaction as this has already been determined. All system exceptions are silently ignored.

---

register_synchronization()

A synchronization object is registered with a transaction by calling the `register_synchronization()` operation on the transaction's coordinator. Assuming the `SynchronizationImpl` class supports the `Synchronization` interface the following code may be used:

```
// Java
//
// Get the control and coordinator object for the
// current transaction.
//
Current tx_current = ...
Control control = tx_current.get_control();
Coordinator coordinator = control.get_coordinator();


//
// Create a synchronization servant and activate it in a
// transactional POA. The OTS Policy should be ADAPTS.
//
SynchronizationImpl servant = new SynchronizationImpl();
POA allows_shared_poa = ...
Synchronization obj = servant.activate(poa);

//
// Register the synchronization once with the transaction.
//
coordinator.register_synchronization(sync);
```

The `register_synchronization()` operation raises the `Inactive` exception if the transaction has started completion or has already been prepared. A synchronization object must only be registered once per transaction, this is the application's responsibility.

Note: Unlike resource objects, synchronization objects are not recoverable. The transaction service does not guarantee that either operation on the interface will be called in the event of a failure. It is imperative that applications use a resource object if they need guarantees in these situations (to release persistent locks for example).

Transaction Identity Operations

Coordinator interface identity operations

The `Coordinator` interface provides a number of operations related to the identify of transactions. Some of these operations are also available in the `Current` interface:

```
// IDL (in module CosTransactions)
interface Coordinator {
  boolean is_same_transaction(in Coordinator tc);
  unsigned long hash_transaction();
  unsigned long hash_top_level_tran();
  string get_transaction_name();
  PropagationContext get_txcontext();
  ...
};
```

Table 6: *Coordinator interface identity operations*

Operation	Description
is_same_transaction()	Takes a transaction coordinator as a parameters and returns true if both coordinator objects represent the same transaction; otherwise returns false.
hash_transaction()	Returns a hash code for the transaction represented by the target coordinator obejct. Hash codes are uniformly distributed over the range of a CORBA unsigned long and are not guaranteed to be unique for each transaction.
get_transaction_name()	Returns a string representation of the transaction's identify. This string is not guaranteed to be unique for each transaction so it is only useful for display and debugging purposes. This operation is also available on the `Current` interface.

Table 6: *Coordinator interface identity operations*

Operation	Description
get_txcontext()	Returns the PropagationContext structure for the transaction represented by the target coordinator object. Amongst other information, the PropagationContext structure contains the transaction identifier in the current.otid field. See "Recreating Transactions" on page 89 for more information on the structure of the PropagationContext.

Maintaining information in individual transactions

The `is_same_transaction()` and `hash_transaction()` operations are useful when it is necessary for an application to maintain data on a per transaction basis (for example, for keeping track of whether a particular transaction has visited the application before to determine whether a Resource or Synchronization object needs to be registered). The `hash_transation()` operation can be used to implement an efficient hash table while the `is_same_transaction()` operation can be used for comparison within the hash table.

For nested transaction families the `hash_top_level_transaction()` is provided. This returns the hash code for the top level transaction.

Transaction Status

Coordinator interface status operations

The `Coordinator::get_status()` operation returns the current status of a transaction. This operation is also provided by `Current::get_status()` for the current transaction. The status returned may be one of the following values:

StatusActive	The transaction is active. This is the case after the transaction has started and before the transaction has started to be committed or rolled back.
StatusCommitted	The transaction has successfully completed its commit protocol.
StatusCommitting	The transaction is in the process of committing.
StatusMarkedRollback	The transaction has been marked to be rolled back.
StatusNoTransaction	There is no transaction. This can only be returned from the `Current::get_status()` operation and occurs when there is no transaction associated with the current thread of control.
StatusPrepared	The transaction has completed the first phase of the 2PC protocol.
StatusPreparing	The transaction is in the process of the first phase of the 2PC protocol.
StatusRolledBack	The transaction has completed rolling back.
StatusRollingBack	The transaction is in the process of being rolled back.
StatusUnknown	The exact status of the transaction is unknown at this point.

The following code shows how to obtain the status of a transaction from the transaction's coordinator object:

```
// Java
Coordinator coord = ...
Status status = coord.get_status();
if (status == StatusActive)
{
  ...
} else if (status == StatusRollingBack)
{
  ...
} else if ...
```

There are two additional status operations for use within nested transaction families:

- `get_top_level_status()` returns the status of the top-level transaction.
- `get_parent_status()` returns the status of a transaction's parent.

Transaction Relationships

Coordinator interface relationship operations

The `Coordinator` interface provides several operations to test the relationship between transactions. Each operation takes as a parameter a reference to another transaction's coordinator object:

```
// IDL (in module CosTransactions)
interface Coordinator {
  boolean is_same_transaction(in Coordinator tc);
  boolean is_related_transaction(in Coordinator tc);
  boolean is_ancestor_transaction(in Coordinator tc);
  boolean is_descendant_transaction(in Coordinator tc);
  boolean is_top_level_transaction();
  ...
};
```

Table 7: *Coordinator interface relationship operations*

Operation	Description
is_same_transaction()	returns true if both coordinator objects represent the same transaction; otherwise returns false.
is_related_transaction()	returns true if both coordinator objects represent transactions in the same nested transaction family; otherwise returns false.
is_ancestor_transaction()	returns true if the transaction represented by the target coordinator object is an ancestor of the transaction represented by the coordinator parameter; otherwise returns false. A transaction is an ancestor to itself and a parent transaction is an ancestor to its child transactions.

Table 7: *Coordinator interface relationship operations*

Operation	Description
is_descendant_transaction()	Returns true if the transaction represented by the target coordinator object is a descendant of the transaction represented by the coordinator parameter; otherwise returns false. A transaction is a descendant of itself and is a descendent of its parent.
is_top_level_transaction()	Returns true if the transaction represented by the target coordinator object is a top-level transaction; otherwise returns false.

Example

The following code tests if the transaction represented by the coordinator c1 is an ancestor of the transaction represented by the coordinator c2:

```
// Java
Coordinator c1 = ...
Coordinator c2 = ...
if (c1.is_ancestor_transaction(c2))
{
  // c1 is an ancestor of c2
}
else
{
  // c1 is not an ancestor of c2
}
```

Recreating Transactions

TransactionFactory interface

The `TransactionFactory` interface provides the `create()` operation for creating new top-level transactions. The interface also provides a `recreate()` operation to import an existing transaction into the local context. The `recreate()` is passed a `PropagationContext` structure and returns a `Control` object representing the recreated transaction. The interfaces and types are declared as follows:

```
// IDL (in module CosTransactions)
struct otid_t {
  long formatID;
  long bqual_length;
  sequence <octet> tid;
};

struct TransIdentity {
  Coordinator coord;
  Terminator term;
  otid_t otid;
};

struct PropagationContext {
  unsigned long timeout;
  TransIdentity current;
  sequence <TransIdentity> parents;
  any implementation_specific_data;
};

interface TransactionFactory
{
  Control recreate(in PropagationContext ctx);
  ...
};

interface Coordinator
{
  PropagationContext get_txcontext();
    raises (Unavailable);
  ...
};
```

The `PropagationContext` is a structure that encodes sufficient information about the transaction to successfully recreate it. To get the `PropagationContext` for a transaction use the `get_txcontext()` operation provided by the `Coordinator` interface.

Example

The following code shows how to use the `get_txcontext()` and `recreate()` operations to explicitly import a transaction given a reference to the Control object for a foreign transaction:

```
// Java
Control foreign_control = ...
Coordinator foreign_coord =
  foreign_control.get_coordinator();
PropagationContext ctx = foreign_coord.get_txcontext();

TransactionFactory tx_factory = ...
Control control = tx_factory.recreate(ctx);
```

The PropagationContext structure contains the transaction's global identifier in the current.otid field. This is essentially a sequence of octets divided into two parts: a global transaction identifier and a branch qualifier. This structure is indented to match the XID transaction identifier format for the X/Open XA specification.

CHAPTER 8

Writing Recoverable Resources

The OTS supports resource objects to allow applications to participate in transactions. For example, an application might maintain some data for which ACID properties are required. This chapter describes the CosTransactions::Resource interface; how resource objects participate in the transaction protocols and the requirements for implementing resource objects.

In this chapter

This chapter discusses the following topics:

The Resource Interface

Resource interface transaction operations

The `CosTransactions::Resource` interface provides a means for applications to participate in an OTS transaction. The interface is defined as follows:

```
// IDL (in module CosTransactions)
interface Resource
{
    void commit_one_phase()
      raises (HeuristicHazard);

    Vote prepare()
      raises (HeuristicMixed,
              HeuristicHazard);

    void rollback()
      raises (HeuristicCommit,
              HeuristicMixed,
              HeuristicHazard);

    void commit()
      raises (NotPrepared,
              HeuristicRollback,
              HeuristicMixed,
              HeuristicHazard);

    void forget();
};
```

Resource object implementations cooperate with the OTS, through these five operations, to ensure the ACID properties are satisfied for the whole transaction. Each resource object represents a single participant in a transaction and throughout the lifecycle of the resource it must respond to the invocations by the OTS until the resource object is no longer needed. This may include surviving the failure of the process or node hosting the resource object or the failure of the process or node hosting the OTS implementation.

Overview of the use of resource objects

Figure 6 shows a high level picture of how clients, applications, the OTS and resource objects interoperate to achieve the ACID properties.

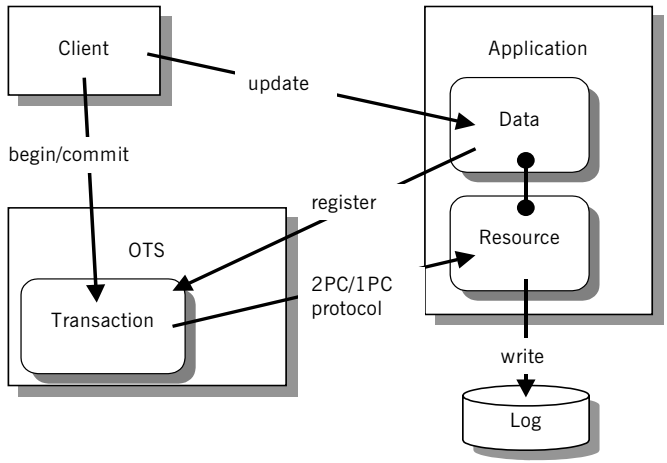

Figure 6: *Relationship between resources and transactions*

The steps involved are:

1. The client contacts the OTS implementation and creates a transaction.
2. The client makes invocations on the application within the context of the transaction and updates some data.
3. The application detects that the data is being updated and creates a resource object. The resource object is registered with the transaction.
4. The client completes by contacting the OTS implementation and attempting to commit the transaction.
5. The transaction initiates the commit protocol. The choice of which protocol to use (either 1PC or 2PC) depends on the number of resource objects registered with the transaction and whether the OTS supports the 1PC optimization.

6. Assuming the 2PC protocol is being used, the OTS sends a prepare message to the resource. The resource stably stores enough information to recover in case of a crash (for example, by writing the changes to a log file). The resource object votes to commit the transaction.
7. The OTS gathers the votes of all resource objects and decides the outcome of the transaction. This decision is send to all registered resource objects.
8. The resource object upon receiving the commit or rollback message makes the necessary changes and saves the decision to the log.
9. The OTS returns the outcome to the client.

Creating and Registering Resource Objects

Implementing servants for resource objects

Implementing servants for resource objects is similar to any servant implementation. The resource servant class needs to inherit from the `POA_CosTransactions::Resource` class to extend the `ResourcePOA` class and provide implementations for the five resource operations. For example, the following class can be used to implement a resource servant:

```
// Java
public class ResourceImpl extends ResourcePOA
{
    public ResourceImpl() { ... }

    public Vote
    prepare()
    throws HeuristicMixed, HeuristicHazard
    { ... }

    public void
    rollback()
    throws HeuristicCommit, HeuristicMixed, HeuristicHazard
    { ... }

    public void
    commit()
    throws NotPrepared, HeuristicRollback, HeuristicMixed,
           HeuristicHazard
    { ... }

    public void
    commit_one_phase()
    throws HeuristicHazard
    { ... }

    public void
    forget()
    { ... }
}
```

Creating resource objects

Resource objects, once prepared, must survive failures until the 2PC protocol has completed. During recovery any resource objects requiring completion must be recreated using the same identifier so the transaction coordinator can deliver the outcome. This means that resource objects must be created within a POA with a `PERSISTENT` lifespan policy and a `USER_ID` ID assignment policy. See the sections “Setting Object Lifespan” and “Assigning Object IDs” in the chapter "Managing Server Objects" in the *CORBA Programmer’s Guide* for more details.

Tracking resource objects

Each resource object can only be used once and may only be registered with one transaction. It is up to the application to keep track of whether it has seen a particular transaction before. This can be done efficiently using the `hash_transaction()` and `is_same_transaction()` operations provided by the `Coordinator` interface to implement a hash map (see “Transaction Identity Operations” on page 83 for details).

Some form of unique identifier must be used for the resource object’s `ObjectId`. One possibility is to use the transaction identifier (obtained from the `otid` field in the transaction’s propagation context).

Registering resource objects

Registration of a resource object with a transaction is done by the `register_resource()` operation provided by the transaction’s coordinator object. For example, the following code sample shows a resource servant and object being created and registered with a transaction:

```
// Java
Current tx_current = ...

// Get the transaction's coordinator object.
Control control = tx_current.get_control();
Coordinator coord = control.get_coordinator();

// Create resource servant.
ResourceImpl servant = new ResourceImpl();
```

```
// Create resource object. The POA referenced by resource_poa
// has the PERSISTENT lifespan policy and the USER_ID ID
// assignment policy.
POA resource_poa = ...
ObjectId oid = ...

resource_poa.activate_object_with_id(oid, servant);

Object obj = resource_poa.servant_to_reference(servant);

Resource resource = ResourceHelper.narrow(obj);

// Register the resource with the transaction coordinator.
RecoveryCoordinator rec_coord =
    coord.register_resource(resource);
```

The `register_resource()` operation returns a reference to a recovery coordinator object:

```
// IDL (in module CosTransactions)
interface Coordinator
{
  RecoveryCoordinator register_resource(in Resource r)
    raises(Inactive);
  ...
};

interface RecoveryCoordinator
{
  Status replay_completion(in Resource r)
    raises(NotPrepared);
};
```

The recovery coordinator object supports a single operation, `replay_completion()`, that is used for certain failure scenarios (see "Failure of the Transaction Coordinator" on page 105). Resource objects must hold onto the recovery coordinator reference.

The `register_resource()` operation raises the `Inactive` exception if the transaction is no longer active.

Resource Protocols

Protocols supported by resource objects

Resource object implementations cooperate with the transaction coordinator to achieve the ACID properties. This section examines the protocols that resource objects are required to support:

- Rolling back a transaction.
- The 2-phase-commit protocol.
- Read-only resources.
- The 1-phase-commit protocol.
- Heuristic outcomes.
- Failure and recovery

Transaction Rollbacks

Up until the time the coordinator makes the decision to commit a transaction, the transaction may be rolled back for a number of reasons. These include:

- A client calling the `rollback()` operation.
- Attempting to commit the transaction after the transaction has been marked to be rolled-back with the `rollback_only()` operation.
- The transaction being timed-out.
- The failure of any participant in the transaction.

When the transaction is rolled-back all registered resource are rolled-back via the `rollback()` operation. Figure 7 shows a transaction with two registered resource objects being rolled back after a timeout.

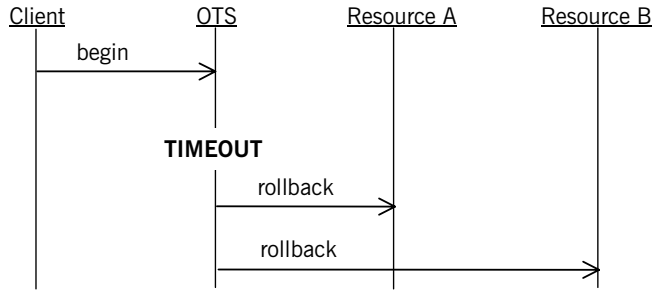

Figure 7: *Rollback after a timeout*

Rollbacks may also occur during the 2PC protocol (see below).

The 2-Phase-Commit Protocol

The 2-phase-commit (2PC) protocol is designed so that all participants within a transaction know the final outcome of the transaction. The final outcome is decided by the transaction coordinator but each resource object participating can influence this decision.

During the first phase, the transaction coordinator invokes the `prepare()` operation on each resource asking it to prepare to commit the transaction. Each resource object returns a vote which may be one of three possible values: VoteCommit indicates the resource is prepared to commit its part of the transaction; VoteRollback indicates the transaction must be rolled-back; and VoteReadOnly indicates the resource is no longer interested in the outcome of the transaction (see "Read-Only Resources" on page 100).

The coordinator makes a decision on whether to commit or rollback the transaction based on the votes of the resource objects. Once a decision has been reached the second phase commences where the resource objects are informed of the transaction outcome.

In order for the coordinator to decide to commit the transaction, each resource object must have either voted to commit the transaction or indicated that it is no longer interested in the outcome. Once a resource has voted to commit, it must wait for the outcome to be delivered via either the `commit()` or `rollback()` operation. The resource must also survive failures.

This means that sufficient information must be stable stored so that during recovery the resource object and its associated state can be reconstructed. Figure 8 shows a successful 2PC protocol with two resources objects. Both resources return VoteCommit from the `prepare()` operation and the coordinator decides to commit the transaction resulting in the `commit()` operations being invoked on the resources.

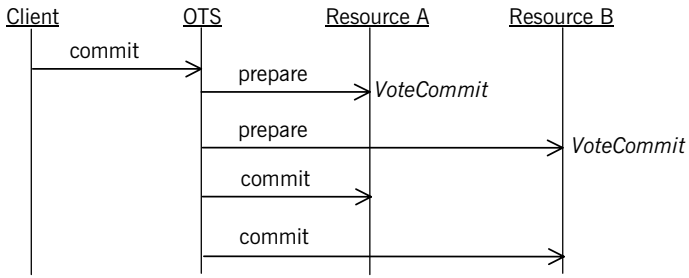

Figure 8: *Successful 2PC protocol with two resources*

If one resource returns VoteRollback the whole transaction is rolled back. Resources which have already been prepared and which voted to commit and resources which have not yet been prepared are told to rollback via the `rollback()` operation. Figure 9 shows VoteRollback being returned by one resource which results in the other resource being told to rollback.

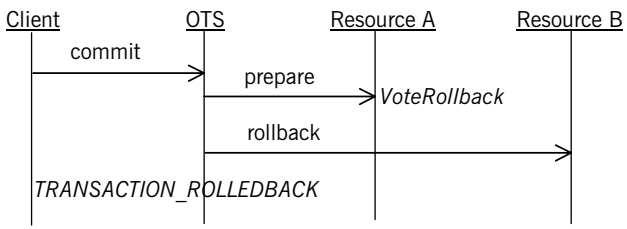

Figure 9: *Voting to rollback the transaction.*

Read-Only Resources

A resource can return VoteReadOnly from the `prepare()` operation which means the resource is no longer interested in the outcome of the transaction. This is useful, for example, when the application data

associated with the resource was not modified during the transaction. Here it does not matter whether the transaction is committed or rolled back. By returning VoteReadOnly the resource is opting out of the 2PC protocol and the resource object will not be contacted again by the transaction coordinator.

Figure 10 shows the 2PC protocol with two resource objects. In the first phase, the first resource returns VoteReadOnly and the second resource returns VoteCommit. During the second phase only the second resource is informed of the outcome (commit in this case).

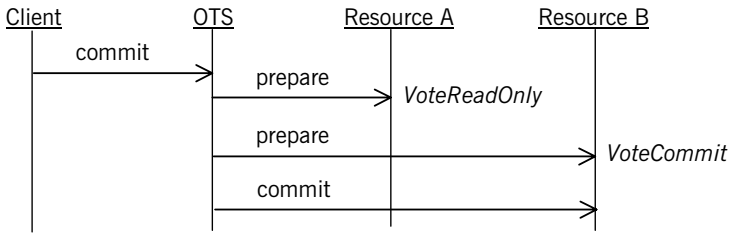

Figure 10: *A resource returning VoteReadOnly.*

The 1-Phase-Commit Protocol

The 1-phase-commit (1PC) protocol is an optimization of the 2PC protocol where the transaction only has one participant. Here the OTS can short circuit the 2PC protocol and ask the resource to commit the transaction directly. This is done by invoking the `commit_one_phase()` operation rather than the `prepare()` operation.

When the 1PC protocol is uses the OTS is delegating the commit decision to the resource object. If the resource object decides to commit the transaction, the `commit_one_phase()` operation returns successfully.

However, if the resource decides to rollback the transaction it must raise the `TRANSACTION_ROLLEDBACK` system exception. Figure 11 shows a successful 1PC protocol.

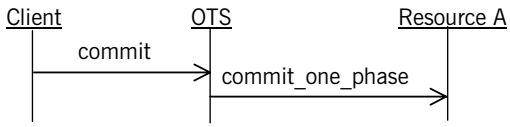

Figure 11: *A successful 1PC protocol.*

Figure 12 shows a 1PC protocol resulting in the transaction being rolled-back.

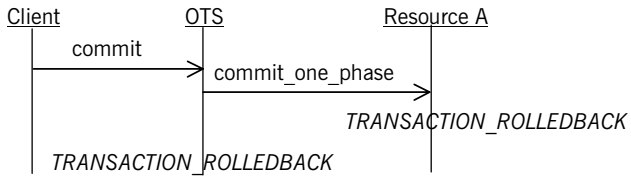

Figure 12: *The 1PC protocol resulting in a rollback.*

It is possible for the `commit_one_phase()` operation to be called even when more than one resource is registered with a transaction when resources return `VoteReadOnly` from `prepare()`. Assume for example there are three resources registered with a transaction. If the first two resources both return `VoteReadOnly` the third resource does not need to be prepared and the `commit_one_phase()` operation can be used instead.

Heuristic Outcomes

Heuristics outcomes occur when at least one resource object unilaterally decides to commit or rollback its part of the transaction and this decision is in conflict with the eventual outcome of the transaction. For example, a resource may have a policy that, once prepared, it will decide to commit if no outcome has been delivered within a certain period. This might be done to free up access to shared resources.

Any unilateral decisions made must be remembered by the resource. When the eventual outcome is delivered to the resource it must reply according to the compatibility of the decisions. For example, if the resource decides to commit its part of the transaction and the transaction is eventually rolled back, the resource's `rollback()` operation must raise the `HeuristicCommit` exception. The following table lists the resource's response for the various possible outcomes.

Table 8: *Heuristic Outcomes*

Resource Decision	Transaction Outcome	Resource's Response
Commit	Commit	commit() returns successfully.
Commit	Rollback	rollback() raises HeuristicCommit
Rollback	Rollback	rollback() returns successfully
Rollback	Commit	commit() raises HeuristicRollback

Once a resource has raised a heuristic exception it must remember this until the `forget()` operation has been called by the OTS (see Figure 13). For example, after a failure the OTS might invoke the rollback operation again in which case the resource must re-raise the `HeuristicCommit` exception. Once the `forget()` operation has been called the resource object is no longer required and can be deleted.

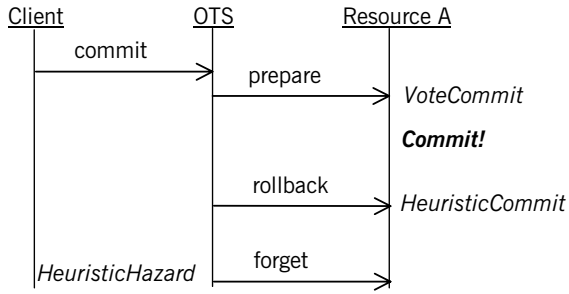

Figure 13: *Raising the HeuristicCommit exception*

Heuristic outcome are reported to the client only if true is passed to the `commit()` operation provided by the OTS Current object. They are reported by raising one of the exceptions: `HeuristicMixed` or `HeuristicHazard`. `HeuristicMixed` means a heuristic decision has been made resulting in some updates being committed and some being rolled back. `HeuristicHazard` indicates that a heuristic decision may have been made.

If the `commit_one_phase()` operation is called by the transaction coordinator, the commit decision is delegated to the resource implementation. This means that if the operation fails (that is results in a system exception other than `TRANSACTION_ROLLEDBACK` being raised) then the coordinator cannot know the true outcome of the transaction. For this case, the OTS raises the `HeuristicHazard` exception.

Failure and Recovery

Resource objects need to be able to deal with the failure of the process or node hosting the resource and the failure of the process or node hosting the OTS implementation.

Failure of the Resource

If the process or node hosting the resource object fails after the resource has been prepared, the resource object must be recreated during recovery so that the outcome of the transaction can be delivered to the resource. Figure 14 shows a crash occurring sometime after the resource has been prepared but before the coordinator invokes the `commit()` operation. When the coordinator does invoke the `commit()` operation the resource object is not active and the coordinator will attempt to commit later. In the meantime

the resource object is recreated and waits for the `commit()` operation to be invoked. The next time the coordinator calls `commit()` the resource receives the invocation and proceeds as normal.

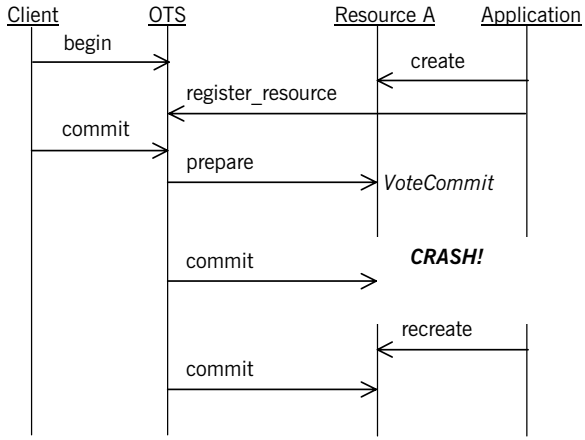

Figure 14: *Recovery after the failure of a resource object*

If the failure occurs before the resource has been prepared, there is no need to recreate the resource during recovery. When the 2PC protocol starts the OTS will not be able to contact the resource and the transaction will be rolled back.

Failure of the Transaction Coordinator

If the process or node hosting the transaction coordinator fails there are two possible ways in which the failure is resolved:

1. The transaction coordinator recovers and eventually sends the outcome to the resource. Here, the resource does not need to participate in the recovery; either the `commit()` or `rollback()` operation will be invoked as normal.
2. The resource detects that no outcome has been delivered and asks the transaction coordinator to complete the transactions. This is done using the `replay_completion()` operation provided by the recovery coordinator object.

The second way of resolving the failure of the OTS is required because the OTS supports a behavior called presumed rollback. With presumed rollback, if a transaction is rolled back the coordinator is not required to stably store this fact. Instead, on recovery if there is no information available on a transaction, the transaction is presumed to have rolled back. This saves on the amount of data that must be stably stored but means the resource object must check to see if the transaction has been rolled back.

Recall from "Creating and Registering Resource Objects" on page 95 when a resource is registered with the coordinator a reference to a recovery coordinator object is returned. The recovery coordinator supports the `RecoveryCoordinator` interface:

```
// IDL (in module CosTransactions)
interface RecoveryCoordinator
{
    Status replay_completion(in Resource r)
      raises (NotPrepared);
};
```

The sole operation, `replay_completion()`, takes a resource object and returns the status of the transaction. If the transaction has not been prepared the `NotPrepared` exception is raised. The `replay_completion()` operation is meant to hint to the coordinator that the resource is expecting the transaction to be completed.

To support detecting presumed rolled-back transactions, the `replay_completion()` operation is used to detect if the transaction still exists. If the transaction still exists the operation will either return a valid status or the `NotPrepared` exception. However, if the transaction no longer exists the `OBJECT_NOT_EXIST` system exception will be raised (other system exceptions should be ignored).

By periodically calling `replay_completion()` and checking for the `OBJECT_NOT_EXIST` exception, the resource object can detect rolled-back transactions (see Figure 15). This periodic calling of `replay_completion()` must be done before the resource has been prepared, after the resource has been prepared and after recovery of the resource due to a crash. To implement the latter, the resource object needs to stably store the recovery coordinator reference (for example using a stringified IOR) so that after a failure, the recovery coordinator can be contacted.

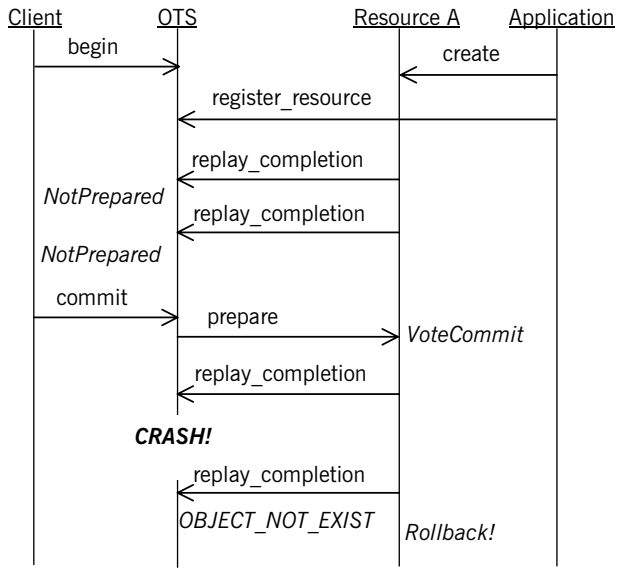

Figure 15: *Use of the replay_completion() operation*

Responsibilities and Lifecycle of a Resource Object

Overview

This section details the responsibilities of a resource object for each operation and shows the lifecycle of a resource object.

prepare()

```
Vote prepare() raises (HeuristicMixed, HeuristicHazard);
```

The `prepare()` operation is called during the first phase of the 2PC protocol allowing the resource to vote in the transaction's outcome and if necessary prepare for eventual commitment.

Voting is done by returning one of the three values `VoteCommit`, `VoteRollback` and `VoteReadOnly`:

VoteCommit	This indicates that the resource is willing to commit its part of the transaction and has fully prepared itself for the eventual outcome of the transaction. The next invocation on the resource will be either `commit()` or `rollback()`.
VoteRollback	This indicates that the resource has decided to rollback the transaction. This ensures that the transaction will be rolled back. The resource object can forget about the transaction and no further operations will be invoked on the resource object.
VoteReadOnly	This indicates that the resource does not want to be further involved in the 2PC protocol. This does not affect the transaction outcome and the resource object can forget about the transaction. No further operations will be invoked on the resource object.

If a resource object returns VoteCommit it must stably store sufficient information so that in the event of a failure, the resource object and its state can be reconstructed and continue to participate in the 2PC protocol. The actual information that is saved depends on the application, but typically it will include the following:

- The identity of the transaction. This can be obtained from the otid field in the transaction's propagation context which in turn is obtained by the `get_txcontext()` operation on the transaction's coordinator.
- The `ObjectID` for the resource.
- The reference for the recovery coordinator object associated with the resource. This can be saved as a stringified IOR obtained by the `object_to_string()` operation.
- Sufficient information to redo or undo any modifications made to application data by the transaction.

The `prepare()` operation can raise two exceptions dealing with heuristic outcomes: `HeuristicMixed` and `HeuristicHazard`. These exceptions may be used internally in an OTS implementation; most resource implementations do not need to raise these exceptions.

---

commit()

```
void commit() raises (NotPrepared, HeuristicRollback,
   HeuristicMixed, HeuristicHazard)
```

The `commit()` operation is called during the second phase of the 2PC protocol after the coordinator has decided to commit the transaction. The `commit()` operation may be invoked multiple times due to various failures such as a network error, failure of the OTS and failure of the application.

Typically the `commit()` operation does the following:

- Make permanent any modifications made to the data associated with the resource.
- Cleans up all traces of the transaction, including information stably stored for recovery.

The `commit()` operation can raise one of four user exceptions: `NotPrepared`, `HeuristicRollback`, `HeuristicMixed`, `HeuristicHazard`. The `NotPrepared` exception must be raised if `commit()` is invoked before the resource has been prepared (that is, returned `VoteCommit` from the `prepare()` operation).

The `HeuristicRollback` exception must be raised if the resource had decided to rollback its part of the transaction after being prepared and prior to the `commit()` operation being invoked. If this exception is raised it must be raised on future invocations of the `commit()` operation and the resource must wait for the `forget()` operation to be invoked before cleaning up the transaction.

The `HeuristicMixed` and `HeuristicHazard` exceptions may be used internally in an OTS implementation; most resource implementations do not need to raise these exceptions.

---

rollback()

```
void rollback() raises (HeuristicCommit, HeuristicMixed,
   HeuristicHazard)
```

There are two occasions when the `rollback()` operation is called:

1. During the second phase of the 2PC protocol after the coordinator has decided to commit the transaction.
2. When the transaction is rolled back prior to the start of the 2PC protocol. This may occur for several reasons including the client invoking the `rollback()` operation on the OTS Current object, the transaction begin timed-out, and an attempt to commit a transaction that has been marked for rollback.

The `rollback()` operation may be invoked multiple times due to various failures such as a network error, failure of the OTS and failure of the application.

Typically the `rollback()` operation does the following:

- Undo any modifications made to the data associated with the resource.
- Cleans up all traces of the transaction, including information stably stored for recovery.

The `rollback()` operation can raise one of three user exceptions: `HeuristicCommit`, `HeuristicMixed`, `HeuristicHazard`. The `HeuristicCommit` exception must be raised if the resource had decided to commit its part of the transaction after being prepared and prior to the `rollback()` operation being invoked. If this exception is raised it must be raised on future invocations of the `rollback()` operation and the resource must wait for the `forget()` operation to be invoked before cleaning up the transaction. Heuristic exceptions can only be raised if the resource has been prepared.

The HeuristicMixed and HeuristicHazard exceptions may be used internally in an OTS implementation; most resource implementations do not need to raise these exceptions.

---

commit_one_phase()

```
void commit_one_phase() raises (HeuristicHazard)
```

The commit_one_phase() operation may be invoked when there is only one resource registered with the transaction. The resource decides whether to commit or rollback the transaction. Typically the commit_one_phase() operation does the following:

- An attempt is made to commit any changes made to the application data. If this succeeds the operation returns normally; otherwise the changes are undone and the TRANSACTION_ROLLEDBACK system exception is raised.
- Cleans up all traces of the transaction.

The HeuristicHazard exception must be raised if the resource cannot determine whether the commit attempt was successful or not. If this exception is raised the resource must wait for the forget() operation to be invoked before cleaning up the transaction.

---

forget()

```
void forget()
```

The forget() operation is called after the resource object raised a heuristic exception from either commit(), rollback() or commit_one_phase(). The forget() operation may be invoked multiple times due to various failures such as a network error, failure of the OTS and failure of the application. Typically the resource cleans up all traces of the transaction, including information stably stored for recovery.

Resource Object Checklist

The following is a list of things to remember when implementing recoverable resource objects:

- A resource object can only be registered with one transaction. At the end of the resource's lifecycle the resource must be deactivated.
- Resource objects need unique identifiers. This means they must be created in a POA with a `USER_ID` ID assignment policy.
- Resource objects must be able to be recreated after a failure. This means they must be created in a POA with a `PERSISTENT` lifecycle policy.
- Resource objects must implement both the 2PC operations (`prepare()`, `commit()`, `rollback()` and `forget()`) as well as the 1PC operation (`commit_one_phase()`).
- Only return `VoteCommit` from the `prepare()` operation if the resource can commit the transaction and has stably stored sufficient state to be recreated after a failure.
- If a resource object wants to opt out of the 2PC protocol, it should return `VoteReadOnly` from the `prepare()` operation.
- If the resource takes heuristic decisions, the decisions must be remembered and reported to the OTS.
- Periodically call the `replay_completion()` operation to check for presumed rollback transactions.
- Resources are expensive in terms of 2PC messages and stable storage for recovery. Design your applications to minimize the number of resources used.

CHAPTER 9

Interoperability

This chapter describes how the Orbix OTS interoperates with older releases of Orbix and with other OTS implementations including the Orbix 3 OTS.

In this chapter

This chapter discusses the following topics:

Use of InvocationPolicies

Deprecated policies

This release of Orbix introduces the OTSPolicies, InvocationPolicies and NonTxTargetPolicies that replace the deprecated TransactionPolicies. The deprecated TransactionPolicies (for example, Requires_shared and Allows_shared) are supported allowing interoperability between different releases of Orbix.

When creating Orbix transactional POAs that must interoperate with previous releases, the policies for the POA must include the deprecated `TransactionPolicy` as well as the OTSPolicy and InvocationPolicy. See "Migrating from TransactionPolicies" on page 59 for more details.

Note: Support for the `TransactionPolicy` type may be discontinued in a future Orbix release. It is recommended that only OTSPolicies and InvocationPolicies be used.

Use of the TransactionalObject Interface

Enabling support for the TransactionalObject interface

Version 1.1 of the OTS specification uses inheritance from the empty `CosTransactions:TransactionalObject` interface to indicate the transactional requirements of an object. For example, the Orbix 3 OTS only supports the `TransactionalObject` interface and not the policies.

Orbix provides support for the `TransactionalObject` interface, allowing different behaviors to be configured. This support needs to be enabled by setting the `plugins:ots:support_ots_v11` configuration variable to `"true"` (by default this support is not enabled). Once enabled, an object which supports the `TransactionalObject` interface is interpreted as having an effective OTSPolicy which depends on the value of the `plugins:ots:ots_v11_policy` configuration variable. Table 9 details this mapping:

Table 9: *Mapping TransactionalObject to OTSPolicies*

Inherits from TransactionalObject	Value of plugins:ots:ots_v11_policy	Effective OTSPolicy Value
No	n/a	FORBIDS
Yes	“requires”	REQUIRES
Yes	“adapts”	ADAPTS

The default value for the `plugins:ots:ots_v11_policy` is “requires” since this is the default behavior for the Orbix 3 OTS. For backward compatibility with previous Orbix releases a value of “allows” is interpreted as “adapts”.

It is recommended that the when support for TransactionalObject is enabled, the NonTxTargetPolicy `PERMIT` should be used.

If an object supports TransactionalObject and also uses OTSPolicies, the OTSPolicies take priority; compatibility checks are not done.

To summarize, to enable support for the `TransactionalObject` interface the following is required:

1. Set the `plugins:ots:support_ots_v11` configuration variable to `"true"`.
2. Set the `plugins:ots:ots_v11_policy` configuration variable to either `"requires"` (the default) or `"adapts"`.
3. Use the `PERMIT` NonTxTargetPolicy (for example, by setting the `policies:non_tx_target_policy` configuration variable to `"permit"`).

Interoperability with Orbix 3 OTS Applications

Overview

This section details how an Orbix client can interoperate with an existing Orbix 3 OTS application. Since Orbix 3 supports only the `TransactionalObject` interface this section is an extension of the previous section "Use of the TransactionalObject Interface" on page 115

Details on using the Encina OTS are covered in "The Encina Transaction Manager" on page 128.

Orbix 3 OTS Interoperability

Figure 16 shows an Orbix client working with an existing Orbix 3 OTS application. The first thing to note is that the Orbix 3 OTS always requires a full 2PC transaction manager such as that provided by the Encina OTS (see "The Encina Transaction Manager" on page 128) or the otstf provided with Orbix 3. A 1PC-only transaction created by the OTS Lite transaction manager will not be usable by the Orbix 3 OTS. This means that the Orbix client must be configured to use an external transaction factory to create transactions.

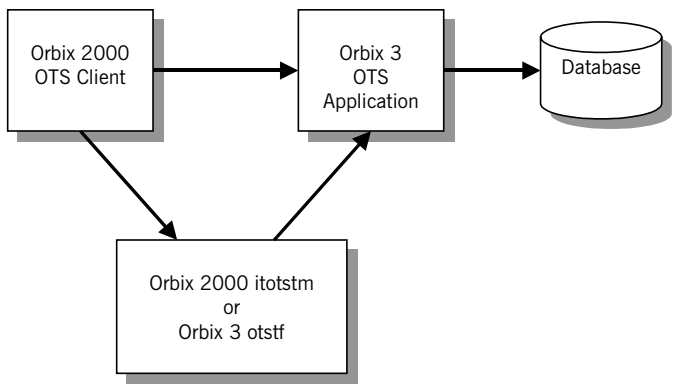

Figure 16: *Interoperability with Orbix 3 OTS Applications*

Using otstf as transaction manager

To get the Orbix client to use the Orbix 3 otstf server as its transaction manager, the `initial_references:TransactionFactory:reference` configuration variable must be set to the reference of the otstf's transaction factory object. This can be done by passing the `-T` switch to the otstf and copying the IOR reference output. Alternatively the otstf can publish its name to the name service using the `-t` switch and a suitable corbaname URL can be used as the reference value (see the section "Resolving Names with corbaname" in the chapter "Naming Service" in the *CORBA Programmer's Guide*).

The Orbix 3 OTS application must be enabled to import standard transaction contexts. This is done by setting the Orbix 3 `OrbixOTS.INTEROP` configuration variable to `"TRUE"`.

The final consideration is the mapping from inheritance from TransactionalObject to the effective OTSPolicy. The Orbix 3 OTS provides a proprietary policy mechanism which mimics the behavior of the OTSPolicies `REQUIRES` and `ADAPTS` (the default being `REQUIRES`). Therefore, when selecting the value for the `plugins:ots:ots_v11_policy` configuration variable, make sure it matches the policy expected by the Orbix 3 application.

Summary

The following is a checklist for enabling interoperability between Orbix clients and Orbix 3 OTS applications.

1. Set the `plugins:ots:support_ots_v11` configuration variable to `"true"`.
2. Set the `plugins:ots:ots_v11_policy` configuration variable to match the equivalent Orbix 3 OTS policy for the `TransactionalObject` interface.
3. Use the `PERMIT` NonTxTargetPolicy.
4. Set the `initial_references:TransactionFactory:reference` configuration variable to refer to either the Orbix 3 otstf's transaction factory another transaction factory that supports 2PC.
5. Set the Orbix 3 `OrbixOTS.INTEROP` configuration variable to `"TRUE"`.

For more information on the use of the otstf server and setting Orbix 3 transaction policies, refer to the Orbix 3 OTS manual.

Using the Orbix 3 otstf with Orbix Applications

Using Orbix 3 otstf transaction manager

Another possible use of Orbix 3 is to use the 2PC otstf transaction manager with an Orbix OTS application. This setup is shown in Figure 17.

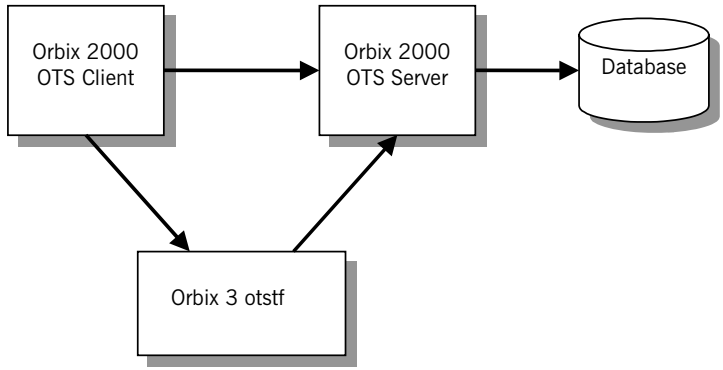

Figure 17: *Using and alternative OTS Implementation*

This setup is achieved by setting the `initial_references:TransactionFactory:reference` configuration variable to refer to the otstf's transaction factory.

CHAPTER 10

OTS Plug-Ins and Deployment Options

Orbix provides a generic OTS plugin that provides an implementation of the OTS Current object including transaction propagation. In addition there are two OTS transaction manager implementations: OTS Lite, which provides a lightweight transaction coordinator supporting only the 1PC protocol, and OTS Encina, which provides full recoverable 2PC support. This chapter discusses deployment options.

In this chapter

This chapter discusses the following topics:

OTS Plug-ins

Orbix provides a generic OTS plugin that provides an implementation of the OTS Current object including transaction propagation.

There are two OTS transaction manager implementations:

- OTS Lite
- OTS Encina.

OTS Lite

OTS Lite provides lightweight transaction coordinator supporting only the 1PC protocol. It is available as an application plug-in and requires minimal configuration and administration but can only be used by applications with only a single resource manager.

OTS Encina

OTS Encina provides full recoverable 2PC support allowing it to be used by applications that are using one or more resource managers. It is available as a standalone service and as a application plug-in.

Note: OTS Encina is only available in the Orbix Enterprise Edition.

Features in OTS

Table 10 shows the features supported by these pieces.

Table 10: *Features in OTS Implementation*

Feature	Generic OTS	OTS Lite	OTS Encina
Current Object	Y		
Transaction Policies	Y		
Old Transaction Policies	Y		
TransactionalObject	Y		
1PC Protocol		Y	Y
2PC Protocol		N	Y
Resource Objects		Y	Y
Synchronization Objects		Y	Y
Nested Transactions		N	N

Table 10: *Features in OTS Implementation*

Feature	Generic OTS	OTS Lite	OTS Encina
IONA Administrator Management		Y	Y
JTA Support	Y	Y	Y
Application Plug-In	Y	Y	N

The OTS Plug-In

Purpose of the OTS plug-in

Any application using the OTS Current object needs to load the OTS plug-in. This plug-in provides an implementation of the OTS Current object which provides the thread/transaction association, propagation of the current transaction to transactional objects and the policies OTSPolicy, InvocationPolicy and NonTxTargetPolicy. In addition the OTS plug-in provides the client stubs for the CosTransactions module, so applications need to load the OTS plug-in classes.

In OTS plug-in does not provide any transaction manager functionality. Instead the OTS plug-in delegates elsewhere using the standard CosTransactions module APIs (see Figure 18). This allows different deployment options to be easily supported through configuration.

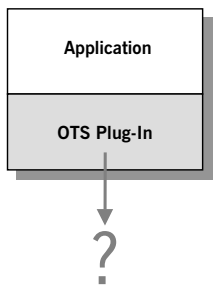

Figure 18: *The Generic OTS Plug-In*

Loading the OTS plug-in

There are two ways in which the OTS plug-in can be loaded:

1. Explicitly adding the plug-in name `"ots"` to the `orb_plugins` configuration variable. For example: `orb_plugins = [..., "ots"];`
2. Setting the `initial_references:TransactionCurrent:plugin` configuration variable to the value `"ots"`. This causes the OTS plug-in to be loaded when `resolve_initial_references("TransactionCurrent")` is called.

When using this way, `resolve_initial_references()` should be called immediately after `ORB_init()` has been called and before any transaction POAs are created.

When the OTS plug-in is initialized it obtains a reference to a transaction factory object by calling resolve_initial_references("TransactionFactory"). So changing which transaction manager to use is just a matter of using configuration to change the outcome of `resolve_initial_references()`.

Deployment scenarios

The remainder of this section describes two possible deployment scenarios for Java:

- Using the OTS Lite plug-in when only 1PC transactions are required.
- Using the itotstm service with the OTS Encina plug-in where recoverable 2PC transactions are required.

For more information, see the *Orbix Deployment Guide*.

The OTS Lite Plug-In

Overview

The OTS Lite plug-in is a lightweight transaction manager that only supports the 1PC protocol. 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 19 shows a client/server deployment that uses the OTS Lite plug-in.

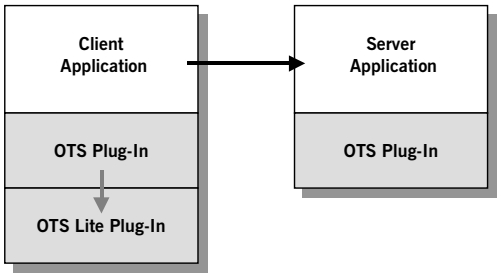

Figure 19: *Deployment using the OTS Lite Plug-In*

As usual both the client and server applications must load the OTS plug-in. In addition the client application loads the OTS Lite plug-in, allowing the client to create 1PC transaction locally.

Note: When using the Orbix configuration tool, `itconfigure`, the OTS Lite plug-in is deployed by default.

Loading the OTS Lite plug-in

As with the OTS plug-in the OTS Lite plug-in can be loaded in two ways:

1. Adding the plug-in name "ots_lite" to the orb_plugins configuration variable. For example: `orb_plugins = [..., "ots", "ots_lite"];`
2. Setting the `initial_references:TransactionFactory:plugin` configuration variable to "ots_lite". This causes the OTS Lite plug-in to be loaded by the OTS plug-in when `resolve_initial_references("TransactionFactory")` is called.

The server application does not need to load the OTS Lite plug-in except when standard interposition is used (that is, when the `plugins:ots:interposition_style` configuration variable is set to "standard"). In this case when the OTS plug-in imports the transaction from the client a transaction manager is required to create the sub-coordinated transaction.

This deployment should be used when the application only accesses on transactional resource (for example, updates a single database).

The Encina Transaction Manager

Overview

The Encina OTS Transaction Manager provides full recoverable 2PC transaction coordination implemented on top of the industry proven Encina Toolkit from IBM/Transarc.

The Encina OTS may be used via the itotstm service.

Configuring the OTS Encina Plug-In

Whether the OTS Encina plug-in is used in the itotstm service or directly in the application, there are a number of administration steps required to successfully use it.

Note: If you selected Distributed Transaction services when running the Orbix configuration tool, `itconfigure`, the administration steps outlined in this subsection are done automatically.

1. Two transient POAs must be created. These serve as namespace POAs off which the OTS Encina plug-in creates its persistent POAs. The first POA is called “iOTS” and the second is a child POA whose name is set by the plugins:ots_encina:namespace_poa. The default value of this configuration variable is `“otstm”` for the itotstm service and `“Encina”` for an application loading the plug-in. The POAs should be created using itadmin as follows:

   ```
   itadmin poa create –transient –allowdynamic iOTS
   itadmin poa create –transient –allowdynamic iOTS/otstm
   ```

2. The Encina OTS is fully recoverable and requires a transaction log to write the state of its transactions. Assuming the log file is to be located in “/local/logs/ots.log” the log is created and initialized using itadmin as follows:

   ```
   itadmin encinalog create /local/logs/ots.log
   itadmin encinalog init /local/logs/ots.log
   ```

The effect of initializing the log is to create a restart file. This a file that contains sufficient information for the OTS Encina plug-in to restart and includes the location of the transaction log. In this example, the restart file is called /local/logs/ots_restart. The name of the restart file must be passed to the OTS Encina plug-in by setting the `plugins:ots_encina:restart_file` configuration variable.

The minimum configuration required to load the OTS Encina plug-in into an applications is:

```
<app-scope> {
  initial_references:TransactionFactory:plugin = “ots_encina”;
  plugins:ots_encina:namespace_poa = “<name>”;
  plugins:ots_encina:restart_file = “<path>”;
}
```

The itotstm Transaction Manager Service

Overview

The itotstm program is a standalone transaction manager service which can be configured to load any transaction manager plug-in. This section shows how it can be used along with the Encina OTS plug-in to provide 2PC transactions for an application. The itotstm service is deployed if you select the Distributed Transaction service when running the Orbix configuration tool, `itconfigure`.

Using itconfigure

If you select the Distributed Transaction service when running the Orbix configuration tool, `itconfigure`, the OTS Lite plug-in and the itotstm service are deployed. By default the OTS Lite plug-in is configured to be used by all clients and servers. To make use of the itotstm service, however, clients need to pick up the `initial_references:TransactionFactory:reference` configuration variable that is set in the `iona_services.otstm client` configuration scope. This can be done this by passing `"-ORBname iona_services.otstm.client"` to the `ORB_init()` operation or by adding a copy of the variable to the application's configuration scope.

Example client/server deployment

Figure 20 shows a client/server deployment where the itotstm in conjunction with the OTS Encina plug-in is used to provide 2PC transaction management. Here, neither the client nor the server needs to load any transaction manager plug-in. Instead the client OTS is configured to pick up its transaction factory reference from the OTS Encina plug-in loaded into the itotstm standalone service.

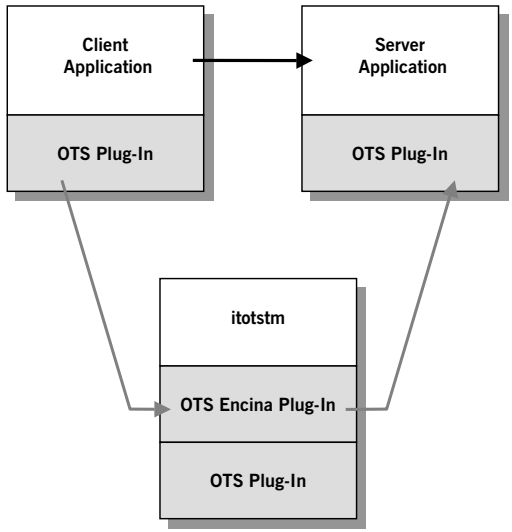

Figure 20: *Using the OTS Encina plug-in with the itotstm service*

There are two parts to setting up such a deployment.

- Configuring the itotstm to load the OTS Encina plug-in.
- Configuring the OTS plug-in to pickup the reference to the OTS Encina transaction factory within the itotstm service.

Configuring itotstm

The itotstm service uses the configuration scope "otstm" by default. This can be changed by using a different ORB name using the -ORBname command line option. Configuring itotstm to load the OTS Encina plug-in can be done in two ways:

1. Adding the OTS plug-in name "`ots_encina`" to the `orb_plugins` configuration variable. For example, `orb_plugins = [..., "ots", "ots_encina"];`
2. Setting the `initial_references:TransactionFactory:plugin` configuration variable to the name of the OTS Encina plug-in "`ots_encina`".

Note that in both cases the `orb_plugins` configuration variable must contain `"ots"` since the OTS plug-in is required for synchronization objects.

The remainder of the otstm scope should contain the configuration necessary for the OTS Encina plug-in.

Configuring the OTS plug-in

Next the OTS plug-in loaded into the applications needs to pick up the transaction factory reference of the OTS Encina plug-in. Essentially this means setting the `initial_references:TransactionFactory:refererence` configuration variable in the applications configuration scope to any suitable reference. Three possible ways of achieving this are:

1. Get the OTS Encina plug-in to export its transaction factory reference to the name service and use a corbaname style URL for the initial reference. This is done by setting the `plugins:ots_encina:transaction_factory_ns_name` configuration variable to the name for the object reference in the name service. For example, if this is set to `"ots/encina"` a URL of the form `"corbaname:rir:#ots/encina"` can be used.
2. Get the itotstm to publish the transaction factory IOR to a file using the "prepare" and "-publish_to_file" command-line switches. Then use the IOR in the file as the transaction factory reference.

The deployment should be used when the application requires or might require full recoverable 2PC transactions. For example, the application make use of ore or more resource managers.

APPENDIX A

OTS Management

This appendix describes the OTS server features that have been exposed for management. It explains all the managed entities, attributes, and operations. These can be managed using the IONA Administrator management consoles.

In this Appendix

This appendix contains the following sections:

Introduction to OTS Management

Overview

This section provides an introduction to the OTS management model and the IONA Administrator management consoles.

OTS Management Model

Figure 21 shows the main components of the OTS management model.

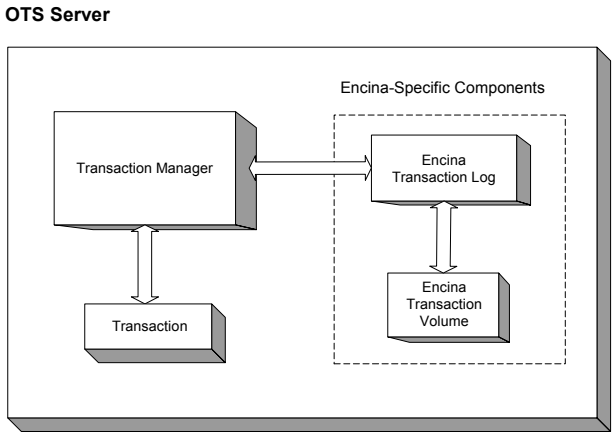

Figure 21: *OTS Management Model*

In Figure 21, the components on the left are common to both OTS Encina and OTS Lite. The components on the right apply to OTS Encina only.

In this model, each OTS server can have multiple Transactions and multiple Encina Transaction Volumes. However, each server can only have one Transaction Manager, and one Encina Transaction Log.

OTS Managed Entities

The following OTS server components have been instrumented for management:

- TransactionManager / Encina TransactionManager
- Transaction / Encina Transaction
- Encina Transaction Log
- Encina Volume

This means that these features can be managed using the IONA Administrator management consoles.

IONA Administrator

IONA Administrator is a set of tools that enables you to manage and configure server applications at runtime. IONA Administrator provides a graphical user interface known as the *IONA Administrator Console*. This enables you to manage applications, configuration settings, event logging, and user roles.

IONA Administrator also provides a web browser interface known as the *IONA Administrator Web Console.* The web console enables you to manage applications and event logging from anywhere, without the need for a lengthy download or installation.

For detailed information about IONA Administrator, see the *CORBA IONA Administrator User's Guide*.

Example Managed Entity

Figure 22 shows an OTS **Encina Transaction Manager** running in the **IONA Administrator** web console**.** It shows the attributes and operations that are exposed for this entity.

The next sections in this chapter describe the attributes and operations that are displayed for each of the OTS managed entities.

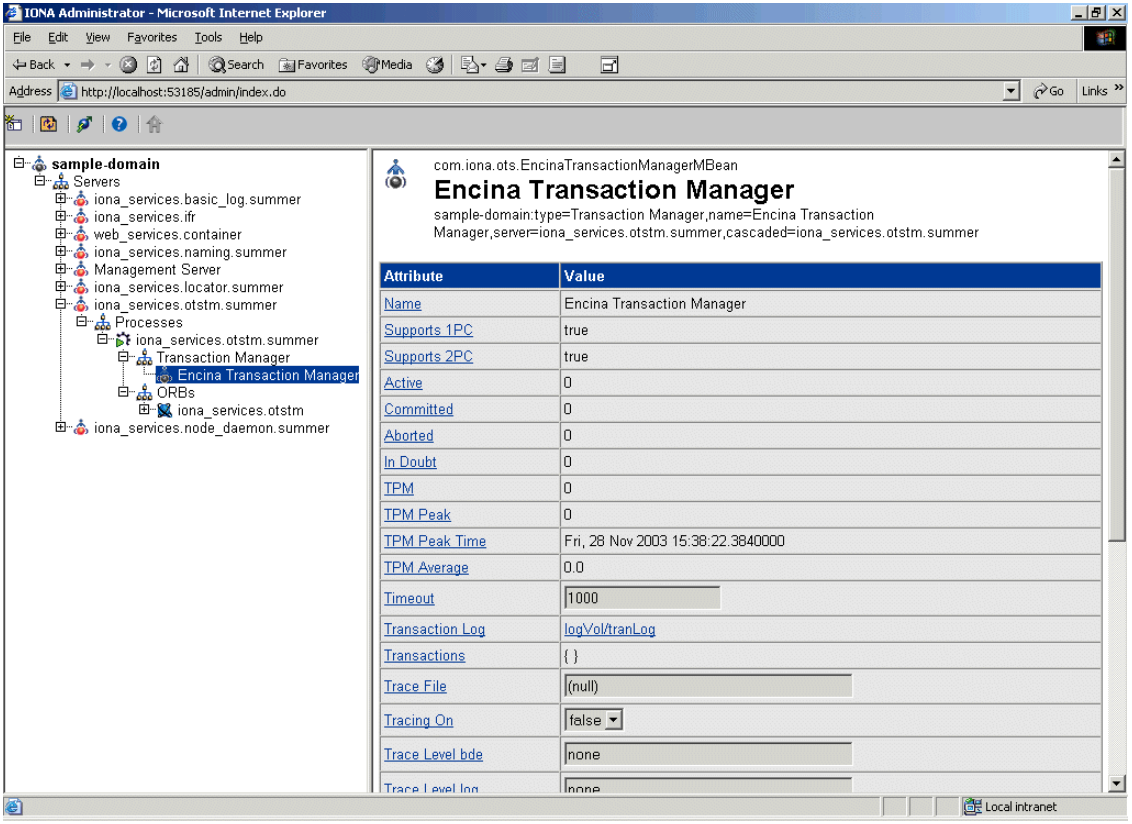

Figure 22: *OTS Encina Transaction Manager Entity*

TransactionManager Entity

Overview

This section describes the managed attributes and operations that are exposed for the TransactionManager and Encina TransactionManager entity. These attributes and operations are displayed in the **IONA Administrator Console**.

TransactionManager Attributes

The managed attributes for the TransactionManager entity are shown in Table 11. These attributes apply to both OTS Encina and OTS Lite.

Table 11: *TransactionManager Attributes (Sheet 1 of 2)*

Attribute	Type	Description
Name	string	Name of the transaction manager.
Supports 1PC	boolean	Whether the manager supports one-phase commit.
Supports 2PC	boolean	Whether the manager supports two-phase commit.
Active	long	Number of active transactions.
Completed	long	Number of completed transactions (since the server started).
Committed	long	Number of committed transactions (since the server started).
Aborted	long	Number of aborted transactions.
In Doubt	long	Number of transactions that are in doubt.
TPM	long	Number of transactions per minute.

Table 11: *TransactionManager Attributes (Sheet 2 of 2)*

Attribute	Type	Description
TPM Peak	long	Maximum number of transactions per minute (since the server started).
TPM Peak Time	string	Time when the maximum transactions per minute was reached.
TPM Peak Average	double	Average transactions per minute (since server started).
Timeout	long	Default value for transaction timeout (same as the `default_transaction_timeout` configuration variable for the `ots_lite` and `ots_encina` plug-ins). This attribute is writable.
Transaction Log	hyperlink	Hyperlink to the Transaction Log entity (null for OTS Lite).

Encina TransactionManager Attributes

The additional managed attributes for the Encina TransactionManager entity are shown in Table 12. These attributes apply to OTS Encina only.

Table 12: *Encina TransactionManager Attributes (Sheet 1 of 2)*

Attribute	Type	Description
Trace File	string	The file to which the trace output is written (`stderr` if the string is empty). This attribute is writable.
Trace On	boolean	Whether Encina tracing is enabled or not. This attribute is writable.

Table 12: *Encina TransactionManager Attributes (Sheet 2 of 2)*

Attribute	Type	Description
Trace Level bde Trace Level log Trace Level restart Trace Level tran Trace Level util Trace Level vol	space-separated list of strings, where each element is one of the following: `GLOBAL, EVENT, PARAM, NONE, INTERNAL_PARAM, INTERNAL_EVENT` (for example, `"EVENT PARAM"`)	These attributes specify the trace level for the corresponding Encina module (one of `bde`, `log`, `restart`, `tran`, `util`, `vol`, respectively). These attributes are writable.

Encina TransactionManager Operations

The managed operations for the Encina TransactionManager entity are shown in Table 13.

Table 13: *Encina TransactionManager Operations*

Operation	Parameters	Type	Description
dump	file name overwrite	string boolean	Writes the contents of the Encina trace buffer to the specified file. Depending on the value of the `overwrite` parameter, appends to an existing file, or overwrites it.

Transaction Entity

Overview

This section describes the managed attributes and operations exposed for the Transaction and Encina Transaction entity. These attributes and operations are displayed in the **IONA Administrator Console**.

Transaction Attributes

The managed attributes for the Transaction entity are shown in Table 14. These attributes apply to both OTS Encina and OTS Lite.

Table 14: *Transaction Attributes*

Attribute	Type	Description
Global TID	string	Global transaction identifier.
Timeout	boolean	Transaction-specific timeout.
Creation Time	boolean	Time when the transaction was created.
Status	long	`CosTransactions::Status` values.
Resources	long	Available resources for the transaction.

Encina Transaction Attributes

The additional managed attributes for the Encina Transaction entity are shown in Table 15. These attributes apply to OTS Encina only.

Table 15: *Encina Transaction Attributes*

Attribute	Type	Description
Local TID	string	Local Encina-specific transaction identifier.

Transaction Operations

The managed operations for the Transaction entity are shown in Table 16. These operations apply to both OTS Encina and OTS Lite.

Table 16: *Transaction Operations*

Operation	Parameter	Description
Rollback	none	Roll back the transaction.
Mark Rollback	none	Mark the transaction for being rolled back.
Commit	none	Commit the transaction.
Remove Resource	string	Remove (unregister) the resource identified by the stringified object reference from the transaction. For example, this enables a transaction to complete if repeated attempts to deliver an outcome to a resource are failing.

Note: These operations are applicable to all transactions. In practice however, these operations will most likely fail for well-behaved transactions because of their short lifetime. They would only be applied in critical situations (for example, on a transaction with resource failures).

Encina Transaction Log Entity

Overview

This section describes the managed attributes and operations exposed for the Encina Transaction Log entity. These attributes and operations are displayed in the **IONA Administrator Console**.

Encina Transaction Log Attributes

The managed attributes for the Encina Transaction Log are shown in Table 17.

Table 17: *Encina Transaction Log Attributes (Sheet 1 of 2)*

Attribute	Type	Description
Name	string	Name of the log (always `tranLog` for the Encina Transaction Log).
Size	long	Size (in pages of 512 K).
Free	long	Free space (in pages).
Threshold	long	Percentage of used pages versus total pages that (when exceeded) cause a management event to be sent to the management service. This attribute is writable.
Check Interval	long	Interval (in seconds) for checking the amount of free space in the log. This attribute is writable.
Growth	long	Difference of free space in the log at beginning and end of the last check interval.
Average Growth	double	Average of the growth rate (in the lifetime of the OTS server).
Archive Device	string	File name of the archive device of the log.

Table 17: *Encina Transaction Log Attributes (Sheet 2 of 2)*

Attribute	Type	Description
Mirrors	list of hyperlinks	List of hyperlinks to Encina Volume entities.

Encina Transaction Log Operations

The managed operations for the Encina Transaction Log are shown in Table 18.

Table 18: *Encina Transaction Log Operations*

Operation	Parameters	Description
Expand	none	Expands the log to maximum possible size. This is necessary to avail of the increased disk space after a mirror has been added.
Add Mirror	string	Creates a new physical volume backed up by the specified disk, and adds it to the list of volumes currently mirroring the transaction log. The raw partition or file specified by the string parameter must exist. You can create files using the `itadmin` tool.

Encina Volume Entity

Overview

This section describes the managed attributes and operations exposed for the Encina (Physical) Volume entity. These attributes and operations are displayed in the **IONA Administrator Console**.

Encina Volume Attributes

The managed attributes for the Encina (Physical) Volume entity are shown in Table 19.

Table 19: *Encina (Physical) Volume Attributes*

Attribute	Type	Description
Name	string	Logical name of the physical volume.
Disks	list of strings	List of fully qualified file or raw partition names for the different disks that backup the volume.

Encina Volume Operations

The managed operations for the Encina (Physical) Volume are shown in Table 20.

Table 20: *Encina (Physical) Volume Operations*

Operation	Parameter	Description
Remove	none	Removes this physical volume.
Add Disk	string	Adds the specified disk to the physical volume. The raw partition or file must exist. You can create files using the `itadmin` tool.

Management Events

The following OTS events are logged with the IONA Administrator management service:

- The heuristic outcome of a transaction.

  This event includes the `otid` and the heuristic outcome type.
- When the used space in the transaction log exceeds the threshold.

  This event includes the actual percentage of used versus the total number of pages in the transaction log.

Glossary

A

administration

All aspects of installing, configuring, deploying, monitoring, and managing a system.

C

client

An application (process) that typically runs on a desktop and requests services from other applications that often run on different machines (known as server processes). In CORBA, a client is a program that requests services from CORBA objects.

configuration

A specific arrangement of system elements and settings.

configuration domain

Contains all the configuration information that Orbix ORBs, services and applications use. Defines a set of common configuration settings that specify available services and control ORB behavior. This information consists of configuration variables and their values. Configuration domain data can be implemented and maintained in a centralised Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organise ORBs into manageable groups, thereby bringing scalability and ease of use to the largest environments. See also configuration file and configuration repository.

configuration file

A file that contains configuration information for Orbix components within a specific configuration domain. See also configuration domain.

configuration repository

A centralised store of configuration information for all Orbix components within a specific configuration domain. See also configuration domain.

configuration scope
Orbix configuration is divided into scopes. These are typically organized into a root scope and a hierarchy of nested scopes, the fully-qualified names of which map directly to ORB names. By organising configuration properties into scopes, different settings can be provided for individual ORBs, or common settings for groups of ORB. Orbix services have their own configuration scopes.

CORBA
Common Object Request Broker Architecture. An open standard that enables objects to communicate with one another regardless of what programming language they are written in, or what operating system they run on. The CORBA specification is produced and maintained by the OMG. See also OMG.

CORBA objects
Self-contained software entities that consist of both data and the procedures to manipulate that data. Can be implemented in any programming language that CORBA supports, such as C++ and Java.

D

deployment
The process of distributing a configuration or system element into an environment.

E

event
The occurrence of a condition or state change, or the availability of some information that is of interest to one or more modules in a system. Suppliers generate events and consumers subscribe to receive them.

I

IDL
Interface Definition Language. The CORBA standard declarative language that allows a programmer to define interfaces to CORBA objects. An IDL file defines the public API that CORBA objects expose in a server application. Clients use these interfaces to access server objects across a network. IDL interfaces are independent of operating systems and programming languages.

IIOP
Internet Inter-ORB Protocol. The CORBA standard messaging protocol, defined by the OMG, for communications between ORBs and distributed applications. IIOP is defined as a protocol layer above the transport layer, TCP/IP.

installation
The placement of software on a computer. Installation does not include configuration unless a default configuration is supplied.

Interface Definition Language
See IDL.

invocation
A request issued on an already active software component.

IOR
Interoperable Object Reference. See object reference.

M

management
To direct or control the use of a system or component. Sometimes used in a more general way meaning the same as Administration. management console

---

N

node daemon
Starts, monitors, and manages servers on a host machine. Every machine that runs a server must run a node daemon.

O

object reference
Uniquely identifies a local or remote object instance. Can be stored in a CORBA naming service, in a file or in a URL. The contact details that a client application uses to communicate with a CORBA object. Also known as interoperable object reference (IOR) or proxy.

object transaction service
See Orbix OTS.

OMG

Object Management Group. An open membership, not-for-profit consortium that produces and maintains computer industry specifications for interoperable enterprise applications, including CORBA. See www.omg.com.

ORB

Object Request Broker. Manages the interaction between clients and servers, using the Internet Inter-ORB Protocol (IIOP). Enables clients to make requests and receive replies from servers in a distributed computer environment. Key component in CORBA.

Orbix OTS

Object Transaction Service. An implementation of the OMG Transaction Service Specification. Provides interfaces to manage the demarcation of transactions and the propagation of transaction contexts.

POA

Portable Object Adapter. Maps object references to their concrete implementations in a server. Creates and manages object references to all objects used by an application, manages object state, and provides the infrastructure to support persistent objects and the portability of object implementations between different ORB products. Can be transient or persistent.

protocol

Format for the layout of messages sent over a network.

S

server

A program that provides services to clients. CORBA servers act as containers for CORBA objects, allowing clients to access those objects using IDL interfaces.

T

transaction manager

Manages global transactions on behalf of application programs. A transaction manager coordinates commands from application programs and resource managers to start and complete global transactions. When an application

completes a transaction, either with a commit or rollback request, the transaction manager communicates the outcome with each resource manager.

Index

P

R

S

T

U

V

W

X