MICRO FOCUS

Orbix Mainframe 6.3.1

CORBA OTS Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK
https://www.microfocus.com

2021-03-18

Contents

Part 2 Administration

Part 3 Appendices

List of Figures

List of Tables

Preface

Orbix OTS is a full implementation of the interoperable transaction service as specified by the Object Management Group. It allows:

- COBOL or PL/I CICS transactions, using the Orbix CICS client adapter, to initiate two-phase commit processing.
- COBOL or PL/I IMS transactions, using the Orbix IMS client adapter, to initiate two-phase commit processing.
- C++ programs running on z/OS or z/OS Unix Systems Services to use two-phase commit processing.

Orbix OTS complies with the following specifications:

- CORBA 2.6
- OTS 1.2
- GIOP 1.2 (default), 1.1, and 1.0

Audience

Part 1 of this guide is intended for application programmers who want to develop OTS transactions, using C++ interfaces. Chapter 1 is relevant to C++, COBOL and PL/I. However, the rest of Part 1 is relevant to C++ only.

Note: For information on developing transactions in COBOL or PL/I that can initiate two-phase commit processing from CICS or IMS, see the *COBOL Programmer's Guide and Reference* and *PL/I Programmer's Guide and Reference*.

The guide will help you become familiar with the C++ interfaces used for two-phase commit processing. It assumes that you are familiar with CORBA concepts and C++.

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, see the *CORBA Programmer's Reference, C++*.

Part 2 and Part 3 of this guide are intended for z/OS system programmers who want to familiarize with the administration issues that relate to the use of OTS on the mainframe. All chapters in these parts of the guide are relevant regardless of which programming language is being used.

Note: For information on setting up the Orbix CICS client adapter or Orbix IMS client adapter to support two-phase commit processing for CICS or IMS transactions, see the *CICS Adapters Administrator's Guide* and *IMS Adapters Administrator's Guide*.

Related documentation

For the latest versions of Orbix Mainframe product documentation, see the following web page:

https://www.microfocus.com/documentation/orbix/

Organization of this guide

This guide is divided into the following chapters:

Chapter 1 provides a brief overview of the basic concepts involved in using the transactions service. This chapter is relevant to C++, COBOL and PL/I programmers.

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 of discussion how to implement `CosTransactions::Resource` objects on top of the standard X/Open XA interface to manage transactional resources.

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 Orbix OTS interoperates with older releases of Orbix and with other OTS implementations, including the Orbix 3 OTS.

Chapter 10 discusses the plug-ins that implement the transaction service, and the options for deploying them.

Chapter 11 introduces configuration for the OTS RRS plug-in.

Chapter 12 discusses general configuration items for the OTS RRS plug-in.

Chapter 13 discusses configuration items specific to the OTS RRS plug-in.

Chapter 14 discusses how to use the OTS RRS transaction manager.

Appendix A discusses troubleshooting through the use of RRS panels.

Appendix B provides an introduction on how to set up for management using the Administrator Web Console.

Additional resources

The Knowledge Base contains helpful articles, written by experts, about Orbix Mainframe, and other products:

https://community.microfocus.com/t5/Orbix/ct-p/Orbix

If you need help with Orbix Mainframe or any other products, contact technical support:

https://www.microfocus.com/en-us/support/

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.

Part 1

C++ Programming

In this part

This part contains the following chapters:

Note: "Transaction Service" on page 1 is relevant to C++, COBOL and PL/I programmers. The rest of the chapters in this part are relevant only to C++ programmers.

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. Integration with X/Open XA compliant resource managers is described.

Note: This chapter is relevant to C++, COBOL and PL/I programmers. Chapters 2-9 are relevant only to C++ programmers.

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).
- Integration with resource managers supporting the X/Open XA interface.
- 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 or 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 XA interfaces an application can be easily extended to support multiple resource managers within a transaction easily.

CHAPTER 2

OMG OTS and X/Open XA Interfaces

The OMG OTS provides interfaces to manage the demarcation of transactions and the propagation of transaction contexts. With the X/Open XA interface, integration with compliant resource managers such as databases and message queues is provided.

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.

With the X/Open XA interface, integration with compliant resource managers such as databases and message queues is provided.

Illustration of transaction interfaces

Figure 1 shows these areas of transaction management.

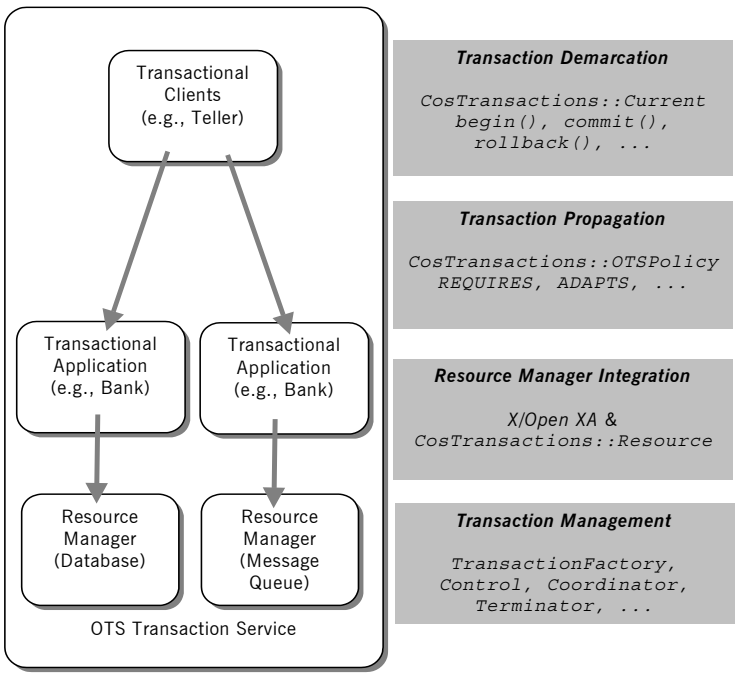

Figure 1: *OTS and XA*

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. 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. 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 XA 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*, *C++*.

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 36 and "Explicit Propagation" on page 58) for more details.

The X/Open XA Interface

XA Interfaces

The X/Open XA interface is a C API between a transaction manager and a resource manager (for example, a database). The C API provides functions for opening and closing connections to the resource manager (`xa_open()` and `xa_close()`), managing associations between the current connection and global transactions (`xa_start()` and `xa_end()`), transaction protocols (`xa_prepare()`, `xa_commit()`, `xa_rollback()` and `xa_forget()`), and functions to support recovery (`xa_recover()`).

Integration with OTS

Integration between XA compliant resource managers and the OTS is provided by several interfaces in the XA module, as detailed in the following table.

Table 2: *XA Interfaces*

Interface	Purpose
Connector	Provides a means of getting `CurrentConnection` and `ResourceManager` objects.
CurrentConnection	Represents the current XA connection to a resource manager.
BeforeCompletionCallback	Allows an application to be called before the completion of a transaction.
ResourceManager	Use to register and unregister `BeforeCompletionCallback` 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 XA-compliant database and use SQL to access the database. This example shows the source and destination accounts using different databases, however they could both be using the same database.

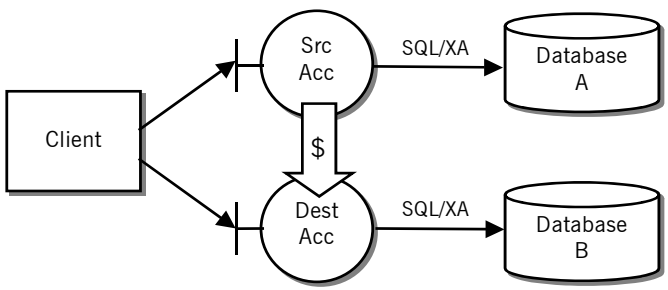

Figure 2: Example OTS Application – Funds Transfer

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:

```
// C++
Bank::Account_var src_acc = ...
Bank::Account_var dest_acc = ...
Bank::CashAmount 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 16.
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 19
3. The implementation of the account objects must be integrated with an XA compliant database. This is covered in "XA Resource Manager Integration" on page 21.

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")`.

Note that the file CosTransactions.hh must be included to use the interfaces defined in the CosTransactions module. Error handling has been omitted for clarity:

```
// C++
...
#include <CosTransactions.hh>
...
int main(int argc, char** argv)
{
  CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);

  CORBA::Object_var obj =
    orb->resolve_initial_references("TransactionCurrent");
  CosTransactions::Current_var tx_current =
    CosTransactions::Current::_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:

```
// C++
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:

```
// C++
Bank::Account_var src_acc = ...
Bank::Account_var dest_acc = ...
Bank::CashAmount 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:

```
// C++
try {
  tx_current->commit(IT_false)
} catch (CORBA::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 101 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:

```
// C++
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.

```
// C++

CORBA::ORB_var orb = ...

// Create a policy object for the REQUIRES OTS Policy.
CORBA::Any policy_val;
policy_val <<= CosTransactions::REQUIRES;
CORBA::Policy_var tx_policy =
    orb->create_policy(CosTransactions::OTS_POLICY_TYPE,
                       policy_val);

// Add OTS policy to policy list (just 1 policy in this case).
CORBA::PolicyList policies(1);
policies.length(1);
policies[0] = CORBA::Policy::_duplicate(tx_policy);

// Get a reference to the root POA.
CORBA::Object_var obj =
    orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
    PortableServer::POA::_ narrow(obj);

// Create a new POA with the OTS Policy.
PortableServer::POA_var 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(...);
PortableService::ObjectId_var id =
  tx_poa->activate_object(servant);
obj = tx_poa->servant_to_reference(servant);
Bank::Account_var account = Bank::Account::_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.

XA Resource Manager Integration

Process of using an XA Resource Manager

Integrating an XA compliant resource manager with OTS managed transactions involves three steps:

1. Setting up configuration variables for the resource manager.
2. Application initialization.
3. Accessing the database during an OTS transaction.

Full details are in Chapter 6.

Resource Manager Configuration

Each resource manager used by an application requires configuration. The configuration is placed in a namespace that is passed to the `create_resource_manager()` operation during application initialization. The minimum configuration is the specification of the resource manager's open-string. This is a resource manager specific string that is passed to the `xa_open()` call and contains sufficient information to create an XA connection to the database. For example this can contain user name and password details.

The following example shows the configuration for an Oracle database using the `xa_resource_managers:oracle` namespace. The `thread_model` configuration variable specifies scope of an XA connection (either thread or process):

```
xa_resource_managers:oracle:thread_model = "PROCESS";
xa_resource_managers:oracle:open_string =
   "Oracle_XA+Acc=P/SCOTT/TIGER+SesTm=60+SqlNet=osol"
```

Application Initialization

Applications using XA resource managers must include the file `omg/XA.hh` to access the interfaces in the XA module. During application initialization `ResourceManager` and `CurrentConnection` objects are created to represent

the resource manager being integrated. This is done by getting a reference to the Connector object (by passing "XAConnector" to `resolve_initial_references()`) and calling `create_resource_manager()`:

```
// C++
...
CORBA::ORB_var orb = ...

// Get reference to the XAConnector object.
CORBA::Object_var xa_connector_obj =
  orb->resolve_initial_references("XAConnector");
  XA::Connector_var xa_connector =
  XA::Connector::_narrow(xa_connector_obj);

// Get XA Connection object for the resource manager.
XA::CurrentConnection_var current_connection;
XA::ResourceManager_var rm =
  xa_connector->create_resource_manager(
    "xa_resource_managers:oracle",
    xaosw, "",
    current_connection);
```

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).

Accessing the Database within an OTS Transaction

The application 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 the `start()` operation must be called on the XA Connection object to associate the connection with the current transaction.
2. After the database access the `end()` operation must be called on the XA Connection object to remove the association with the current transaction.
3. Resource manager operations related to transaction management such as the embedded SQL operations BEGIN, COMMIT, or ROLLBACK must not be used.

The following shows how integration with an XA-compliant database is achieved using embedded SQL:

```
// C++
void AccountImpl::deposit(float amt)
{
  // Get the coordinator and otid for the current
  // transaction.
  CosTransactions::Current_var tx_current = ...
  CosTransactions::Control_var control =
    tx_current->get_control();
  CosTransactions::Coordinator_var tx =
    control->get_coordinator();
  CosTransactions::PropagationContext_var ctx =
    tx->get_txcontext();
  const CosTransactions::otid_t& otid = ctx->current.otid;

  // Associate current transaction with the XA connection
  // to the database.
  XA:CurrentConnection_var current_connection = ...
  current_connection->start(tx, otid);

  EXEC SQL BEGIN DECLARE SECTION
    unsigned long acc_id  = m_accId;
    float         balance = 0.0;
  EXEC SQL END DECLARE SECTION

  // Get the current balance.
  EXEC SQL SELECT BALANCE
    INTO :balance
    FROM ACCOUNTS
    WHERE ACC_ID = :acc_id;

  // Update balance.
  balance += amt;
  EXEC SQL UPDATE ACCOUNTS
    SET BALANCE = :balance
    WHERE ACC_ID = :acc_id;

  // Dissociate the current transaction from the XA
  // connection to the database.
  current_connection->end(tx, otid, IT_true);
}
```

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. 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`).

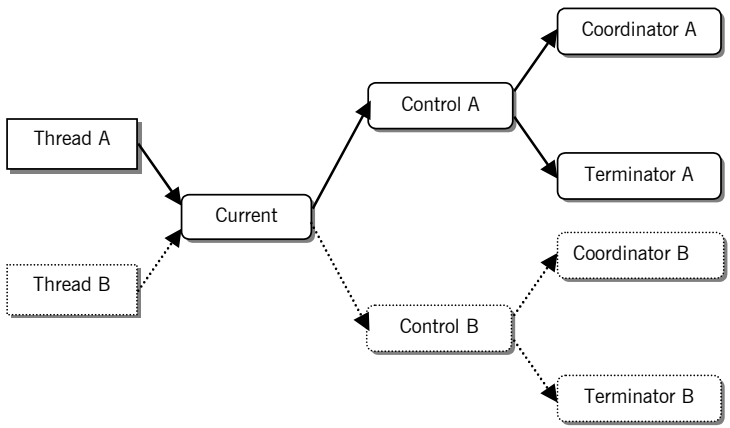

Figure 3: *Thread and Transaction Associations*

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:

```
// C++
CosTransactions::Current_var tx_current;
try {
  CORBA::ORB_var orb = ...
  CORBA::Object_var obj =
    orb->resolve_initial_references("TransactionCurrent");

  tx_current = CosTransactions::Current::_narrow(obj);
}
catch (CORBA::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:

```
// C++
CosTransactions::Current_var tx_current = ...
try
{
  tx_current->begin();
}
catch (CosTransactions::SubtransactionsUnavailable& ex)
{
  // Already in a transaction and nested transaction are not
  // supported.
}
catch (CORBA::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:

```
// C++
CosTransactions::Current_var tx_current = ...
try
{
  // Attempt to commit the current transaction.
  tx_current->commit(IT_false);
}
catch (CORBA::TRANSACTION_ROLLEDBACK&)
{
  // The transaction was rolled back.
}
catch (CORBA::SystemException& ex)
{
  // Error handling...
}
catch (CosTransactions::NoTransaction& ex)
{
  // 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 101 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:

```
// C++
CosTransactions::Current_var tx_current = ...
try
{
  tx_current->rollback();
}
catch (CORBA::SystemException& ex)
{
  // Error handling...
}
catch (CosTransactions::NoTransaction& ex)
{
  // 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:

```
// C++
CosTransactions::Current_var 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:

```
// C++
CosTransactions::Current_var tx_current = ...
tx_current->begin();
account->deposit(100.0);

// Suspend the current transaction.
CosTransactions::Control_var control =
  tx_current->suspend();

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

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

tx_current->commit(IT_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

Direct Transaction Demarcation

Using the transaction factory to create transactions

The alternative to using the OTS `Current` object is to use the transaction factory directly to create transactions.

Example

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

```
// C++
//
// Get a reference to the transaction factory.
CORBA::ORB_var orb = ...
CORBA::Object_var obj =
  orb->resolve_initial_references("TransactionFactory");
CosTransactions::TransactionFactory_var tx_factory =
  CosTransactions::TransactionFactory::_narrow(obj);

// Create a transaction with a timeout of 60 seconds.
CosTransactions::Control_var 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:

```
// C++
//
try {
  CosTransactions::Terminator_var term =
    control->get_terminator();
  term->commit(IT_true);
} catch (CORBA::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, 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 56 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 52; 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 45.

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 on Using Policies in the *CORBA Programmer's Guide*, *C++*. For an example of using a `NonTxTargetPolicy` see "Example Use of a NonTxTargetPolicy" on page 47.

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.

```
// C++
//
// Create CORBA policy object for ADAPTS OTSPolicy
CORBA::Any tx_policy_value;
tx_policy_value <<= CosTransactions::ADAPTS;

CORBA::ORB_var orb = ...
CORBA::Policy_var tx_policy = orb->create_policy(
  CosTransactions::OTS_POLICY_TYPE, tx_policy_value);

// Create a POA using the transactional policy.
CORBA::PolicyList policies(1);
policies.length(1);
policies[0] = CORBA::Policy::_duplicate(tx_policy)

// Get a reference to the root POA.
CORBA::Object_var obj =
orb->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa = PortableServer::_narrow(obj);
```

```
// Set up nil POAManager reference.
PortableServer::POAManager_var nil_mgr =
PortableServer::POAManager::_nil();

PortableServer::POA_var tx_poa =
root_poa->create_POA("TX ADAPTS", nil_mgr, 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(...);

PortableServer::ObjectId_var id =
  tx_poa->activate_object(servant);

obj = tx_poa->servant_to_reference(servant);
Account_var account = Account::_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:

```
// C++
//
// Get reference to PolicyCurrent object.
CORBA::ORB_var orb = ...
CORBA::Object_var obj =
  orb->resolve_initial_references("PolicyCurrent");

CORBA::PolicyCurrent_var policy_current =
  CORBA::PolicyCurrent::_narrow(obj);

// Create PERMIT NonTxTarget policy.
CORBA::PolicyList policy_list(1);
policy_list.length(1);

CORBA::Any tx_policy_value;
tx_policy_value <<= CosTransactions::PERMIT;

policy_list[0] = orb->create_policy(
    CosTransactions::NON_TX_TARGET_POLICY_TYPE,
    tx_policy_value);

// Set policy overrides.
policy_current->set_policy_overrides(policy_list,
                                     CORBA::ADD_OVERRIDE);

// Invoke on target object
CosTransctions::Current_var tx_current = ...
Account_var account = ...
AuditLog_var log = ...

tx_current->begin();
account->deposit(100.00);
log->append("User ... deposited 100 to account ...");
tx_current->commit(IT_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):

```
// C++
void AccountImpl::deposit(float amount)
{
  CosTransactions::Current_var tx_current = ...

  // Test if a transaction was propagated from the client.
  CosTransactions::Control_var control =
    tx_current->get_control();

  if (CORBA::is_nil(control))
  {
    // No current transaction, so create one.
    tx_current->begin();
  }

  // Do the transactional work
  ...

  // If a local transaction was created, commit it.
  if (CORBA::is_nil(control))
  {
    tx_current->commit(IT_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):

```
// C++
// Deposit money into a single account (no transaction
// needed).
Account_var acc = ...
acc->deposit(100.00);

// Transfer money between two account (this requires a
// transaction)
Account_var src_acc = ...
Account_var dest_acc = ...
CosTransactions::Current_var tx_current = ...

tx_current->begin();
src_acc->withdraw(200.00);
dest_acc->deposit(200.00);
tx_current->commit(IT_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 52.
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 53.

Automatic Transactions

The `ADAPTS` OTSPolicy (see "Use of the ADAPTS OTSPolicy" on page 50) 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 include the file <orbix/cos_transactions.hh> and narrow the OTS current object to the `IT_CosTransactions::Current interface`.

```
// C++
CosTransactions::Current_var tx_current = ...

IT_CosTransactions::Current_var it_tx_current =
  IT_CosTransactions::Current::_narrow(tx_current);

Account_var 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(IT_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 3: *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 3; 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:

```
// C++
CosTransactions::TransactionFactory_var tx_factory = ...
CosTransactions::Control_var control =
  tx_factory->create(60);

Bank::Account_var src_acc = ...
Bank::Account_var dest_acc = ...
Bank::CashAmount amount = 100.0;
src_acc->withdraw(amount, control);
dest_acc->deposit(amount, control);

CosTransactions::Terminator_var term =
  control->get_terminator();
term->commit(IT_true);
```

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

CHAPTER 6

Using XA Resource Managers with OTS

This chapter describes how to integrate with transactional systems by implementing CosTransactions::Resource objects on top of the standard X/Open XA interface.

In this chapter

This chapter discusses the following topics:

The XA Interface

Resource objects

To use a transactional system (such as a database system) with the transaction service, you must connect the transactions provided by the transactional system to the distributed transactions managed by the transaction service. With the transaction service, this is achieved by implementing `CosTransactions::Resource` objects — each resource represents a local transaction in the transactional system — and registering these Resource objects with the distributed transactions.

Because many systems provide a standard interface to their transactional capabilities — the X/Open XA interface — you can implement `CosTransactions::Resource` objects on top of the XA interface, and provide an easy-to-use integration with the transaction service. This is precisely what the Orbix XA plug-in provides.

XA Overview

XA (X/Open CAE Specification, Distributed Transaction Processing: The XA specification, December 1991, ISBN: 1 872630 24 3) specifies a standard C API provided by transactional systems (called Resource Managers in the XA specification) that want to participate in distributed transactions

managed by transaction managers developed by other vendors. XA defines a set of C-function pointers, and a C-struct that holds these function pointers, `xa_switch_t` (see `orbix_sys/xa.h`):

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

Function pointers

Each XA Resource Manager must provide a global instance of `xa_switch_t`. For example, Oracle's global `xa_switch_t` instance is called `xaosw`.

The function pointers provided by this `xa_switch_t` instances can be divided into four categories:

- Functions to connect and disconnect to the XA Resource Manager:`xa_open()` and `xa_close()`. The string passed to `xa_open()` typically contains connection information, e.g. a database name and a username and password.
- Transaction completion functions `xa_prepare()`, `xa_commit()`, `xa_rollback()`, `xa_forget()` correspond to the `CosTransactions::Resource` operations.

- Recovery function `xa_recover()` is currently not used by the XA plug-in.
- Functions used to start and end associations between connections and a transactions: `xa_start()`, `xa_end()`

  In order to use an XA connection to do some work within a distributed transaction, it is necessary to create an association between this connection and the distributed transaction. `xa_start()` is used to create such an association; `xa_end(TMSUSPEND)` suspends the association, without releasing the connection; `xa_start(TMRESUME)` resumes a suspended association; `xa_end(TMSUCCESS)` terminates an association with success; and `xa_end(TMFAIL)` terminates an association and marks the transaction rollback-only.

Note: `xa_complete()` is only used for asynchronous XA, an optional part of XA which is not supported by any popular XA implementation.

XA and Multi-Threading

In the XA specification, the scope of an XA connection is called thread-of-control. Each thread-of-control can only use the connections that it has established (using `xa_open()`). The XA specification maps thread-of-control to operating system process (2.2.8). Each thread in a process has access to all the XA connections established by this process. This is clearly specified in the JTA specification (XA for Java).

Unfortunately, for the C XA API, most vendors implement the following:

- a thread-unsafe mode, in which the scope of each XA connection is the process (XA thread-of-control maps to process)
- a thread-safe mode, in which the scope of each XA connection is the thread by which is was created (XA thread-of-control maps to thread)

For example, with Oracle, the `+threads={true,false}` option of the OracleXA open string lets the application programmer choose between these two modes.The thread-of-control equal thread model sometimes simplifies the API used to access the data. For example, Oracle embedded SQL in C/C++ (Pro*C/C++) has a notion of a default database connection for each thread of control.

When the model is thread-of-control equal process, and a process has a pool of connections to the same database, it is necessary to explicitly specify which connection to use (with an Oracle AT clause):

```
EXEC SQL AT :db_name INSERT VALUES(123, 43, 3.49) INTO
   SALE_DETAILS;
```

But when the model is thread-of-control equal thread, and each thread has one connection to a given database, there is no need to explicitly specify the connection to use (no AT clause):

```
EXEC SQL INSERT VALUES(123, 43, 3.49) INTO SALE_DETAILS;
```

The EXEC SQL statements used in a multi-threaded multi-connection application look very much like the EXEC SQL statement used in a single-threaded single-connection application.

The main drawback of tying connection and threads is flexibility since it prevents the application from managing connections independently of threads, which limits the kind of connection pooling that can be implemented. Also, a CORBA server typically dispatches different requests to different threads: the thread-of-control equal thread model prevents the

use of `xa_end(TMSUSPEND)` at the end of a request and `xa_start(TMRESUME)` at the beginning of the next request in the same transaction, since an association must be resumed by the thread of control from which it was suspended.

Using the Orbix XA Plug-In

The Orbix XA plug-in implements and manages `CosTransactions::Resource` objects on behalf of the application. It supports the two thread-of-control models described in the previous paragraph: when the thread model is `XA::PROCESS`, it uses a single-threaded persistent POA to host its `CosTransactions::Resource` servants. When the thread model is `XA::THREAD`, it uses a multi-threaded persistent POA.

You access the XA plug-in by obtaining a reference to the `XA::Connector` local object through `resolve_initial_references()`:

```
#include <omg/xa.hh>
CORBA::Object_var xa_connector_obj =
   orb- >resolve_initial_references("XAConnector");
XA::Connector_var xa_connector =
   XA::Connector::_narrow(xa_connector_obj);
```

Then you create an `XA::ResourceManager`, by calling `create_resource_manager` on the connector. This operation creates a persistent POA that hosts the resource manager's servant and will host the `CosTransactions::Resource` servants. The `create_resource_manager` operation also returns an `XA::CurrentConnection` local object, which

establishes (with `xa_open()`) connections when needed, and lets you start, suspend, resume, and end associations between any transaction and the current XA thread of control's connection.

```
XA::CurrentConnection_var current_connection;
XA::ResourceManager_var rm =
   xa_connector->create_resource_manager(
      "xa_resource_managers:oracle",
      // the name of an Orbix configuration namespace
      xaosw,       // XA switch
      "",          // empty open-string, i.e. the unsecured
                   // open-string is specified in configuration
      "",          // empty close-string, i.e. the unsecured
                   // close-string is specified in the
   configuration
      XA::PROCESS, // thread-model
      false,       // no automatic association
      false,       // do not use dynamic registration
      current_connection // (out) current connection local object
   );
```

The first parameter of `create_resource_manager` is the name of an Orbix configuration namespace; this configuration namespace defines the name of the resource manager persistent POA (defaults to the given namespace name), the open string when the `open_string` parameter is empty, the close string when the `close_string` parameter is empty, and various other properties. The resource manager id can also be set in the configuration using the `rmid` variable. When the `rmid` variable is set, the XA integration uses the value as the `rmid` passed to `xa_open()` and all subsequent `xa_` calls. When the `rmid` variable is not set, the XA integration generates a new `rmid` value for each `CurrentConection` object.

Associations between Transactions and Connections

The `CurrentConnection` local interface is defined in the `XA` module as follows:

```
enum ThreadModel { PROCESS, THREAD };
local interface CurrentConnection
{
    void
    start(
       // xa_start(TMNOFLAGS) or xa_start(TMJOIN)
       in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid
       );
    void
    suspend(
       // xa_end(TMSUSPEND)
       in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid
       );
    void resume(
       // xa_start(TMRESUME)
       in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid
       );
    void end(
       // xa_end(TMSUCCESS) or xa_end(TMFAIL)
       in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid,
       in boolean success
       );
    ThreadModel thread_model();
    long rmid();
};
```

When the thread model is PROCESS, `xa_open()` is called by the first start call or the first operation performed by a Resource servant; and `xa_close()` is called during shutdown. When the thread model is THREAD, `xa_open()` is called the first time a thread calls `CurrentConnection::start`, or any operation on a Resource servant; `xa_close()` is called when this thread exits.

In order to do some work within a distributed transaction with a given resource manager, you have to associate the resource manager's current connection with this transaction, by calling `CurrentConnection::start`:

```
// assuming the OTS transaction is associated with the current
// thread
    CosTransactions::Control_var control =
       tx_current->get_control();
    CosTransactions::Coordinator_var tx =
       control->get_coordinator();
    CosTransactions::PropagationContext_var ctx =
       tx->get_txcontext();
    const CosTransactions::otid_t& otid = ctx->current.otid;
    current_connection->start(tx, otid);
```

The first time `CurrentConnection::start()` is called with a given transaction, the XA plug-in creates a `CosTransactions::Resource` persistent object and registers this object with the transaction coordinator.

Once you have finished using a connection, it is critical to end the association with the transaction for two reasons:

- It releases the connection, and makes it available for other transactions
- As long as any connection is associated with a transaction, this transaction cannot be committed. Some systems (e.g. Oracle) don't even allow to roll back a transaction while it is associated with any connection.

The recommended way to start and end (or start/suspend/resume/suspend...) an association is to use a helper C++ class: the helper class constructor creates the association by calling start, and the helper class destructor ends the association. The multi-threaded transfer demo provides a helper Association class which uses start and end; the single-threaded farm demo provides a helper Association class which uses start, suspend and resume.

Association State Diagram

Figure 4 shows the state diagram of an association between a transaction and an XA connection. In this diagram all start, suspend, resume, and end calls are successful (they do not raise any exception). When start, suspend, resume or end raises `CORBA::INTERNAL` with the minor code `IT_XA_MinorCodes::INTERNAL::XAER_RMFAIL_` the new state is non-existant. When resume, suspend or end raises `CORBA::TRANSACTION_ROLLEDBACK` with the minor code `IT_XA_MinorCodes::TRANSACTION_ROLLEDBACK::XA_RB_`, the new state is non-existant. When end raises `CORBA::TRANSACTION_ROLLEDBACK` with the minor code `IT_XA_MinorCodes::TRANSACTION_ROLLEDBACK::DEFERRED_ROLLBACK`, the new state is non-existant. For every other exception raised by start, suspend, resume and end, there is no state transition.

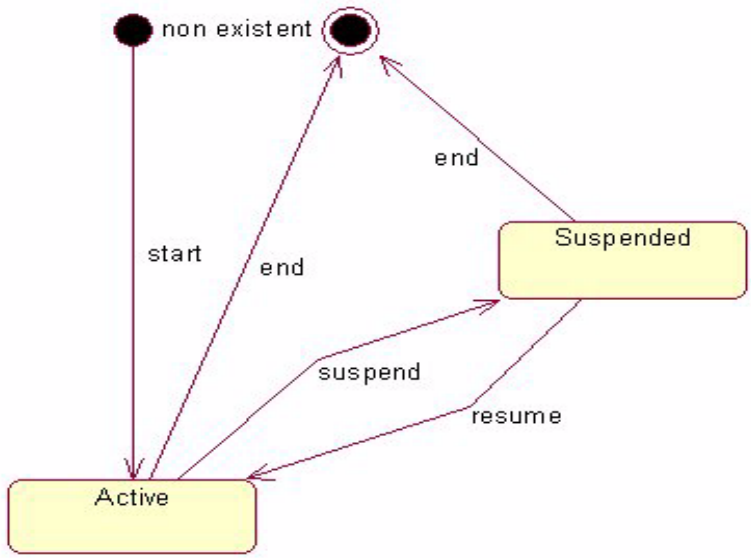

Figure 4: *Association State Diagram*

Using a Remote Resource Manager

The Resource servants and the application logic that performs the transactional data access (for example, through embedded SQL in C/C++ calls) do not need to be in the same process. You use the operation `Connector::connect_to_resource_manager` to connect to a remote `XA::ResourceManager`:

```
XA::CurrentConnection_var current_connection =
   xa_connector->connect_to_resource_manager(
   "xa_resource_managers:oracle",
// the name of an Orbix configuration namespace
   rm,     // object reference to an XA::ResourceManager object
   xaosw,  // XA switch
   "",     //open string (empty string means that the actual open
           // string is in configuration)
   "",     //close string (empty string means that the actual
   close
           // string is in configuration)
   XA::PROCESS, // thread-model
   false,       // no automatic association
   false,       // do not use dynamic registration
);
```

Some systems (e.g. Oracle) even allow you to create associations between a given transaction and connections to the same database established by different processes: this is referred to as tightly coupled threads in the XA specification.

Using a remote resource manager is particularly useful for single-threaded servers, because it allows you to make a data-access server available for other transactions as soon as the transaction has finished with this server (before the completion of the transaction). See the farm demonstration.

Before Completion Callback

You can register with a resource manager any number of BeforeCompletionCallback objects:

```
interface BeforeCompletionCallback
{
    void
    before_completion(
       in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid,
       in boolean success
   );
};
interface ResourceManager
{
    unsigned long register_before_completion_callback(
       in BeforeCompletionCallback bcc);
    void unregister_before_completion_callback(
       in unsigned long key);
};
```

The before completion callbacks objects are called by the Resource servant before `prepare`, `commit_one_phase`, and `rollback` on a non-prepared transaction. If any of these before completion callbacks calls raise an exception, the transaction is rolled back. A typical use of the `BeforeCompletionCallback` is to end a suspended association in a single-threaded server. See the farm demonstration.

Asynchronous Rollback Support

An XA implementation may or may not support asynchronous rollbacks,that is `xa_rollback()` may or may not be called on a transaction while this transaction is actively associated with some connection. This is typically not documented by the XA implementation — OracleXA does not support asynchronous rollbacks, while SybaseXA does.

When you set `supports_async_rollback` to `false` and use a remote resource manager, the XA plug-in uses a transient object to handle asynchronous rollbacks (by deferring them until the association is ended). This transient object is hosted by the root POA, so you have to activate the root POA manager.

Ping Period

The Resource Manager can periodically check that the transactions with which the Resource servants it manages were registered are still alive by calling `get_status` on their respective coordinators. When a call to `get_status` fails (that is, it raises any exception), and the associated Resource is not prepared, this Resource is immediately rolled back.

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 an XA 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 XA 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 XA 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:

```
// C++
//
// Get the control and coordinator object for the
// current transaction.
//
CosTransactions::Current_var tx_current = ...
CosTransactions::Control_var control =
    tx_current->get_control();
CosTransactionsCoordinator_var 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();
PortableServer::POA_var poa = ...
CosTransactions::Synchronization_var obj =
    sync_servant->activate(poa);

//
// Register the synchronization once with the transaction
//
coord->register_synchronization(obj);
```

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 4: *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 4: *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 87 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:

```
// C++
CosTransactions::Coordinator_var coord = ...
CosTransactions::Status status = coord->get_status();
if (status == CosTransactions::StatusActive)
{
  ...
} else if (status == CosTransactions::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 5: *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 5: *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:

```
// C++
CosTransactions::Coordinator_var c1 = ...
CosTransactions::Coordinator_var 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:

```
// C++
CosTransactions::Control_var foreign_control = ...
CosTransactions::Coordinator_var foreign_coord =
  foreign_control->get_coordinator();
CosTransactions::PropagationContext_var ctx =
    foreign_coord->get_txcontext();

CosTransactions::TransactionFactory_var tx_factory = ...
CosTransactions::Control_var 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 5 shows a high level picture of how clients, applications, the OTS and resource objects interoperate to achieve the ACID properties.

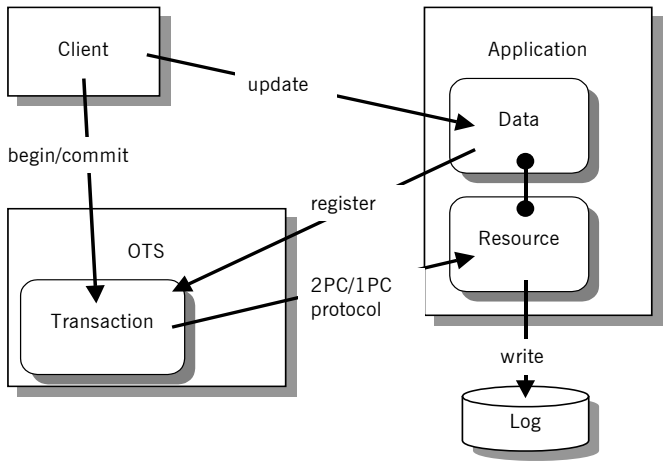

Figure 5: *Relationship between resources and transactions*

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:

```
// C++
class ResourceImpl : public POA_CosTransactions::Resource
{
  public:

   ResourceImpl();

   virtual ~ResourceImpl();

    CosTransactions::Vote
    prepare()
    throw (CORBA::SystemException,
           CosTransactions::HeuristicMixed,
           CosTransactions::HeuristicHazard);

    void
    rollback()
    throw (CORBA::SystemException,
           CosTransactions::HeuristicCommit,
           CosTransactions::HeuristicMixed,
           CosTransactions::HeuristicHazard);

    void
    commit()
    throw(CORBA::SystemException,
          CosTransactions::NotPrepared,
          CosTransactions::HeuristicRollback,
          CosTransactions::HeuristicMixed,
          CosTransactions::HeuristicHazard);
```

```
    void
    commit_one_phase()
    throw(CORBA::SystemException,
          CosTransactions::HeuristicHazard);

    void
    forget()
    throw (CORBA::SystemException);
};
```

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. For more details see the sections on Setting Object Lifespan and Assigning Object IDs in the chapter on Managing Server Objects in the *CORBA Programmer's Guide, C++* 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 81 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:

```
// C++
CosTransactions::Current_var tx_current = ...

// Get the transaction's coordinator object.
CosTransactions::Control_var control =
  tx_current->get_control();
CosTransactions::Coordinator_var 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.
PortableServer::POA_var resource_poa = ...
PortableServer::ObjectId_var oid = ...

resource_poa->activate_object_with_id(oid, servant);

CORBA::Object_var obj =
  resource_poa->servant_to_reference(servant);

CosTransactions::Resource_var resource =
  CosTransactions::Resource::_narrow(obj);

// Register the resource with the transaction coordinator.
CosTransactions::RecoveryCoordinator_var 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 104). 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 6 shows a transaction with two registered resource objects being rolled back after a timeout.

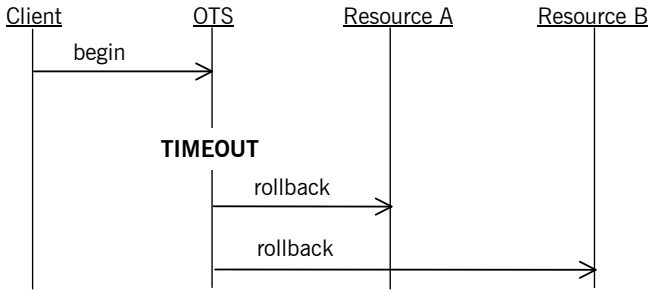

Figure 6: *Rollback after a timeout*

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 99).

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 7 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.

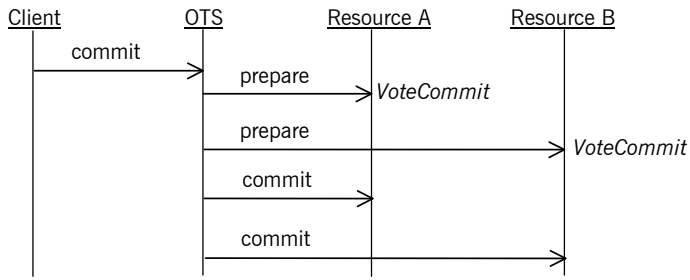

Figure 7: *Successful 2PC protocol with two resources*

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 8 shows VoteRollback being returned by one resource which results in the other resource being told to rollback.

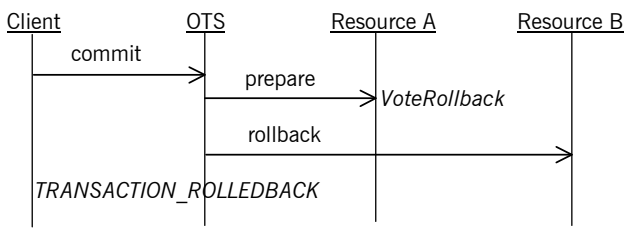

Figure 8: *Voting to rollback the transaction.*

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 9 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).

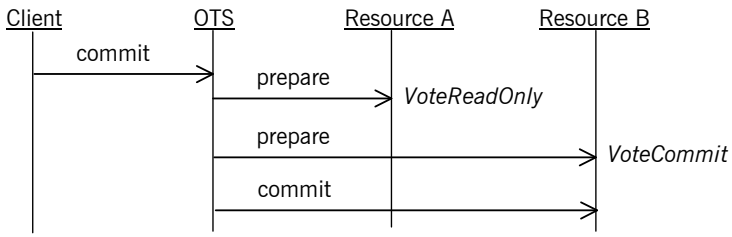

Figure 9: *A resource returning VoteReadOnly.*

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 10 shows a successful 1PC protocol.

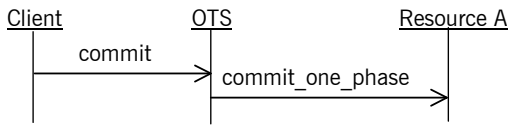

Figure 10: *A successful 1PC protocol.*

Figure 11 shows a 1PC protocol resulting in the transaction being rolled-back.

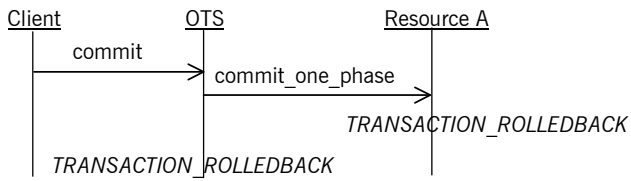

Figure 11: *The 1PC protocol resulting in a rollback.*

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 6: *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 12). 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.

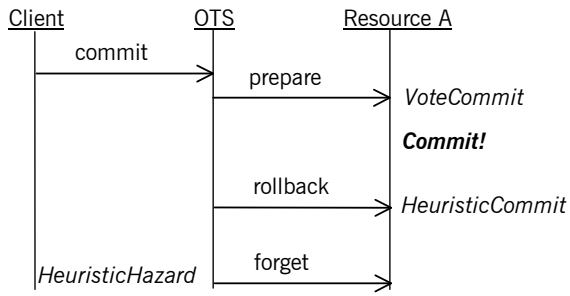

Figure 12: *Raising the HeuristicCommit exception*

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 13 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.

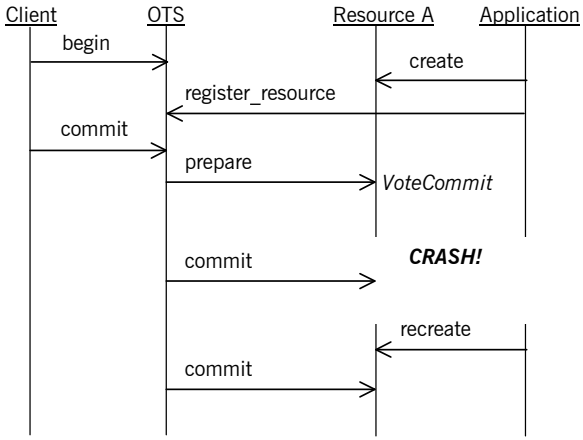

Figure 13: *Recovery after the failure of a resource object*

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 93 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 14). 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.

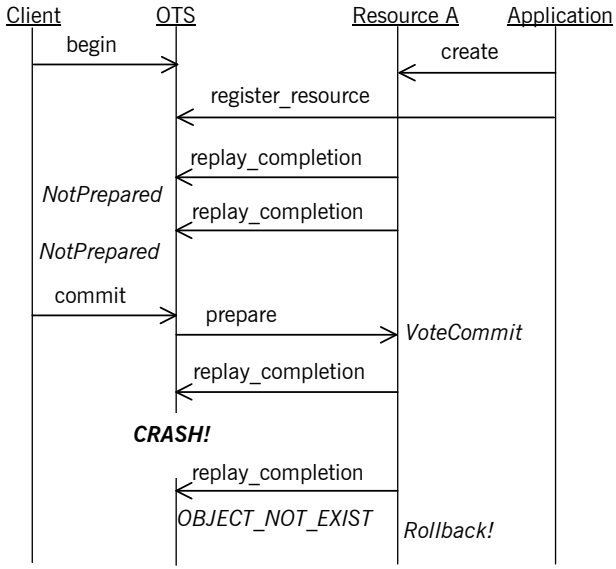

Figure 14: *Use of the replay_completion() operation*

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 56 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 7 details this mapping:

Table 7: *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.

Note: Orbix 3 has deprecated support for OTS since Orbix 3.3.13. The information in this section applies to older versions of Orbix 3.

Orbix 3 OTS Interoperability

Figure 15 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 RRS (see "The OTS RRS Transaction Manager" on page 131) 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.

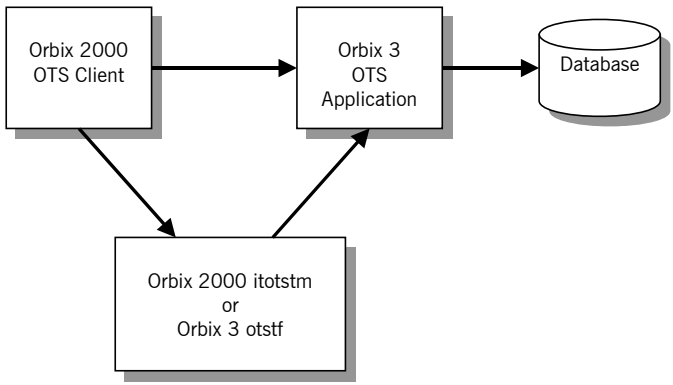

Figure 15: *Interoperability with Orbix 3 OTS Applications*

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 on Resolving Names with corbaname in the chapter on the Naming Service in the *CORBA Programmer's Guide, C++*).

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.

Bypassing otstf

It is possible to bypass the use of the otstf server and use the transaction factory provided by the Orbix 3 OTS application. This is done by modifying the Orbix 3 application to publish its internal transaction factory reference. This is illustrated in the following code:

```
// Orbix 3 OTS C++ Application Code
CORBA::ORB_var orb = ...
OrbixOTS::Server_var ots = ...

// Get reference to the local transaction factory.
CosTransactions::TransactionFactory_var tx_factory =
  ots->get_transaction_factory_reference();

// Publish reference (eg, to the name service or a file)
```

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"`.

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 16.

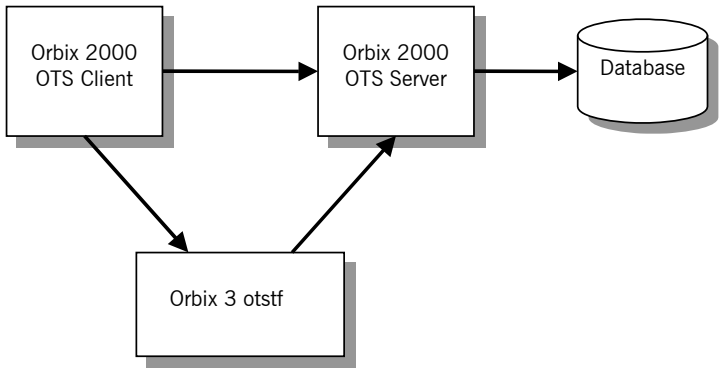

Figure 16: *Using and alternative OTS Implementation*

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

Part 2

Administration

In this part

This part contains the following chapters:

Note: All of these chapters are relevant regardless of which programming language is being used for application development.

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. Additionally, there are three OTS transaction manager implementations: OTS Lite, which provides a lightweight transaction coordinator supporting only the 1PC protocol; OTS Encina, which provides full recoverable 2PC support in non-mainframe environments; and OTS RRS, which provides full recoverable 2PC support in mainframe environments. This chapter discusses deployment options.

In this chapter

This chapter discusses the following topics:

The itotstm Transaction Manager Service page 132

Note: Because OTS Encina is not supported by Orbix Mainframe, it is not discussed in this chapter.

Overview

Overview

This section provides an overview of the OTS plug-ins that Orbix Mainframe supports. It 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 supported by Orbix Mainframe:

- OTS Lite
- OTS RRS

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 RRS

OTS RRS provides full recoverable 2PC support in mainframe environments. It can be used by the following:

- CICS transactions initiating two-phase commit processing.
- IMS transactions initiating two-phase commit processing.
- C++ programs requiring two-phase commit processing in z/OS and z/OS UNIX System Services.

It is available as a standalone service and as an application plug-in.

Features in OTS

Table 8 shows the features supported by these pieces.

Table 8: *Features in OTS Implementation*

Feature	Generic OTS	OTS Lite	OTS RRS
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
Web Console Management		N	Y
XA Support		Y	Y
Application Plug-In	Y	Y	Y

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 link with the OTS plug-in library.

The 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 17). This allows different deployment options to be easily supported through configuration.

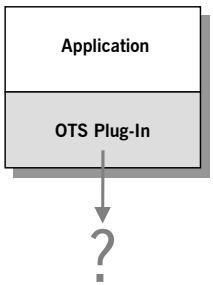

Figure 17: *The Generic OTS Plug-In*

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 three possible deployment scenarios for C++:

- Using the OTS Lite plug-in when only 1PC transactions are required.
- Using the itotstm service with the OTS RRS plug-in where recoverable 2PC transactions are required.
- Using the OTS RRS plug-in loaded into the application itself.

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

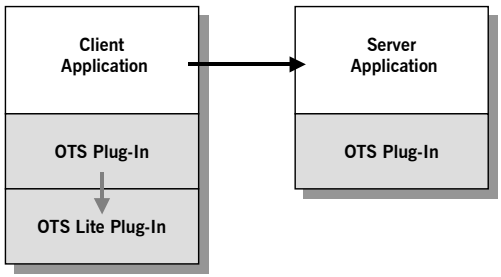

Figure 18: *Deployment using the OTS Lite Plug-In*

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.

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 OTS RRS Transaction Manager

Overview

The OTS RRS Transaction Manager provides full recoverable 2PC transaction coordination for applications running on the mainframe.

There are two ways in which the OTS RRS Transaction Manager may be used:

- By configuring the itotstm service to load the OTS RRS plug-in.
- By loading the OTS RRS plug-in directly into the application.

Configuring the OTS RRS Plug-In

Various administration steps must be performed before you can successfully use the OTS RRS plug-in, regardless of whether it is used in the itotstm service or directly in the application.

Note: If you ran the *orbixhlq*.JCLLIB(DEPLOY3) JCL to deploy OTS RRS on the mainframe, the required administration steps are performed automatically.

Two transient POAs must be created. These serve as namespace POAs based on which the OTS RRS 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_rrs:namespace_poa configuration item. The default value of this configuration variable is "otstm" for the itotstm service, and "RRS" 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
```

The minimum configuration required to load the OTS RRS plug-in into an application is as follows:

```
<app-scope> {
    initial_references:TransactionFactory:plugin = "ots_rrs";
    plugins:ots_rrs:namespace_poa = "<name>";
}
```

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 RRS OTS plug-in to provide 2PC transactions for an application.

Deploying on the mainframe

The JCL in *orbixhlq*.JCLLIB(DEPLOY3) is run to deploy OTS RRS. Running this JCL sets the initial_references:TransactionFactory:reference configuration item in the iona_services.otstm client scope. Clients can use this reference 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.

The CICS and IMS client adapters can be configured to use the itotstm Transaction Manager service, by adding the initial_references:TransactionFactory:reference configuration item into the iona_services.cics_client and iona_services.ims_client configuration scopes. The Orbix Mainframe configuration is shipped with this configuration item defined. Running the JCL in orbixhlq.JCLLIB(DEPLOY3) sets it to the correct value.

Example client/server deployment

Figure 19 shows a client/server deployment where the itotstm, in conjunction with the OTS RRS plug-in, is used to provide 2PC transaction management. In this case, 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 RRS plug-in loaded into the itotstm standalone service.

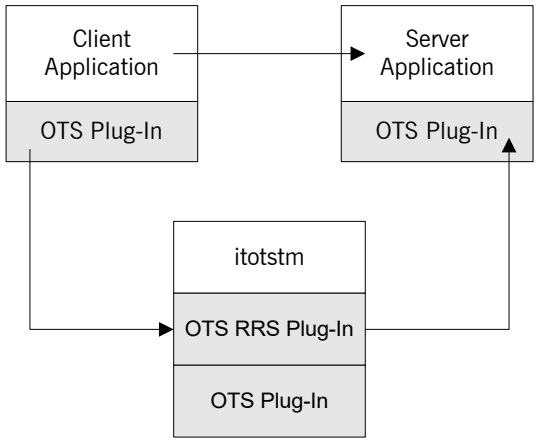

Figure 19: *Using the OTS RRS plug-in with the itotstm service*

There are two parts to setting up such a deployment:

- Configuring the itotstm to load the OTS RRS plug-in.
- Configuring the OTS plug-in to pick up the reference to the OTS RRS transaction factory within the itotstm service.

Configuring itotstm

The itotstm service uses a configuration scope of `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 RRS plug-in can be done by setting the `initial_references:TransactionFactory:plugin` configuration variable to the name of the OTS RRS plug-in `"ots_rrs"`. Orbix Mainframe is shipped with this configuration item defined in the `otstm` scope.

Note: The `orb_plugins` configuration variable must contain `"ots"`, because the OTS plug-in is required for synchronization objects.

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

Configuring the OTS plug-in

Next the OTS plug-in loaded into the application needs to pick up the transaction factory reference of the OTS RRS plug-in. Essentially this means setting the `initial_references:TransactionFactory:refererence` configuration variable in the application's configuration scope to any suitable reference. To do this, get the itotstm service 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.

Note: This is performed automatically when you run the *orbixhlq*`.JCLLIB(DEPLOY3)` JCL.

This deployment option should be used when the application requires (or might require) full recoverable 2PC transactions. For example, the application makes use of one or more resource managers.

Loading the OTS RRS Plug-In into the Application

An alternative to loading the OTS RRS plug-in into the itotstm service is to load the plug-in directly into the application, such as the client adapter. This deployment is shown in Figure 20.

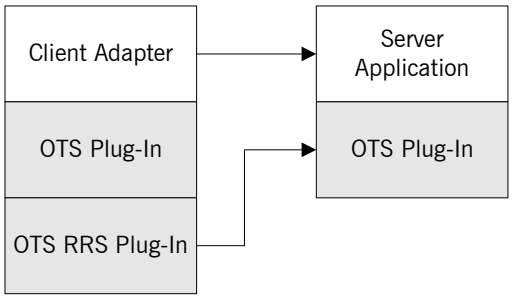

Figure 20: *Loading the OTS RRS Plug-In into the Client Adapter*

This deployment option should be used when the application requires full recoverable 2PC transactions and also wants to improve performance by eliminating some of the network messages that are necessary when the standalone itotstm service is used.

To configure this deployment, follow the instructions for configuring the OTS RRS plug-in and ensure the configuration is performed within the application's scope. For example, to configure the IMS client adapter to load the OTS RRS plug-in, make the following changes in the `iona_services.ims_client` scope:

```
…
plugins:amtp_appc:maximum_sync_level = "2";
initial_references:TransactionFactory:plugin = "otr_rrs";
plugins:ots_rrs:namespace_poa = "otstm";
…
```

Note: Also, ensure that the `initial_references:TransactionFactory:reference` configuration item is preceded by a comment character.

CHAPTER 11

OTS RRS Transaction Manager Configuration

This chapter provides information needed to configure the OTS RRS Transaction Manager and its components (plug-ins). It provides descriptions of all the configuration items involved in running the OTS RRS Transaction Manager. It also provides details on configuring the various system components used by the OTS RRS Transaction Manager. These components include the OTS_RRS plugin.

In this chapter

This chapter discusses the following topics:

Note: Because OTS Encina is not supported by Orbix Mainframe, it is not discussed in this chapter.

OTS RRS Transaction Manager Sample Configuration

Overview

A sample configuration member is supplied with your Orbix Mainframe installation that provides an example of how you might configure and deploy the OTS RRS Transaction Manager on both native z/OS and UNIX System Services.

This section discusses the following topics:

- "Location of configuration templates" on page 138.
- "Configuration scope" on page 138.
- "Configuration scope example" on page 139.
- "Configuring a domain" on page 140.

Location of configuration templates

Sample configuration templates are supplied with your Orbix Mainframe installation in the following locations:

- *Non-TLS*—*orbixhlq*.CONFIG(BASETMPL)
- *TLS*—*orbixhlq*.CONFIG(TLSTMPL)

The *orbixhlq*.CONFIG(ORXINTRL) member contains internal configuration settings.

Configuration scope

The OTS RRS Transaction Manager uses an ORBname of iona_services.otstm. The items specific to the OTS RRS Transaction Manager configuration are scoped in the iona_services.otstm configuration scope.

Configuration scope example

The following is an example of the `iona_services.otstm` configuration scope.

```
otstm
{
     event_log:filters = ["*=WARN+ERROR+FATAL",
                          "IT_OTS_SRV=*",
                          "IT_OTS_RRS=*"];

     plugins:ots_rrs:managed = "false";

     policies:iiop:server_address_mode_policy:local_hostname
            = "%{LOCAL_HOSTNAME}";

     plugins:ots_rrs:direct_persistence = "false";

     # Settings for well-known addressing:
     # (mandatory if direct_persistence is enabled)
     #
     # plugins:ots_rrs:iiop:port = 5003;
     # plugins:ots_rrs:iiop:host = "%{LOCAL_HOSTNAME}";

     plugins:orb:is_managed = "false";
     plugins:it_mgmt:managed_server_id:name
            = "iona_services.otstm";

     client
     {
         initial_references:TransactionFactory:reference
                = "%{LOCAL_OTSTM_REFERENCE}";
     };
};
```

The *orbixhlq*`.CONFIG(ORXINTRL)` member contains the following, which re-opens the scope in the preceding example:

```
otstm,
{
     initial_references:TransactionFactory:plugin = "ots_rrs";

     plugins:ots_rrs:namespace_poa = "otstm";

};
```

Configuring a domain

See the *CORBA Administrator's Guide* for details on how to configure an Orbix domain.

Configuration Summary of OTS RRS Plug-Ins

Overview

Orbix configuration allows you to configure an application on a per-plug-in basis. This section provides a summary of the configuration items associated with plug-ins specific to the OTS RRS Transaction Manager.

This section discusses the following topics:

OTS RRS plug-ins

The OTS RRS Transaction Manager consists of the `ots_rrs` plugin which uses Resource Recovery Services (RRS) to provide two-phase commit services for CICS and IMS transactions using the client adapter, or C++ processes running on z/OS or z/OS Unix System Services.

Summary of items for the ots_rrs plug-in

The following is a summary of the configuration items associated with the `ots_rrs` plug-in. (See "OTS RRS Plug-In Configuration Items" on page 151 for more details):

`allow_registration_after_rollback_only`	Specifies whether registration of resource objects is permitted after a transaction is marked for rollback. The default is `"false"`.
`debug_exits`	Determines whether debugging WTO messages are enabled. The default is `"false"`.
`direct_persistence`	Specifies whether the transaction factory object can use explicit addressing (for example, a fixed port). The default is `"false"`.
`global_namespace_poa`	Specifies the top-level transient POA used as a namespace for OTS implementations. The default is `"iOTS"`.

`high_water_mark`	Specifies the maximum number of threads allowed in the thread pool for RRS exit and restart events. The default is `10`.
`iiop_host`	Specifies the host on which the OTS RRS is running, when run in direct persistence mode.
`iiop_port`	Specifies the port on which OTS RRS listens on when running in direct persistence mode.
`initial_threads`	Specifies the number of initial threads in the thread pool for RRS exit and restart events. The default is `low_water_mark`, or `1` if low_water_mark is not set.
`log_name`	Specifies the resource manager log name. The default value is `rm_name + ".LOG"`.
`low_water_mark`	Specifies the minimum number of threads in the thread pool for RRS exit and restart events. The default is `-1`.
`max_active_timeout_handlers`	Specifies the number of threads to handle timeouts. The default is `5`.
`max_queue_size`	Specifies the maximum number of request items that can be queued on the ORB's internal work queue for RRS exit and restart events. The default is `-1`.
`namespace_poa`	Specifies the transient POA used as a namespace. The default is `"RRS"`.
`orb_name`	Specifies the ORB name used for the plugin's internal ORB when `use_internal_orb` is set to true. The default is the application's ORB name.

otid_format_id	Specifies the value of the formatID field of a transaction's identifier (CosTransactions::otid_t). The default is 0x494f4e41.
resource_retry_limit	Specifies the maximum number of retries to deliver a transaction outcome to an unresponding resource object. The default is -1, indicating no limit.
resource_retry_timeout	Specifies the time in seconds to pause between retries to an unresponding resource object. The default is 5.
rm_name	Specifies the resource manager name. The default value is "ORBIX.OTS".
transaction_factory_name	Specifies the initial reference for the transaction factory. The default is "TransactionFactory".
transaction_timeout_period	Specifies the time, in milliseconds, of which all transaction timeouts are multiples. The default is 1000.
use_internal_orb	Specifies whether the ots_rrs plugin creates an internal ORB for its own use. The default is false.

Summary of remaining configuration items

The following is a summary of the remaining configuration items. (See the *CICS Adapters Administrator's Guide*, *IMS Adapters Administrator's Guide*, and *CORBA Administrator's Guide* for more details):

`event_log:filters`	Specifies the types of events the OTS RRS plug-in logs.
`initial_references: Transaction_Factory: plugin`	Specifies the OTS transaction manager plugin.
`plugins:orb:is_managed`	Specifies whether the OTS RRS Transaction Manager is managed using the management service.
`plugins:it_mgmt:managed_ server_id:name`	Specifies the server name that you wish to appear in the Administrator management console.
`policies:iiop:server_ address_mode_policy: local_hostname`	Specifies the server host name that is advertised by the locator daemon/configuration repository, and listened on by server-side IIOP.

CHAPTER 12

OTS RRS General Configuration

This chapter provides details of the configuration items for the OTS RRS Transaction Manager. These details specify configuration items such as the level of Orbix event logging, hostname, and management.

Overview

This chapter discusses the following topics:

Orbix event logging

The related configuration item is `event_log:filters`. It specifies the level of event logging. To obtain events specific to OTS RRS, the IT_OTS_SRV and IT_OTS_RRS event logging subsystems can be added to this list. For example:

```
event_log:filters = ["*=WARN+ERROR+FATAL",
                     "IT_OTS_SRV=*",
                     "IT_OTS_RRS=*"];
```

This logs all IT_OTS_SRV and IT_OTS_RRS events, and any warning, error, and fatal events from all other subsystems (for example, IT_CORE, IT_GIOP, and so on). The level of detail provided for IT_OTS_SRV and IT_OTS_RRS events can be controlled by setting the relevant logging levels. See the *CORBA Administrator's Guide* for more details.

Transaction Factory Plug-in

The related configuration item is `initial_references:TransactionFactory:plugin`. This specifies the OTS transaction manager plugin that is to be loaded. To load the OTS RRS plugin set the value to `"ots_rrs"`.

Is managed

The related configuration item is `plugins:orb:is_managed`. This specifies whether OTS RRS can be managed using the management service. Setting this to `true` allows for management tasks such as viewing the TimeRunning attribute of OTS RRS, or shutting down ORTS RRS. The default is `"false"`, which means the management service does not manage the service.

Managed server ID name

The related configuration item is `it_mgmt:managed_server_id:name`. This specifies the server name that you wish to appear in the Administrator management console.

To enable management on a server, ensure that the following configuration variables are set:

```
plugins:orb:is_managed = true;
plugins:it_mgmt:managed_server_id:name = <your_server_name>;
```

Local hostname

The related configuration item is `policies:iiop:server_address_mode_policy:local_hostname`. This specifies the server host name that is advertised by the locator daemon/configuration repository, and listened on by server-side IIOP. This variable enables support for multi-homed server hosts. These are server machines with multiple hostnames or IP addresses (for example, those using multiple DNS aliases or multiple network interface cards). The `local_hostname` variable enables you to explicitly specify the host name that the server listens on and publishes in its IORs. For example, if you have a machine with two network addresses (`207.45.52.34` and `207.45.52.35`), you can explicitly set this variable to either address. For example:

```
policies:iiop:server_address_mode_policy:local_hostname =
   "207.45.52.34";
```

By default, the `local_hostname` variable is unspecified.

CHAPTER 13

Configuring the OTS RRS Plug-in

The ots_rrs plug-in allows CICS client transactions, IMS client transactions, and C++ programs running on z/OS or z/OS UNIX System Services to take advantage of two-phase commit processing.

In this chapter

This chapter discusses the following topics:

Setting up RRS for the OTS RRS Plug-in

Overview

This section provides details of how to set up RRS so that you can use the OTS RRS plug-in. It discusses the following topics:

- "Prerequisites to using the OTS RRS plugin".
- "Further reading".

Prerequisites to using the OTS RRS plugin

Before you can use the OTS RRS plug-in, you must first enable the required RRS functionality on your z/OS system. The following components should be made available:

- RRS itself—The OTS RRS plug-in depends on RRS. It must be running before you attempt to run the OTS RRS Transaction manager or any Orbix service that loads the OTS RRS plugin.
- RRS panels—The panels provide for browsing and managing RRS data. See "RRS Panels" on page 173 for more information on using RRS panels.

Further reading

For more information on setting up RRS, refer to the IBM publication *MVS Programming: Resource Recovery SA22-7616*

OTS RRS Plug-In Configuration Items

Overview

This section discusses the following topics:

Allow registration after rollback

The related configuration item is `plugins:ots_rrs:allow_registration_after_rollback_only`. This specifies whether registration of resource objects is permitted after a transaction is marked for rollback.

If this item is set to `true`, it means that resource objects can be registered after a transaction is marked for rollback. If it is set to `false`, it means that resource objects cannot be registered after a transaction is marked for rollback.

This has no effect on the outcome of the transaction. The default is `false`.

Debug Exits

The related configuration item is `plugins:ots_rrs:debug_exits`. This indicates whether debugging WTO messages are enabled. When RRS reports an event to the OTS RRS plug-in via an exit, two WTO messages are issued, as follows:

1. Upon entry, the RRS exit number and URI (Unit of Recovery Identifier) are displayed.
2. Upon exit, the return code passed to RRS from the OTS RRS plug-in is displayed.

The values for the exit numbers and return codes can be found in the IBM publication *MVS Programming: Resource Recovery SA22-7616*.

The default value is `false`.

Direct persistence

The related configuration item is `plugins:ots_rrs:direct_persistence`. It indicates whether the transaction factory object can use explicit addressing—for example, a fixed port. If this item is set to `true`, the addressing information is picked up from `plugins:ots_rrs`. For example, to use a fixed port, set `plugins_ots_rrs:iiop:port`. The default is `false`.

Global namespace POA

The related configuration item is `plugins:ots_rrs:global_namespace_poa`. This specifies the top-level transient POA used as a namespace for OTS implementations. The default is `iOTS`.

High water mark

The related configuration items are `plugins:ots_rrs:exit_pool:high_water_mark` and `plugins:ots_rrs:restart_pool:high_water_mark`. These specify the maximum number of threads allowed in the thread pool used for RRS exit and restart events. The default is `10` in each case.

You must ensure that the `high_water_mark` thread limit does not exceed any OS-specific thread limit (for example, `nkthreads` or `max_thread_proc`). Otherwise, thread creation failure could put your process into an undefined state.

IIOP host

The related configuration item is `plugins:ots_rrs:iiop:host`. It specifies the host on which OTS RRS is running. This is only required when `direct_persistence` is set to `true`.

IIOP port

The related configuration item is `plugins:ots_rrs:iiop:port`. It specifies the port on which OTS RRS listens when it is running in direct persistent mode. This is only required when `direct_persistence` is set to `true`.

Initial threads

The related configuration items are `plugins:ots_rrs:exit_pool:initial_threads` and `plugins:ots_rrs:exit_pool:initial_threads`. These specify the number of initial threads in the thread pool used for RRS exit and restart events. These default to the `low_water_mark` thread limit (or `1`, if the low_water_mark is not set).

Log name

The related configuration item is `plugins:ots_rrs:log_name`. It specifies the resource manager log name. The default value is `rm_name + ".LOG"`.

Low water mark

The related configuration items are `plugins:ots_rrs:exit_pool:low_water_mark` and `plugins:ots_rrs:restart_pool:low_water_mark`. These specify the minimum number of threads in the thread pool used for RRS exit and restart events. If this variable is set, the ORB terminates unused threads until only this number exists. The ORB can then create more threads, if needed, to handle the items in its work queue.

The default is `-1` in each case, which means do not terminate unused threads.

Maximum active timeout handlers

The related configuration item is `plugins:ots_rrs:max_active_timeout_handlers`. This specifies number of threads to handle timeouts. The default is `5`.

Maximum queue size

The related configuration items are `plugins:ots_rrs:exit_pool:max_queue_size` and `plugins:ots_rrs:restart_pool:max_queue_size`. These specify the maximum number of request items that can be queued on the ORB's internal work queue for RRS exit and restart events. If this limit is exceeded, Orbix considers the server to be overloaded, and gracefully closes down connections to reduce the load. The ORB will reject subsequent requests until there is free space in the work queue.

The default is `-1` in each case, which means that there is no upper limit on the size of the request queue. In this case, the maximum work queue size is limited by how much memory is available to the process.

There is no direct relationship between `max_queue_size` and `high_water_mark`. A particular value for `high_water_mark` does not require a corresponding value for `max_queue_size`. For example, even if the queue size is unbounded, each work item should be serviced eventually by the ORB's available threads. However, this will not occur if the threads are unavailable indefinitely and unable to execute a new request from the work queue.

Namespace POA

The related configuration item is `plugins:ots_rrs:namespace_poa`. This specifies the transient POA used as a namespace. This is useful when there are multiple instances of the plug-in being used. Each instance must use a different namespace POA to distinguish itself. The default is `RRS`.

ORB name

The related configuration item is `plugins:ots_rrs:orb_name`. This specifies the ORB name used for the plug-in's internal ORB when `use_internal_orb` is set to `true`. The ORB name determines where the ORB obtains its configuration information, and is useful when the application ORB configuration needs to be different from that of the internal ORB. This defaults to the ORB name of the application ORB.

OTID format ID

The related configuration item is `plugins:ots_rrs:otid_format_id`. This specifies the value of the `formatID` field of a transaction's identifier (`CosTransactions::otid_t`). The default is `0x494f4e41`.

Resource manager name

The related configuration item is `plugins:ots_rrs:rm_name`. This specifies the resource manager name. When using the RRS ISPF panels, this name appears as both a "Work Manager Name" and a "Resource Manager" name in the logs. The default is `"ORBIX.OTS"`.

Resource retry limit

The related configuration item is `plugins:ots_rrs:resource_retry_limit`. This specifies the maximum number of retries to deliver a transaction outcome to an unresponding resource object. The default is `-1`, indicating no limit.

Resource retry timeout

The related configuration item is `plugins:ots_rrs:resource_retry_timeout`. This silicifies the time, in seconds, between retrying a failed invocation on a resource object. A negative value means the default is used. The default is `5`.

Transaction factory name

The related configuration item is `plugins:ots_rrs:transaction_factory_name`. This specifies the initial reference for the transaction factory. This option must match the corresponding entry in the configuration scope of your generic OTS plug-in, to allow it to successfully resolve a transaction factory. The default is `TransactionFactory`.

Transaction timeout period

The related configuration item is `plugins:ots_rrs:transaction_timeout_period`. This specifies the time, in milliseconds, of which all transaction timeouts are multiples. A low value increases accuracy of transaction timeouts, but increases overhead. This value is multiplied to all transaction timeouts. To disable all timeouts, set this item to `0` or a negative value. The default is `1000`.

Use internal ORB

The related configuration item is `plugins:ots_rrs:use_internal_orb`. This specifies whether the `ots_rrs` plugin creates an internal ORB for its own use. By default the `ots_rrs` plugin creates POAs in the application's ORB. This option is useful if you want to isolate the transaction service from your application ORB. The default is `false`.

CHAPTER 14

Using OTS RRS Transaction Manager

This chapter provides information on running and using the OTS RRS Transaction Manager. It provides details on how to prepare, start, and stop OTS RRS.

In this chapter

This chapter discusses the following topics:

Preparing the OTS RRS Transaction Manager

Overview

This section describes what needs to be done to run the OTS RRS Transaction Manager in prepare mode. It discusses the following topics:

Prerequisites to running the OTS RRS Transaction Manager in prepare mode

The load library PDSes in the STEPLIB concatenation of JCL PROC `orbixhlq.JCLLIB(ORXG)` must all be APF-authorized. If any PDS in the STEPLIB concatenation is not APF-authorized, the prepare job will fail.

Use the following system command to list APF-authorized datasets:

```
D PROG,APF
```

If any PDS in the STEPLIB concatenation does not appear in the list, the PDS can be temporarily authorized using the following system command:

```
SETPROG APF,ADD,DSN=pds.name,SMS
```

Note: Contact your z/OS system programmer to make the data sets permanently authorized.

After the STEPLIB PDSes are APF-authorized, run the locator and node daemon. Ensure that these are prepared as described in the *Mainframe Installation Guide* before running them.

RRS must be running before attempting to run the OTS RRS Transaction Manager.

Running the OTS RRS Transaction Manager in prepare mode

Run the OTS RRS Transaction Manager in prepare mode. This generates IORs and writes them to a file, which you can then include in your configuration file. A job to run the OTS RRS Transaction Manager in prepare mode is provided in *orbixhlq*.JCLLIB(DEPLOY3).

Sample JCL to run the OTS RRS Transaction Manager in prepare mode

This JCL contains the default high-level qualifier, so change it to reflect the proper value for your installation:

```
//DEPLOY3  JOB   (),
//         CLASS=A,
//         MSGCLASS=X,
//         MSGLEVEL=(1,1),
//         NOTIFY=&SYSUID,
//         REGION=0M,
//         TIME=1440,
//         COND=(0,NE)
//*
//         JCLLIB ORDER=(HLQ.ORBIX63.PROCLIB)
//         INCLUDE MEMBER=(ORXVARS)
//*//*************************************************************
   ****
//* JCL to deploy the OTS TM
//* Requires locator and node_daemon to be running
//*************************************************************
//*
//* Make the following changes before running this JCL:
//*
//* 1.  If you ran DEPLOY1 (or DEPLOYT) to configure in a domain
//*     other than the default, please ensure that dataset
//*     &ORBIXCFG(ORBARGS) has the domain name used by DEPLOY1
//*     (or DEPLOYT).
//*
//* 2.  Make sure ALL the load libraries in the STEPLIB
//*     concatenation of JCL PROC ORXG are APF authorized.
//*
```

```
//* Prepare the Transaction Service
//*
//PREPOTS  EXEC PROC=ORXG,
//         PROGRAM=ORXOTSTM,
//         LOADLIB=&ORBIX..LOADLIB,
//        PPARM='prepare -publish_to_file=DD:ITCONFIG(IOROTSTM)'
//ORBARGS  DD DSN=&ORBIXCFG(ORBARGS),DISP=SHR
//*
//* Update configuration domain with OTS TransactionFactory IOR
//*
//ITCFG1   EXEC ORXADMIN
//SYSIN DD *
    variable modify \
      -type string \
      -value --from_file:3 //DD:ITCONFIG(IOROTSTM) \
    LOCAL_OTSTM_REFERENCE
/*
//ORBARGS  DD DSN=&ORBIXCFG(ORBARGS),DISP=SHR
//*
//* Update configuration domain with OTS TransactionServiceAdmin
   IOR
//*
//ITCFG2   EXEC ORXADMIN
//SYSIN DD *
    variable modify \
      -type string \
      -value --from_file:6 //DD:ITCONFIG(IOROTSTM) \
    LOCAL_OTSTM_ADM_REFERENCE
/*
//ORBARGS  DD DSN=&ORBIXCFG(ORBARGS),DISP=SHR
```

Location of the OTS RRS Transaction Manager IORs

When complete, the IORs for the OTS RRS Transaction Manager should be in *orbixhlq*.CONFIG(IOROTSTM). The file contains two IORs:

- The TransactionFactory IOR.
- The TransactionServiceAdmin IOR.

The TransactionFactory IOR

The TransactionFactory IOR is used by the client adapter to communicate with the OTS RRS Transaction Manager for two-phase commit processing. The *orbixhlq*.JCLLIB(DEPLOY3) JCL copies this IOR into the LOCAL_OTSTM_REFERENCE configuration item, which is found in the *orbixhlq*.CONFIG PDS, in the member that corresponds to your configuration domain name. (The default configuration domain name is DEFAULT@).

The TransactionServiceAdmin IOR

The TransactionServiceAdmin IOR is used by itadmin to direct commands at the OTS RRS Transaction Manager. The *orbixhlq*.JCLLIB(DEPLOY3) JCL copies this IOR into the LOCAL_OTSTM_ADM_REFERENCE configuration item, which is found in the *orbixhlq*.CONFIG PDS, in the member that corresponds to your configuration domain name. (The default configuration domain name is DEFAULT@).

Sample configuration file

The following is an extract from a working configuration file for you to compare your file with.

```
…
LOCAL_OTSTM_REFERENCE =
   "IOR:000000000000004249444c3a696f6e612e636f6d2f\
49545f436f735472616e73616374696f6e732f5375626f7264696e6174655472
   616e736\
16374696f6e466163746f72793a312e3000000000000000100000000000000c20
   0010200\
0000001d706561636f636b2e6475626c696e2e656d65612e696f6e612e636f6d
   0000426\
90000004d3a3e0232311c706561636f636b2e6475626c696e2e656d65612e696
   f6e612e\
636f6d13694f5453006f7473746d00666163746f72790016f2d7c36de3998195
   a28183a\
3899695c68183a39699a8000000000000003000000000000000080000000049545
   f410000\
00010000001c0000000010020417000000010001000100010100000000010001
   0109000\
000060000000600000000035";
LOCAL_OTSTM_ADM_REFERENCE =
   "IOR:000000000000003249444c3a696f6e612e636f\
6d2f49545f4f54535f536572766963656416d696e2f5365727669636541646d
   696e3a3\
12e3000000000000001000000000000000be000102000000001d706561636f636
   b2e6475\
626c696e2e656d65612e696f6e612e636f6d000042690000004c3a3e0232311c
   7065616\
36f636b2e6475626c696e2e656d65612e696f6e612e636f6d11694f5453006f7
   473746d\
0061646d696e0017e3998195a28183a3899695e28599a5898385c18494899500
   0000030\
000000000000008000000049545f41000000010000001c00000000100204170
   0000001\
000100010001010000000001000101090000000600000006000000000033";
…
```

Running the OTS RRS Transaction Manager on z/OS UNIX System Services

You can also run the OTS RRS Transaction Manager in prepare mode from the UNIX System Services prompt.

Before running the command, do the following:

- Ensure that the environment variable `_BPX_SHAREAS` is not set in the shell where itotstm will be run.
- Navigate to the *orbixhlq*`/asp/`*version*`/bin` directory and issue the following command to mark `itotstm` as authorized:

```
extattr +a itotstm
```

- Navigate to the `orbixhlq/shlib` directory and issue the following command to mark DLLs in z/OS UNIX System Services as authorized:

```
extattr +a ORX*
```

- Ensure that all the load libraries in the STEPLIB concatenation of PROC *orbixhlq*`.JCLLIB(ORXG)` are APF-authorized as described in Prerequisites to running the OTS RRS Transaction Manager in prepare mode.

The command is as follows:

```
itotstm prepare
```

The two IORs for TransactionFactory and TransactionServiceAdmin are then displayed on the console. You can copy them to the appropriate places as described above. However, in general, it might be easier to obtain the TransactionFactory and TransactionServiceAdmin IORs using the *orbixhlq*`.JCLLIB(DEPLOY3)` JCL. This is because it automatically copies the IORs into the PDS-based configuration file.

Starting the OTS RRS Transaction Manager

Overview

This section describes how to start the OTS RRS Transaction Manager. It discusses the following topics:

- "Starting the OTS RRS Transaction Manager on native z/OS" on page 164.
- "Starting the OTS RRS Transaction Manager on z/OS UNIX System Services" on page 165.
- "Running with a different configuration scope" on page 165.

Starting the OTS RRS Transaction Manager on native z/OS

In a native z/OS environment, you can start the OTS RRS Transaction Manager in any of the following ways:

- As a batch job.
- Using a TSO command.
- As a started task (by converting the batch job into a started task).

The following is sample JCL to run the OTS RRS Transaction Manager:

```
//OTSTM    JOB   (),
//         CLASS=A,
//         MSGCLASS=X,
//         MSGLEVEL=(1,1),
//         NOTIFY=&SYSUID,
//         REGION=0M,
//         TIME=1440
//*
//         JCLLIB ORDER=(HLQ.ORBIX63.PROCLIB)
//         INCLUDE MEMBER=(ORXVARS)
//*
//* Run the Orbix OTS TM
//*
//* Make the following changes before running this JCL:
//*
//* 1.  Change 'SET DOMAIN='DEFAULT@' to your configuration
//*     domain name.
//*
//* 2.  Make sure ALL the load libraries in the STEPLIB
//*     concatenation of JCL PROC ORXG are APF authorized.
//*
```

```
//          SET DOMAIN='DEFAULT@'
//*
//GO  EXEC PROC=ORXG,
//    PROGRAM=ORXOTSTM,

//    LOADLIB=&ORBIX..LOADLIB,
//    PPARM='run'
//ITDOMAIN DD DSN=&ORBIXCFG(&DOMAIN),DISP=SHR
```

Note: For the STEPLIB of PROC ORXG, all data sets must be APF-authorized. See "Prerequisites to running the OTS RRS Transaction Manager in prepare mode" on page 158 for instructions on how to make the data sets APF-authorized.

Starting the OTS RRS Transaction Manager on z/OS UNIX System Services

On z/OS UNIX System Services, you can start the OTS RRS Transaction Manager from the shell. The following command is used to run the OTS RRS Transaction Manager:

```
$ itotstm
```

Note: Before running itotstm, see "Running the OTS RRS Transaction Manager on z/OS UNIX System Services" on page 163 for a description of environment variables and extended attributes that must be set.

Running with a different configuration scope

To run the OTS RSS Transaction Manager with a different configuration scope on native z/OS, set the value of PPARM to the new scope, for example:

```
PPARM='run -ORBname iona_services.otstm_test'
```

To run the OTS RRS Transaction Manager with a different configuration scope on z/OS UNIX System Services, run a command similar to the following:

```
$ itotstm -ORBname iona_services.otstm_test
```

Stopping the OTS RRS Transaction Manager

Overview

This section describes how to stop OTS RRS. It discusses the following topics:

Stopping the OTS RRS Transaction Manager on native z/OS

To stop the OTS RRS Transaction Manager job on native z/OS, issue the `STOP (P)` operator command from the console.

Stopping the OTS RRS Transaction Manager on z/OS UNIX System Services

To stop the OTS RRS Transaction Manager process on z/OS UNIX System Services, use the `kill` command or press **Ctrl-C** if it is running in an active `rlogin` shell.

Stopping the OTS RRS Transaction Manager using itadmin

Whether using an itadmin JOB on native z/OS, or the interactive shell on z/OS UNIX System Services, use the `otstm stop` command to stop the OTS RRS Transaction Manager.

Stopping the OTS RRS Transaction Manager using the Administrator

The OTS RRS Transaction Manager can be stopped from the Administrator Web Console. In your browser, navigate to the OTS RRS Transaction Manager server and invoke the `shutdown` operation.

See "Introduction to OTS Management" on page 171 on how to set up for the Administrator Web Console.

Difficulty stopping the OTS RRS Transaction Manager

After a request to stop has been sent to the OTS RRS Transaction Manager, RRS itself might decide that it will not allow the OTS RRS Transaction Manager to unregister itself as a resource manager. This results in the OTS RRS Transaction Manager continuing to run, despite the stop request.

If this happens, edit and submit the JCL in *orbixhlq*.JCLLIB(RRSUNSET) to unregister the OTS RRS Transaction Manager from RRS as a resource manager, allowing it to stop.

Part 3

Appendices

In this part

This part contains the following chapters:

Note: Both of these appendices are relevant regardless of which programming language is being used for application development.

APPENDIX A

Introduction to OTS Management

This appendix provides an introduction on how to set up for management using Orbix Administrator.

In this Appendix

This appendix discusses the following topics:

Orbix Administrator

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

Orbix Administrator also provides a web browser interface known as the 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 Orbix Administrator, see the *Management User's Guide*.

Configuring for Management

Before Orbix Administrator can be used, the Orbix environment must first be configured. This involves:

- Running the management service off-host.
- Configuring the management service IORs.
- Configuring the OTS RRS Transaction Manager service, so that it can be managed by the off-host management service.

See the *Mainframe Management Guide* for details on running the management service off-host, and configuring the management service IORs.

To enable the OTS RRS Transaction Manager service for management by the off-host management service, the following configuration variables must be set in the `otstm` scope:

```
plugins:ots_rrs:debug_exits = "true";
plugins:it_mgmt:managed_server_id:name =
   "iona_services.otstm.myhost";
```

Note: The name specified as the setting for `plugins:it_mgmt:managed_server_id:name` should be set to a meaningful name for your installation. This name will appear in the Administrator Web Console.

What can be managed?

The following can be managed:

- Attributes for the OTS RRS Transaction Manager service can be browsed.
- The OTS RRS Transaction Manager can be shut down.
- Attributes for the OTS RRS Transaction Manager process can be browsed.
- Attributes for the OTS RRS Transaction Manager ORB can be browsed.
- Attributes for the OTS RRS Transaction Manager workqueues can be browsed.
- The event log filter attribute can be browsed and dynamically updated.

RRS Panels

When processing transactions using the two-phase commit protocol, failures might occur. These failures might leave data in an inconsistent state, and have to be investigated. Often the investigation will include looking at and manipulating information maintained by RRS. This appendix discusses troubleshooting through the use of RRS panels.

In this appendix

This appendix discusses the following topics:

RRS ISPF Panels

IBM provides a set of Interactive System Productivity Facility (ISPF) panels that allow for browsing and taking actions. The information that can be browsed includes:

- RRS Logs
- Resource manager information
- Unit of Recovery (UR) information
- Work manager information
- RRS system information

The types of actions that can be taken include:

- Act upon URs that are in an `InDoubt` state due to some failure while processing a two-phase commit transaction.
- Remove a resource manager's interest in a UR.
- After a system failure, determine if a resource manager can be restarted.

For more information see the IBM publication *MVS Programming: Resource Recovery SA22-7616*.

Correlate RRS and Client Adapter Information

To correlate information between two-phase commit transactions processed by the client adapter, and information kept in RRS, increase the logging level of the client adapter to get log messages related to two-phase commit processing.

To do this, configure the client adapter event filter to include `INFO_LOW`. This setting logs messages about two-phase commit transactions processed by the client adapter.

Client Adapter Log Messages

When a two-phase commit transaction is processed by the client adapter, a series of messages similar to the following are logged:

```
(IT_MFU:216) I - Process client request as a transaction:
LUW ID:                    EXPNET.IMSLU02 459C42471260 0001
Target:                    corbaloc:rir:/DataObjectA
Operation:                 write
Interface repository ID:   IDL:Data:1.0
…
(IT_MFU:216) I - Two-phase commit begins for updates under LUW
   ID:  EXPNET.IMSLU02 459C42471260 0001.
(IT_MFU:217) I - Prepare vote is 'commit' for updates under LUW
   ID:  EXPNET.IMSLU02 459C42471260 0001.
(IT_MFU:220) I - All resources have voted to commit.  Proceeding
   to commit updates under LUW ID:  EXPNET.IMSLU02 459C42471260
   0001.
(IT_MFU:221) I - Successfully committed updates under LUW ID:
   EXPNET.IMSLU02 459C42471260 0001.
(IT_MFU:219) I - Two-phase commit ends for updates under LUW ID:
   EXPNET.IMSLU02 459C42471260 0001.
```

The log messages indicate the following:

- A request flowing through the client adapter is to be processed as a transaction using two-phase commit. A log message identifies the Localize Unit of Work ID (LUW ID), which ties related two-phase commit messages together. The target server and the operation for that server to perform are identified, as well as the interface repository ID.
- After all requests to the server have been completed, the transaction running in CICS or IMS initiates two-phase commit processing.
- The client adapter detects that the CICS or IMS transaction has initiated two-phase commit processing, and sends a 'prepare' request to the server. The server replies with a vote to 'commit'. The client adapter issues a log message indicating the vote from the server, and then returns this vote to the OTS RRS Transaction Manager.
- The OTS RRS Transaction Manager collects the prepare votes from all participants in the transaction. The result of the vote is returned to the client adapter, and a log message is issued indicating the result of all prepare votes.
- The client adapter sends a 'commit' request to the server. The server replies that the commit was successful, and the client adapter logs a message indicating the successful commit.
- The client adapter logs a message indicating two-phase commit processing for the transaction is completed.

Browsing the RRS Log Stream

The LUW ID can be used to correlate a two-phase commit transaction processed by the client adapter with information kept in RRS. Using the RRS panels:

- Select the option for **"Browse an RRS log stream"**.
- Select the option for **"RRS Unit of Recovery State logs"** and the option for a **"Summary"** report.
- This places you in an ISPF browser. Use the ISPF `'find'` command to find the LUW ID displayed in the client adapter's log messages.

In the following example, there are four entries in the RRS Unit of Recovery State logs related to the LUW ID, as follows:

- This RRS log entry indicates that the work manager is the IMS region that initiated the transaction. The state of the unit of work is `InPrepare`.

```
HOST  2005/05/19 12:40:04.244794 BLOCKID=000000000008D68C
  URID=BD08459B7E5503740000051C01010000
   LOGSTREAM=ATR.IONAPLEX.DELAYED.UR
   PARENT URID=00000000000000000000000000000000
   SURID=N/A
   WORK MANAGER NAME=HOST.IMS81JD3.0076
   STATE=InPrepare    EXITFLAGS=00000000 FLAGS=A0000000
  LUWID=EXPNET.IMSLU02 459C42471260 0001    TID=            GTID=
…
```

- This RRS log entry indicates that the work manager is the IMS client adapter. Note the different URID from the IMS region's URID in the preceding log message. The state of the unit of work has transitioned to `InDoubt`.

```
HOST  2005/05/19,12:40:04.331445,BLOCKID=000000000008DE82
  URID=BD08459C7E5506E8000000E801010000
   LOGSTREAM=ATR.IONAPLEX.DELAYED.UR
  PARENT URID=00000000000000000000000000000000
  SURID=N/A
  WORK MANAGER NAME=HOST.IMSCLADP.0042
  STATE=InDoubt      EXITFLAGS=00000000 FLAGS=A0000000
  LUWID=EXPNET.IMSLU02 459C42471260 0001    TID=            GTID=
…
```

- This RRS log entry indicates that the work manager is the IMS region that initiated the transaction. The state of the unit of work is `InCommit`.

```
HOST  2005/05/19,12:40:04.335325,BLOCKID=000000000008E134
  URID=BD08459B7E5503740000051C01010000
   LOGSTREAM=ATR.IONAPLEX.DELAYED.UR
  PARENT URID=00000000000000000000000000000000
  SURID=N/A
  WORK MANAGER NAME=HOST.IMS81JD3.0076
  STATE=InCommit     EXITFLAGS=00800000 FLAGS=A0000000
  LUWID=EXPNET.IMSLU02 459C42471260 0001    TID=            GTID=
…
```

- This RRS log entry indicates that the work manager is the IMS client adapter. The state of the unit of work is `InCommit`.

```
HOST  2005/05/19,12:40:04.337006,BLOCKID=000000000008E431
  URID=BD08459C7E5506E8000000E801010000
   LOGSTREAM=ATR.IONAPLEX.DELAYED.UR
  PARENT URID=00000000000000000000000000000000
  SURID=N/A
  WORK MANAGER NAME=HOST.IMSCLADP.0042
  STATE=InCommit     EXITFLAGS=00800000 FLAGS=A0000000
  LUWID=EXPNET.IMSLU02 459C42471260 0001    TID=            GTID=
…
```

Failure during Two-phase commit

During two-phase commit processing several failures can occur such as:

- z/OS might fail.
- CICS or IMS might fail.
- A CICS or IMS transaction might fail.
- The client adapter might fail.
- APPC/MVS might fail.
- The operating system on which the server is running might fail.
- The server might fail.
- The link between the client adapter and the server might fail.

Failures such as those in the preceding list might cause data to be inconsistent if they occur while a two-phase commit transaction is being processed.

RRS Unit of Recovery

If a failure has occurred during two-phase commit processing, or a two-phase commit transaction appears to be hanging, use the "Display/Update RRS Unit of Recovery information" panel to look at Unit of Recovery information.

Filters can be used to list Units of Recovery (URs) that are not in a "good state". For example, a list of URs that are `InDoubt` can be listed.

Use the panels to view the URs, and correlate the LUW ID back to the client adapter log messages to identify any transactions that might have been processed by the client adapter but that have not completed successfully. An investigation should then follow to see if there are any data inconsistencies. Check to see if CICS or IMS has committed data that the

server has not committed. Similarly, check to see if the server has committed data that CICS or IMS has not committed. Manual intervention might be required to correct any inconsistencies.

After the inconsistency is corrected, the RRS panels can be used to set the UR state to `InCommit` or `InBackout`. For more information, see the IBM publication *MVS Programming: Resource Recovery SA22-7616*.

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 centralized Orbix configuration repository or as a set of files distributed among domain hosts. Configuration domains let you organize 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 centralized 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 organizing 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

S

T

U

V

X