OrbixOTS Programmer's and Administrator's Guide

IONA Technologies PLC
September 2000

M 2 4 7 6

Contents

Part II Programming

Part III Advanced Programming

Preface

OrbixOTS is an implementation of the CORBA Object Transaction Service (OTS). As a CORBA Service, the OTS is an integral part of the the Object Management Group (OMG) vision of truly reusable and reliable object-based software components. OrbixOTS was developed in collaboration with the Transarc Corporation to bring the powerful computing concept of transaction processing to distributed objects.

Orbix documentation is periodically updated. New versions between releases are available at this site:

`http://www.iona.com/docs/orbix/orbix33.html`

If you need assistance with Orbix or any other IONA products, contact IONA at `support@iona.com`. Comments on IONA documentation can be sent to `doc-feedback@iona.com`.

Audience

This book is for administrators and programmers.

Programmers should have the following knowledge:

- Experience of the C++ or Java languages.
- Experience with Orbix programming.
- Knowledge of transaction concepts.
- If using the Java classes, experience with Orbix Java programming.

Organization of this Guide

This guide is divided into the following parts:

Part I Introduction and Administration

This part reviews the basics of transactions, describes what happens during a transaction, and gives a basic overview of the architecture of OrbixOTS.

This part also has a chapter on OrbixOTS administration.

Part II Programming

This part describes how to start programming with OrbixOTS and includes discussions on specifying transactional classes and writing OrbixOTS servers and clients. It also describes basic programming with the Java classes.

Part III Advanced Programming

Topics in this part include such things as controlling transactions, writing a recoverable resource, advanced XA programming, and concurrency.

Part IV Programmer's Reference

This reference covers details of modules, interfaces, and classes for OrbixOTS.

Appendix A, "The DTP Reference Model"

This appendix describes the Distributed Transaction Processing (DTP) Reference Model.

Appendix B, "The OrbixOTS Transaction Factory"

This describes the otstf tool which can be used to develop Java applications without writing any C++ code.

Appendix C, "OrbixOTS Configuration Variables"

This appendix describes the OrbixOTS configuration values.

Document 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.

This guide uses the following conventions for user interface instructions:

Entering commands	When instructed to "enter" or "issue" a command, type the command and then press `Return`. For example, the instruction "Enter the `ls` command" means type the `ls` command and then press the `Return` key.
Clicking items	When instructed to "click" an item from a set of buttons or other options, use the mouse or keyboard to choose that item.
Selecting items	When instructed to "select" a menu, menu item, or multiple items, use the mouse or keyboard to highlight the item on the screen.

This guide may use the following keying conventions:

No prompt	When a command's format is the same for multiple platforms, no prompt is 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, or Windows 95 command prompt.
`Return` key	The notation "`Return` key" refers to the key that is labelled with the word Return, the word Enter, or the left arrow.
... . . .	Horizontal and vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify a discussion.
< >	Angle brackets enclose the names of keys on the keyboard. Also, the notation <Ctrl-x> (where x is the name of a key) indicates a control-character sequence. For example, <Ctrl-c> means hold down the <Ctrl> key while you press the <c> key.
[ ]	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 I

Introduction and Administration

1

Introduction to OrbixOTS

This chapter provides a brief overview of transaction processing concepts and standards, and gives a broad outline of the functionality that OrbixOTS can provide.

Orbix, IONA Technologies' flagship product, gives separate software objects the power to interact freely even if they are on different platforms or written in different languages. OrbixOTS 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. "Basics of Transactions" on page 4 discusses the transaction concept in detail.

This chapter introduces OrbixOTS and demonstrates how it can improve software development in your enterprise.

OrbixOTS Features

OrbixOTS offers the following features for object-oriented, distributed, transaction-processing applications:

- Complete implementation of the Object Management Group's Object Transaction Service (OMG OTS).
- C++ and Java classes for developing OrbixOTS applications.
- Integration of XA-compliant databases.
- Complete implementation of the OMG Object Concurrency Control Service (OCCS).

Basics of Transactions

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 total amount of money in the system should be the same, before and after each transaction.
- 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.

These 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 all committed or all 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 system.
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.

Distributed Transaction Processing (DTP)

OrbixOTS is an implementation of the Object Transaction Service (OTS), which is an OMG standard for a CORBA transaction manager. The design is based on the distributed transaction processing reference model of The X/Open Company, Ltd. *Distributed Transaction Processing* (DTP) gives a transactional mechanism to update two or more independent data resources.

An external entity, usually called a *transaction manager*, provides the framework that allows a transaction to span more then one application, process, or machine. It does this by keeping track of the resources involved in the transaction. It coordinates transaction completion by contacting those resources individually when issued with a commit or rollback instruction from the client that originated the transaction.

OMG OTS improves on the DTP reference model with two ehancements:

- The procedural XA and TX interfaces have been replaced with a set of CORBA interfaces defined in IDL.
- All inter-component communication is mandated to be via CORBA invocations on instances of these interfaces.

Thus the DTP reference standard has been upgraded to an object-oriented model, and interprocess communication mechanisms have been defined to give a common standard for vendor interoperability.

See Appendix A, "The DTP Reference Model" for more background informtion on distributed transaction processing.

What Happens During a Transaction

This section gives a broad overview of how OrbixOTS is involved in coordinating a typical distributed transaction. Figure 1.1 through Figure 1.4 depict a hypothetical situation where two servers, each with its own database, are distributed using OrbixOTS. The servers can be on different machines or in different processes on the same machine.

Suppose that a client wants to use an object that requires that the two servers each update their respective databases. OrbixOTS mediates between the applications, ensuring that the database updates are performed atomically. OrbixOTS is shown here as separate from the applications—this is conceptual: it is actually distributed between the applications. Thus calls that here seem to be between processes, may in fact be local.

1. A client begins a transaction by making a call on OrbixOTS (Figure 1.1). The client is now in the context of a created transaction.

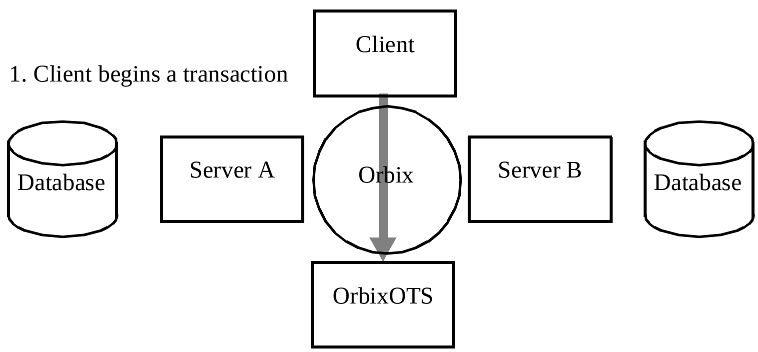


Figure 1.1: *Client Begins a Transaction*

2. The client next invokes Server A and B over Orbix, by making a call on a transactional object (Figure 1.2 on page 7). These invocations can be in parallel by using separate threads if necessary. These calls carry with them knowledge of the transaction that has begun. Servers A and B are said to be participants in the global transaction.
3. The servers proceed to update their databases, but do not commit the updates; OrbixOTS is responsible for performing the commit.

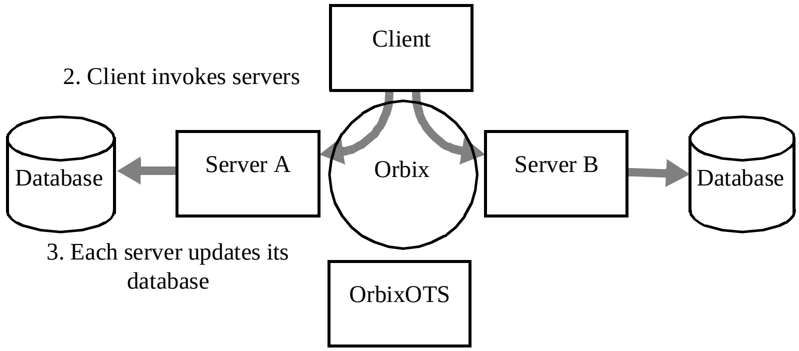


Figure 1.2: *Client Invokes Servers*

A server typically communicates with the database via an XA resource manager (not shown). Resource managers are typically registered with OrbixOTS in server startup by using a special operation.

As an alternative to using an XA resource manager, the server can create a resource object for each transaction upon the first invocation. The server then registers these resource objects with OrbixOTS prior to database updates.

4. The client now requests completion of the transaction by invoking the commit operation on OrbixOTS (Figure 1.3).

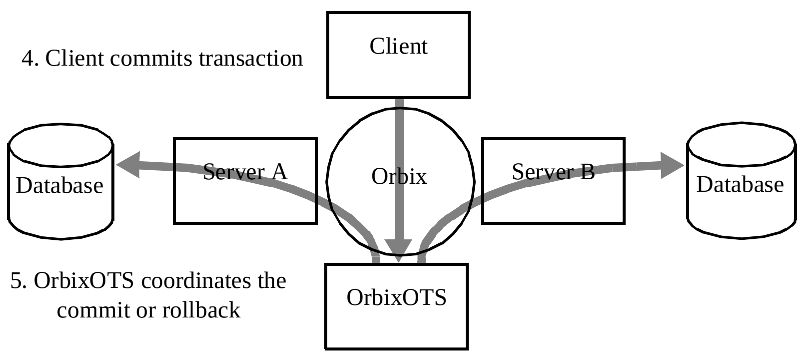


Figure 1.3: *Client Requests a Commit of the Transaction*

5. OrbixOTS coordinates the commit or rollback. OrbixOTS completes the transaction with all resource managers, using a two-phase commit protocol.

A Two-Phase Commit

The OrbixOTS transaction manager uses a *two-phase commit protocol* (2PC protocol) to commit a transaction to the relevant resources: first, all resources for the transaction are asked to prepare the transaction and return a vote to indicate whether they are willing to make the changes durable. Based on the responses received from this phase, the transaction manager begins the second phase of completion: if all resources voted to commit, then they are asked to commit in turn; if one or more resources voted to rollback, then all the others are requested to rollback. In this way atomicity is assured.

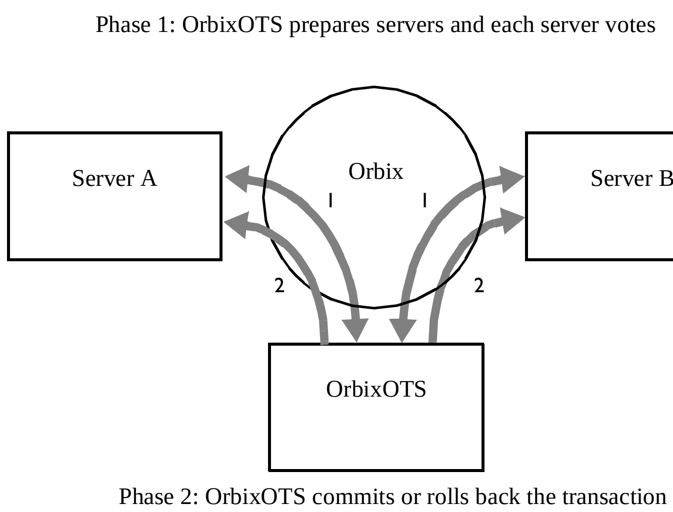


Figure 1.4: *A Two-Phase Commit*

The Components of OrbixOTS

OrbixOTS has a modular architecture that includes such services as distributed transactions (via a Transaction Manager) and an XA Manager that uses the XA protocol to integrate applications with databases or queuing systems. All interprocess communication takes place using Orbix. The architecture includes the OMG Object Concurrency Control Service (OCCS), and services for logging and failure recovery. The components differ for C++ and Java applications.

C++ Server Components

Figure 1.5 illustrates the OTS C++ server components.

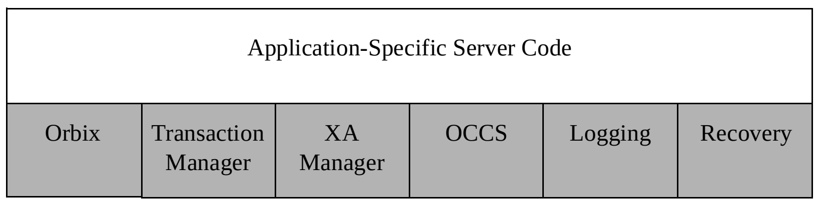


Figure 1.5: *OrbixOTS C++ Server Components*

Transaction Manager

The transaction manager is implemented as a linked-in library, therefore, transactional applications have an instance of the transaction manager that cooperates to implement distributed transactions. This architecture has the advantage that the transaction manager is linked-in (and there is no dedicated “transaction server”). There is no central point of failure. Application or resource failures during the two-phase commit protocol will block the committing transaction, but will not stop new transactions from executing.

The transaction manager provides a full implementation of the OMG OTS interface, which includes several advanced features. For example, nested transactions are supported, and both clients and servers can be multithreaded.

XA Manager

OrbixOTS provides resource manager support for the X/Open XA interface. Many products support the XA interface including Oracle, Sybase, and Informix relational databases, as well as IBM's MQ Series queuing product.

OCCS

The OCCS is an advanced locking service that fully supports nested transactions and works in cooperation with the transaction manager. The OCCS implementation component is linked into the application that is acquiring the locks. Hence, the OCCS is not a true distributed locking service, but because the interfaces are defined using CORBA IDL, servers can be developed that export the OCCS interface to provide a server that effectively implements a distributed locking service.

Logging

Logging provides a durable record of the progress of transactions so that OrbixOTS servers can recover from failure. OrbixOTS permits both ordinary files and raw devices to be used for transaction logs. A transaction log can be expanded at runtime and it can be mirrored for redundancy. Also, an OrbixOTS server can provide a logging service for other recoverable servers.

Recovery

The transaction log is used during recovery after a crash to restore the state of transactions that were in progress at the time of the crash.

C++ Client Components

The simpler client OrbixOTS architecture (See Figure 1.6 on page 11) includes only the components for interprocess communication and distributed transactions.

OrbixOTS also includes a set of CORBA IDL interfaces for administering transactional servers. The OrbixOTS administration interfaces are used by a command-line tool that allows users to query the active transactions at a server, rollback active transactions, and force the outcome of prepared transactions, among other things.

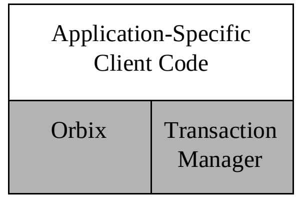


Figure 1.6: *The Client Components*

Java Components

OrbixOTS includes Java classes built with Orbix Java Edition, and a C++ transaction factory server tool that exports the C++ server component interfaces employed by the Java servers.

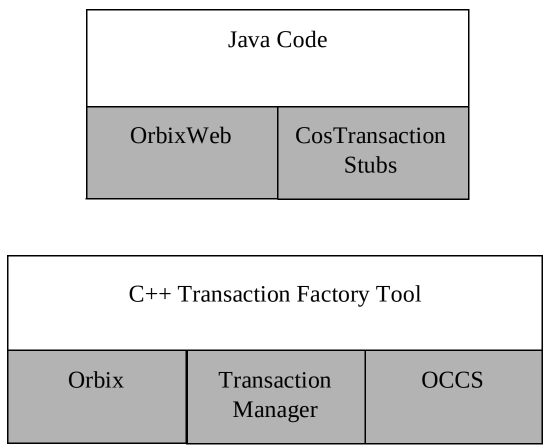


Figure 1.7: *The Java Components*

Java clients can initiate transactions, and Java servers can implement transactional interfaces. The transaction factory manages the transaction on behalf of the Java server. See Appendix B, "The OrbixOTS Transaction Factory". In OMG terminology, Java OTS servers are "transactional servers". The Java servers can also be recovered if they implement a resource.

Overview of the OMG OTS

OrbixOTS supports the OMG Object Transaction Service (OTS) and the Object Concurrency Control Service (OCCS). The following is an overview of both services which introduces the main interfaces, concepts, and terms used in OrbixOTS documentation.

The Object Transaction Service

The OTS provides distributed transaction processing similar to the X/Open DTP model. The X/Open model is supported by allowing X/Open XA compliant resource managers to participate in OTS transactions. In addition, the OTS provides a set of IDL interfaces for controlling transactions and to allow multiple objects distributed over a network to participate in transactions.

The OTS also supports sub-transactions in which transactions can be nested in a top-level transaction. A sub-transaction can roll back without causing its parent to be rolled back. This means that a transaction is isolated from the failures of its sub-transactions. This results in greater control over the granularity of failure when programming with transactions. A single top-level transaction and all of its sub-transactions are called a transaction family (each sub-transaction has a single parent transaction and a transaction may have several child transactions).

The following is a list of the main interfaces supported by the OTS. All interfaces are part of the IDL module `CosTransactions`.

OTS Interface	Function
`Control`	Represents a single top-level or nested transaction. Objects supporting this interface provide access to the transaction's coordinator and terminator options.

Table 1.8: *OTS Interfaces*

OTS Interface	Function
`Coordinator`	Provides operations to register objects that participate in the transaction. There are three types of object that can participate in a transaction: resource objects and synchronisation objects participate in top-level transactions; sub-transaction-aware resource objects participate in nested transactions. XA resource managers can also participate but the `Coordinator` interface is not used for this purpose.The `Coordinator` interface also provides an operation to create nested transactions and for obtaining information about transactions.
`Current`	A pseudo IDL interface that provides the concept of a current transaction that is associated with the current thread of control. Operations are provided to create new transactions (top-level or nested) and to commit or roll back the current transaction.The `Current` interface supports a subset of the operations provided by the `Coordinator` and `Terminator` interfaces.
`RecoveryCoordinator`	Used in certain failure cases to complete a commit protocol for a registered resource object.
`Resource`	Represents a participant in a transaction. Objects supporting this interface are registered with a transaction's coordinator object, and are then invoked at key points in the transaction's commit protocol or when the transaction rolls back.
`SubtransactionAwareResource`	Represents a participant in a sub-transaction. Objects supporting this interface are registered with a sub-transaction's coordinator object and are then invoked when the transaction commits or rolls back.

Table 1.8: *OTS Interfaces*

OTS Interface	Function
`Synchronization`	Provides a means of synchronising transient data with an X/Open XA resource manager. Objects supporting this interface are registered with a transaction's coordinator object, and are then invoked before the start of the commit protocol and at the end of the commit protocol.
`Terminator`	Provides a means of directly committing or rolling back a transaction.
`TransactionalObject`	An empty interface that serves to mark interfaces as being transactional. Any object that supports this interface is implicitly associated with the transaction performing invocations on it.
`TransactionFactory`	Provides a means of directly creating top-level transactions. Each OrbixOTS server has a single transaction factory object.

Table 1.8: *OTS Interfaces*

The OTS supports two modes of controlling transactions: *direct* and *indirect*. In the direct mode, top-level transactions are created using a transaction factory, and are committed or rolled-back using a terminator object. Nested transactions are created using a coordinator object. Applications directly access the OTS objects representing the transaction, which provides a high degree of flexibility but can be difficult to manage. With the indirect mode, transactions are created, committed and rolled-back using a `current` pseudo object. Compared with the explicit mode, use of the `current` pseudo object makes control of transactions much easier. The `current` object here is used to represent the current transaction, and the transaction is implicitly associated with the current thread-of-control. When a transaction is created, if there is no current transaction, a top-level transaction is created. Otherwise a sub-transaction is created.

Similarly, the OTS supports two modes of passing information about transactions between clients and servers: *explicit* and *implicit*. In the explicit mode, each IDL operation includes a reference to the transaction's control object. Thus client applications must explicitly pass information about the transaction to the server. The implicit mode makes use of the `current` object to pass information to any object supporting the `TransactionalObject` interface. Thus, client applications do not need to do anything to ensure that the information is passed to the server.

Taken together, the indirect and implicit modes provide the simplest mechanism for programming with transactions. These modes have the further advantage of the automatic participation of registered X/Open XA resource managers in transactions. The direct and explicit modes are more difficult to manage but provide greater flexibility.

The `Resource` and `SubtransactionAwareResource` interfaces are provided so that applications can implement their own recoverable resources. These allow objects to become full participants in a transaction's distributed two-phase-commit protocol.

The Object Concurrency Control Service

In addition to the transaction service, OrbixOTS also provides an implementation of the OMG Object Concurrency Control Service (OCCS). This service mediates between concurrent transactions attempting to access a shared resource.

The OCCS uses locks as the basis of its concurrency control. There are several lock modes that include support for read/write locking and hierarchical locking. Transactions acquire locks on lock-set objects which are associated with a shared resource.

The IDL module `CosConcurrencyControl` provides the following interfaces:

CosConcurrencyControl Interfaces	Function
`LockCoordinator`	Each lock-set object has a lock coordinator that provides a means of dropping all locks at once.

Table 1.9: *OCCS Interfaces*

CosConcurrencyControl Interfaces	Function
`LockSet`	Represents an implicit lock set. Requests to acquire or release a lock are made on behalf of the current transaction.
`LockSetFactory`	Provides a means of creating lock-set objects (both implicit and explicit). Each OrbixOTS server contains a single lock-set factory object.
`TransactionalLockSet`	Represents an explicit lock-set. Requests to acquire or release a lock are made on behalf of the transaction whose coordinator reference is explicitly passed as a parameter to the operation.

Table 1.9: *OCCS Interfaces*

The OCCS is typically used to provide concurrency control for applications that implement their own recoverable resources using the `CosTransactions::Resource` interface.

2

OrbixOTS Configuration and Administration

You can set the basic OrbixOTS configuration values using the standard Orbix configuration mechanism. But OrbixOTS also provides you with `otsadmin`*–a powerful tool that allows you to administer OrbixOTS servers.*

The basic OrbixOTS uses the same configuration mechanism as Orbix and by convention the variables are contained in the `orbixots.cfg` file. These variables may be set using the Orbix configuration tool and are described in Appendix C, "OrbixOTS Configuration Variables"on page 283. But you can also fine-tune OrbixOTS with its administration tool, `otsadmin`.

The `otsadmin` tool provides administrative functions in four different areas:

- The ability to list the transactions that a server "knows" about, to abort these transactions, and to force a prepared transaction to complete.
- The ability to work on the server's transaction log. This allows the log to be expanded and replicated.
- The ability to control OrbixOTS servers.
- The ability to trace the execution of a server with diagnostics.

This chapter also describes the transaction log used by OrbixOTS servers.

Running the otsadmin Tool

The OrbixOTS administration tool is called `otsadmin`. This tool can be run in either interactive or non-interactive modes. To use the interactive mode by simply running the tool–enter the comand `otsadmin` at the command-line. You are prompted for commands which execute until you issue the `quit` command. For example:

```
% otsadmin
otsadmin> list tran -server Bank
…
otsadmin> abort tran 123 -server Bank
otsadmin> quit
```

To run `otsadmin` in the non-interactive mode, specify the command with complete command-line arguments. For example:

```
% otsadmin abort tran 123 -server Bank
```

Commands are directed to a particular OrbixOTS server, which should be running when you issue the command. However, if the server is not running but is registered to start automatically (as a non-persistent server), then issuing a command against that server will cause the server to start. Use the `-server` option to specify the name of the server. As an alternative, you can set the environment variable `ENCINA_SERVER_NAME`. For example:

```
% ENCINA_SERVER_NAME=Bank
% export ENCINA_SERVER_NAME
% otsadmin list tran
```

To specify a server running on a differnet host use the `-host` option. For example:

```
% otsadmin list tran -server bank -host cherub
```

You can abbreviate command names so long as the abbreviation is unambiguous. For example, you can abbreviate the `list tran` command as `l t`. Use the `help` command to obtain information about a particular command. For example:

```
otsadmin> help force tran
```

Table 2.1 shows a complete list of the `otsadmin` commands:

Command	Brief Description
`abort tran`	Abort a transaction.
`add mirror`	Add a volume mirror.
`dump component`	Dump recent traces for a component.
`dump ringbuffer`	Dump recent trace to a file.
`exit`	Exit from the `otsadmin` tool.
`expand mirror`	Add space to a volume mirror.
`expand vol`	Expand a volume to the maximum of underlying mirrors.
`force tran`	Finish a transaction.
`help`	Display help message for a given command.
`list tran`	List unfinished transactions.
`list vol`	List all server volumes.
`query mirror`	Obtain information about a volume mirror.
`query trace`	Query trace mask settings.
`query tran`	Obtain information about a transaction.
`query vol`	Obtain information about a volume.
`quit`	Exit from the `otsadmin` tool.
`remove mirror`	Remove a volume mirror.
`shutdown server`	Shuts down the server.
`trace specification`	Add a trace specification.

Table 2.1: *The otsadmin Commands*

Administering Transactions

You can use the otsadmin tool to list and query the transactions that a server knows about, to abort running transactions, and to force the completion of prepared transactions.

Listing Transactions in a Server

Use the `list tran` command to list all transactions that a server knows about. This command displays, for each transaction, the transaction's local identifier and its current state. The local identifier is an integer value and is used to identify the transaction in the other otsadmin commands. Table 2.2 lists the possible transaction states and their meanings:

State	Meaning
abort_complete	The transaction has been rolled-back and all participants have been informed, but the outcome may not have been reported to the transaction originator. (For example, because there may have been heuristic outcomes.)
aborted	The transaction has been rolled-back.
aborting	The transaction is in the process of being rolled-back.
active	The transaction is currently active in the server.
before_abort	The transaction has been rolled-back but has not yet started the rollback protocol.
commit_complete	The transaction has committed and all participants have been informed, but the outcome may not have been reported to the transaction originator. (For example, there may have been heuristic outcomes.)
committed	The transaction has been committed.
committing	The transaction is in the process of being committed
finished	The transaction has completed.

Table 2.2: *Transaction States*

State	Meaning
`inactive`	The transaction is not currently active in the server.
`none`	The server knows about the transaction, but the server is not a participant in the transaction.
`prepared`	The transaction has been prepared.
`preparing`	The transaction is in the process of being prepared.
`present`	The transaction is active in the server but is not (yet) a participant in the transaction.
`unknown`	The transaction has an unknown state.

Table 2.2: *Transaction States*

Several of these states only exist for short periods and are unlikely to be visible with the `list tran` command. The following example shows the `list tran` command being used to list all transactions that the server named `Bank` knows about:

```
otsadmin> list tran -server Bank
12 inactive (H)
17 active (W)
25 commit_complete
26 inactive
29 prepared
```

Transactions currently holding an OCCS lock are marked with "`(H)`" and transactions waiting for an OCCS lock are marked with "`(W)`" (see transactions 12 and 17 in the above list).

To obtain more information about a particular transaction use the `query tran` command. This takes, as an argument, the local identifier of the transaction being queried. The following example queries transaction 12:

```
otsadmin> query tran 12 -server Bank
globalId: 00010000010c0102420b21fe6a346d61676f
beginner: 0102420b21fe6a346d61676f
```

This displays information such as the global identifier, `globalId`, for the transaction. (Similar to the XID in XA). `beginner` specifies the ID for the application that started the transaction.

Rolling Back Transactions

You use the `abort tran` command to rollback a running transaction. The effect is the same as when an application calls the `rollback()` function. This takes as its argument the local identifier of the transaction to be rolled-back. The following example rolls-back the transaction whose local identifier is 17:

```
otsadmin> abort tran 17 -server Bank
```

This command also allows a complete transaction family to be rolled-back by passing the `-family` option:

```
otsadmin> abort tran 17 -family -server Bank
```

Only transactions that are not yet in the prepared phase of the two-phase commit (2PC) protocol may be rolled-back using this command.

Completing Transactions

When a transaction has completed the prepare phase of its 2PC protocol, you can use the `force tran` command to force the transaction to either commit or rollback. This can be useful if for some reason the 2PC protocol cannot be completed in a timely manner. By forcing the transaction to complete, resources used by the transaction can be released. Use the `force tran` command only rarely, such as after a crash.

WARNING: Use the `force tran` command with caution as it can result in data inconsistencies.

Pass to the command the local identifier of the transaction you want completed. The default behaviour forces the transaction to rollback. For example:

```
otsadmin> force tran 29 -server Bank
```

To force a transaction to commit instead of rolling back, use `-commitdesired` . For example:

```
otsadmin> force tran 29 -commitdesired -server Bank
```

Finally, to force a transaction to complete without necessarily informing all participants, use the `-finish` option. For example:

```
otsadmin> force tran 29 -finish -server Bank
```

otsadmin and SSL

To manage transactions running in SSL enabled OTS servers, a client certificate and private key are required for the `otsadmin` tool. The `otsadmin` tool is built as an OrbixSSL client. Refer to the *OrbixSSL C++ Programmer's and Administrator's Guide* for general information on creating and administering SSL applications.

A configuration variable must be set to enable OrbixSSL to locate the `otsadmin` certificate. This variable is called `IT_CERTIFICATE_FILE` and is located in the `OrbixOTS.otsadmin` scope of the OrbixSSL configuration file. For example, if the `otsadmin` certifcate is located in a file called `/opt/iona/config/repositories/certs/services/orbixots`, there must be a section of the OrbixSSL configuration file like this:

```
OrbixOTS {
   otsadmin {
      IT_CERTIFICATE_FILE="/opt/iona/config/repositories/certs/
services/orbixots";
   };
};
```

OrbixSSL private keys are usually password protected. If the `otsadmin` private key file requires a password, this can be embedded into the `otsadmin` executable using the OrbixSSL `update` utility. For example, if the certificate file is protected using `demopassword` as the password:

```
update otsadmin demopassword 0
```

Logging in OrbixOTS

Each recoverable OrbixOTS server[1] requires a transaction log which is used to record the progress of transactions. The transaction log is only used to record the state of transactions; no application-specific data is stored in these logs. For example, when using a database, the database has its own log data for records

1. A recoverable server is one that manages its own resources using the `Resource` interface, is integrated with an XA resource manager, or acts as a coordinator for transactions.

that are modified during a transaction. The log is used after a crash during recovery to restore the state of transactions that were in progress at the time of the crash.

OrbixOTS permits the use of both ordinary files and raw devices for transaction logs. The recommended minimum size for the transaction log is 8 Mb. Note that logs never really "fill up" as records of completed transactions are no longer needed. This is true whether the transaction rolls back or commits.

A transaction log can be expanded at runtime and mirrored to provide redundancy.

Finally, a C++ server can provide a logging service to other recoverable servers.

Running the otsmklog Tool

The `otsmklog` tool simplifies OTS log, mirror and restart file creation. You can use it to create log files of specific sizes with specific names in specific locations. It also initializes the log by default.

Note, however, that initializing an existing log file or deleting restart or mirror files can destroy valuable logging data, and so disable application recovery in the event of a failure.

Usage:

```
otsmklog [-?hqv] { -p | [-rn] [-s X[K|k|M|m] [-t <restartfile>] [-
m <restartmirrorfile>] } <file>
```

Options:

`-s <size>`	Specify the size of the log. If this option is not supplied the default is 8 Mb. The size may be specifed in kilobytes or megabytes by appending "K" or "M" to the size respectively.
`-r`	Replace an existing file.
`-q`	Quiet mode, do not emit completion status.
`-t`	Specify the name of the restart file.
`-m`	Specify the name of the mirror restart file.

`-p`	Initialize an existing uninitialized log file or partition.
`-n`	Suppress log file initialization.
`-h|?`	Display this help text.
`-v`	Display version information.

Here are some examples of how to use `otsmklog`:

- Create and initialize a log file called `/local/logs/ots.log` using the default names for the restart and mirror files. The files created are `/local/logs/ots.log`, `ots.restart` and `ots.restartmirror`:

  ```
  otsmklog/local/logs/ots.log
  ```

- Create and initialize a log file called `/disk1/ots.log` using `/disk2/ots.r1` as the restart file and `/disk3/ots.r2` as the mirror file:

  ```
  otsmklog -t /disk2/ots.r1 -m /disk3/ots.r2 /disk1/
  ots.log
  ```

- Replace but do not initialize an existing log file called `ots.log`:

  ```
  otsmklog -n -r ots.log
  ```

- Initialize a raw disk partition called `/dev/rdsk/c01t0d0s2`:

  ```
  otsmklog -p /dev/rdsk/c01t0d0s2
  ```

It is recommended that full paths be used for log files created using `otsmklog` rather than relative paths. This is because the path is recorded in the restart files. Log files used for transaction logs should be at least 4Mbytes; the recommended size is 8Mbytes.

Server Initialization

During initialization of a recoverable server, information about the transaction log must be specified. The information consists of the path for the log device and the paths for two restart files. The restart files contain information about the log, including the path for the files it uses. Initially the restart files do not exist, but, once they exist, the path for the log device may be omitted. Deleting the restart files causes the log file to be reinitialized. There are always two restart files for redundancy. If one of the restart files is lost, the other restart file is used to recreate the lost file. "Initializing a Server" on page 37 shows how a programmer initializes an OrbixOTS server.

Raw Disk Logs

A transaction log can be either an ordinary file or a raw disk device. When using ordinary files, the operating system may buffer the output. This can lead to data loss if a crash occurs. However, raw disk devices bypass any operating system buffering. It is recommended that you use ordinary files only during development, and that you use raw disk devices in a production system.

In addition, to reduce the chances of accidental or deliberate corruption of log files, both the log files and the restart files should be owned by the user running the server, and only that user should be able to write to the files or remove them.

Using Volumes and Mirrors

A server's transaction log may be mapped to a set of files or raw disk devices. The transactional log is implemented as a *logical volume,* which is an abstraction of physical storage. A logical volume is mapped to one or more *physical volumes*. (See Figure 2.3.)

Using multiple physical volumes allows the transaction log to be mirrored to add redundancy. A physical volume can consist of multiple ordinary files, raw disk partitions, or a combination of both. An OrbixOTS server (with a log) will always have one logical volume. It can use the Toolkit VOL component for other things such as a data log. In this case there may be more than one volume.

For clarity, the examples in this section all use ordinary files. However, raw disk partitions could be used instead with no changes.

Listing Logical Volumes

You can use the `otsadmin` tool to obtain information about the transaction log used by a recoverable OrbixOTS server. Use the `list vol` command to list the logical volumes in use by the server. (The logical volume for the transaction log is always named `logVol`.) For example:

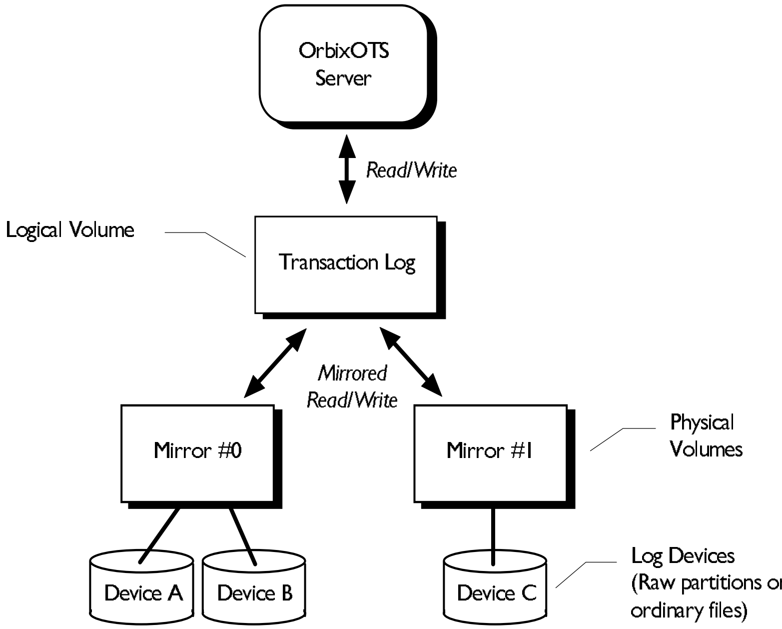


Figure 2.3: *Using Volumes and Mirrors*

```
% otsadmin list vol -server Bank
logVol
```

Querying for Details About Volumes

Use the `query vol` command to obtain information about the `logVol` volume:

```
% otsadmin query vol -server Bank logVol
Volume information for logVol:
Volume size (in pages): 1016
Log free space (in pages): 768
Volume mirrors:
logVol_mirror0
```

The information consists of the size of the volume, the free space in the log, and the names or the physical volumes in use. Initially each server has a single physical volume called `logVol_mirror0`.

Note: Although the physical volume here is called `logVol_mirror0`, there is no mirroring done at this stage.

Use the `query mirror` command to obtain information about a particular physical volume or mirror:

```
% otsadmin query mirror -server Bank logVol_mirror0
Volume mirror logVol_mirror0 occupies the following disks:
/logs/BANK_1.log
```

Extending a Log's Size

Extending the size of the transaction log requires two steps:

1. First, the underlying mirrors must be expanded. Assuming there is a file called `/logs/BANK_2.log`, the "`logVol_mirror0`" volume is expanded by using the `expand mirror` command:

   ```
   % otsadmin expand mirror -server Bank \
   logVol_mirror0 /logs/BANK_2.log
   ```

   Now if you query the mirror, the extra file is shown:

   ```
   % otsadmin query mirror -server Bank
   logVol_mirror0
   Volume mirror logVol_mirror0 occupies the
   following disks:
   /logs/BANK_1.log
   /logs/BANK_2.log
   ```

2. The second step is to expand the logical volume with the `expand vol` command:

   ```
   % otsadmin expand vol -server Bank logVol
   ```

   Now if you query the logical volume, the extra space has been added:

   ```
   % otsadmin query vol -server Bank logVol
   Volume information for logVol:
   Volume size (in pages): 1520
   Log free space (in pages): 1280
   Volume mirrors:
   logVol_mirror0
   ```

Adding a Mirror

You use the `add mirror` command to add a mirror to the transaction log for extra redundancy. The following example assumes that there is a file called `/logs/BANK_3.log`:

```
% otsadmin add mirror -server Bank logVol logVol_mirror1 \
   /logs/BANK_3.log
```

For consistency the name “`logVol_mirror1`” was used for the mirror. Note that the two logs and the two restart files should be on separate disks in a production system.

Now if you query the transaction log volume, the new mirror is listed:

```
% otsadmin query vol -server Bank logVol
Volume information for logVol:
Volume size (in pages): 1520
Log free space (in pages): 1280
Volume mirrors:
logVol_mirror0
logVol_mirror1
```

Note that adding a mirror does not increase the size of the logical volume.

Removing a Mirror

Finally, use the `remove mirror` command to remove a mirror:

```
% otsadmin remove mirror -server Bank logVol logVol_mirror0
```

Using Another Server’s Log

An OrbixOTS server can use the transaction log of another OrbixOTS server provided that both servers are running on the same host. Use the `logServer` attribute to specify the name of the server to use. See “Recoverable Servers” on page 82 for details of how to progam a server to take advantage of this feature.

Controlling Servers

The `otsadmin` tool provides the `shutdown server` command to shutdown a server:

```
otsadmin>shutdown server -server Bank
```

This causes the server to call the `OrbixOTS::Server::exit()` operation.

Tracing Clients and Servers

You can use the `otsadmin` tool to request an OrbixOTS server to output diagnostics. Each component in OrbixOTS has a trace mask that controls what diagnostics (if any) are output.

Querying for Trace Settings

Use the `query trace` command to list the components that can be traced and to see their current trace masks:

```
otsadmin>  query trace -server Bank
log:  none
tran:  none
bde:  none
sutils:  none
tranLog_log:  none
tranLog_tran:  none
restart:  none
vol:  none
util:  none
ots:  none
```

This output indicates that there are ten components and each of the trace masks is empty, so there is no diagnostic output.

Turning Tracing On

Use the `trace specification` command to turn on tracing on one or more components. For example, the following command turns on all tracing for the `ots` component and basic tracing for the `tran` component:

```
otsadmin> trace specification tran=basic,ots=all -server Bank
```

The `trace specification` command takes one parameter: a comma-separated list of component names and the desired tracing level for that component. If you issue the `query trace` command again you can see the desired result:

```
otsadmin>  query trace -server Bank
log:  none
tran:  entry param 0x10000700
bde:  none
sutils:  none
tranLog_log:  none
tranLog_tran:  none
restart:  none
vol:  none
util:  none
ots:  event entry param internal_param internal_entry 0xffffffa0
```

You can use environment variables `ENCINA_TRACE` and `ENCINA_TRACE_VERBOSE` before a server runs to specify a trace. For example:

```
% ENCINA_TRACE="ots=all,tran=basic"
% ENCINA_TRACE_VERBOSE=1
% export ENCINA_TRACE
% export ENCINA_TRACE_VERBOSE
```

`ENCINA_TRACE` specifies the trace specification and `ENCINA_TRACE_VERBOSE` is used to turn tracing on (1) and off (0). You can also trace clients by using these variables.

Dumping Trace Diagnostics

The `dump ringbuffer` command writes the contents of the ring-buffer (an internal structure that holds recent trace diagnostics) to a file. For example, use the following command to write the ring-buffer to the file `TRACE`:

```
otsadmin> dump ringbuffer TRACE -server Bank
```

Normally, the ring-buffer is appended to the output file. However, if you use the `-overwrite` option, the existing file (if any) is overwritten.

If you use the `-binary` option, the output is stored in a shorter binary form.

Note that when an OrbixOTS server crashes, the current contents of the ring-buffer are output to a file called `EncinaTraceBuffer.`*PID*, where *PID* is the process identifier of the server.

You can use the `dump component` command to copy the trace diagnostics for a particular component to the ring-buffer. For example, to copy the `ots` component trace diagnostics, use the following command:

```
otsadmin> dump component ots -server Bank
```

Part II

Programming

3

Getting Started Programming OrbixOTS

This chapter describes how to do a transaction with OrbixOTS. It includes the basic steps needed to develop a distributed application with OrbixOTS. The chapter shows the most typical use of OrbixOTS: XA database integration using the indirect/implicit mode.

This chapter assumes that you are familiar with creating client and server applications with Orbix.

Overview

Servers and clients are implemented as objects in OrbixOTS applications. The OrbixOTS interface supplies a server class and a client class that you use to initialize servers and clients. This chapter describes how to do the following key tasks:

1. Specify transactional classes with the Interface Definition Language (IDL). You use the IDL to define the interface to transactional objects.
2. Write an OrbixOTS server. After initializing a server, you can use operations to register XA resource managers, make server objects available to clients, and listen for client requests.
3. Write an OrbixOTS client. The transaction demarcation is in the client. You begin a transaction, do application-specific operations within the transaction, and commit or rollback the transaction.

4. Complete a sample application by compiling, linking, and running it.

Specifying Transactional Classes

You define interfaces for objects in OrbixOTS applications in a similar way to those defined for Orbix applications. Objects that participate in transactions or make transactional requests on other objects are called *transactional objects*. You use the CORBA Interface Definition Language (IDL) to specify interfaces to transactional objects. The operations defined by an object’s interface are used to communicate between the client and server.

You use the Orbix IDL compiler to generate the C++ code classes for each interface.

The following code shows example interface definitions for transactional objects. This `TransBank` application is a simple OrbixOTS application that shows the transfer of money between two bank accounts:

```
    //IDL code
1   #include <OrbixOTS.idl>
    exception DBError { string reason; };
    const long AccountNameLen = 20;
    typedef string<AccountNameLen> AccountName;

2   interface TransAccount : CosTransactions::TransactionalObject {
      void makeLodgement(in float amount)
        raises (DBError);

      void makeWithdrawal(in float amount)
        raises (DBError);

      void query(out AccountName accName, out float accBalance)
        raises (DBError);
    };

    interface TransBank : CosTransactions::TransactionalObject {
      typedef sequence<long> AccountNumSeq;

      TransAccount newAccount(in AccountName accName,
          in float accBalance,
          out long accNumber)
```

```
    raises (DBError);

  TransAccount lookupAccount(in long accNumber)
    raises (DBError);

  void getAllAccounts(out AccountNumSeq accounts)
    raises (DBError);
};
```

The code is described as follows:

1. The interface file for a transactional object must include `OrbixOTS.idl`, the IDL file that defines the OMG OTS interfaces.
2. You generally make an object transactional by specifying that its interface is derived from the class `CosTransactions::TransactionalObject`.

You implement this interface when you write the OrbixOTS server in the next section.

Note: The use of oneway operations within a transaction is not permitted. Interfaces inheriting from `TransactionalObject` should not use the `oneway` keyword.

Writing an OrbixOTS Server

This section covers the basic issues involved in writing OrbixOTS servers. It describes how to create server objects and how to implement the server's interface for transaction objects.

Initializing a Server

This section describes how to initialize and terminate an OrbixOTS server application. OrbixOTS server applications typically perform the following basic steps:

1. Create one OrbixOTS server class instance to manage the server.
2. Register any resource managers required by the server (optional).
3. Create one or more CORBA server objects.

4. Listen for requests.
5. Terminate the server.

The following example illustrates typical code for a simple server that uses Oracle's XA interface. Note that OrbixOTS provides the `OrbixOTS.hh` header file for use with C++ applications. This header file automatically includes the Object Transaction Service (OTS) and the Object Concurrency Control Service (OCCS) interface declarations.

```
      //C++ code
      #include <iostream.h>
      #include <stdio.h>
      #include <OrbixOTS.hh>
      #include "TransBank_i.hh"
      extern struct xa_switch_t xaosw;// The Oracle XA switch.

      main(){
         ...
1        OrbixOTS::Server_var ots = OrbixOTS::Server::IT_create();
         ots->serverName("TransBank/oracle");
         ots->logDevice("ots.log");
         ots->restartFile("ots.r1");
         ots->mirrorRestartFile("ots.r2");
         ...
         // Build an XA open string. Requires an Oracle account &
         // password.
2       int rm_id = ots->register_xa_rm(&xaosw, openString, "", 0);
3       ots->init();
         ...
        try {
4             ots->impl_is_ready();
        } catch (const CORBA::SystemException &sysEx) {
          cerr << "Unexpected system exception" << endl;
          cerr << sysEx << endl;
          ots->exit(1);
        } catch (...) {
          cerr << "Exception raised" << endl;
          ots->exit(1);
        }
5       ots->shutdown();
      }
```

The code is described as follows:

1. The server application creates an instance of the `OrbixOTS::Server` pseudo interface to represent the application server. Only one instance is permitted per server application. You specify startup information explicitly for servers using the following attributes:

`serverName`	A name for the server. This is the name usually passed to `CORBA::Orbix::impl_is_ready()`.
`logDevice`	A name of an available file or device that the server can use to log information.
`restartFile`	Name for one of the server's restart files. This file contain information about the log. If this file already exists, the log parameter is not required. If you run the server for the first time and this file does not exist, the log is formatted and the file is created.
`mirrorRestartFile`	Name for a copy of the server's restart file.

   If you do not use logging, a server could not recover if a failure were to occur. However, the restart files and log are not required. If the server is a shared Orbix server (started dynamically via the Orbix daemon), the server name is not required either.

2. After the server instance is created, a server can register any resource managers that it requires. This step is optional. The function `OrbixOTS::Server::register_xa_rm()` registers XA-compliant resource managers and makes the server recoverable. This example registers Oracle as an XA resource manager with OrbixOTS. If no XA-compliant resource managers are required for the application but you still want the server to be recoverable, you can instead use the function `OrbixOTS::Server::recoverable()`.

   The four parameters are the XA switch, the database open string, the close string, and a boolean specifying whether the XA library can support multiple threads. This application does not support multiple threads so 0 is passed.

3. The next step is to initialize the underlying OrbixOTS components and services by calling the `OrbixOTS::Server::init()` function. This must be done after any XA resources have been registered.

Typical Orbix programming requires that the server create one or more server objects (`bank`) to handle incoming requests. To create a server object in an OrbixOTS server application, you simply create an instance of the implementation class you defined for the TransBank interface.

4. After server objects are created, you must start the server listening for requests. You can use either the `OrbixOTS::Server::impl_is_ready()` function or the `CORBA::Orbix.impl_is_ready()` function that Orbix provides to start the server listening.

   The `OrbixOTS::Server::impl_is_ready()` function takes an optional parameter that sets the concurrency mode for the server and creates a pool of threads for OrbixOTS requests. This parameter determines whether transactions and incoming requests are serialized at the server. Modes include:

   ```
   concurrent
   serializeRequests
   serializeRequestsAndTransactions
   ```

   The default value is `serializeRequestsAndTransactions` which permits only one transaction to access the server at a time. This is the most restrictive concurrency mode and used in this example. Passing in the value `concurrent` permits concurrent requests and transactions at the server.

   If the server has not already called the `OrbixOTS::Server::init()` function, the `OrbixOTS::Server::impl_is_ready()` function initializes OrbixOTS before the server begins listening for requests. As no timeout is specified, the default is used.

5. Terminating a server stops the server from listening for incoming requests and stops the underlying OrbixOTS services. Many servers are shut down administratively, but some shut down because of a system failure. If you need to terminate the server application in your program, use the `OrbixOTS::Server::shutdown()` function. This function shuts OTS down cleanly.

   The example code also shows the `exit()` function being used to terminate a server application when an exception is thrown. OrbixOTS applications use the standard C++ try-catch exception-handling mechanism to throw (raise) and catch exceptions when error conditions occur, rather than testing status values to detect errors. This exception-handling mechanism is also used to integrate CORBA exceptions into OrbixOTS.

Implementing Transactional Classes

To implement the transactional classes for the interface, you must define a C++ class and class functions corresponding to the interface definition in the IDL file.

OrbixOTS servers can use either of the Orbix approaches to implement the IDL interface. These are the Basic Object Adapter implementation (BOAImpl) or the TIE approaches. If you use the BOA approach, your implementation class must inherit from the BOA class defined in the header file generated by the IDL compiler. Refer to the Orbix documentation for details on using these approaches.

The following code implements the interfaces in `TransBank.idl`. The code shows an example class definition that uses the BOAImpl approach. This is standard Orbix coding.

```
#include "TransBank.hh"

class TransBank_i : public virtual TransBankBOAImpl {
public:
  TransBank_i();
  virtual ~TransBank_i();
  TransAccount_ptr newAccount(const char* accName,
              float accBalance,
              CORBA::Long& accNumber,
              CORBA::Environment& IT_env)
    throw (DBError);

  TransAccount_ptr lookupAccount(CORBA::Long accNumber,
              CORBA::Environment& IT_env)
    throw (DBError);

  void getAllAccounts(AccountNumSeq*& accounts,
     CORBA::Environment& IT_env)
    throw (DBError);
};

class TransAccount_i : public virtual TransAccountBOAImpl {
public:
```

```
  TransAccount_i(CORBA::Long accNumber);

  virtual ~TransAccount_i();

  void makeLodgement(CORBA::Float amount,
      CORBA::Environment& IT_env)
    throw (DBError);

  void makeWithdrawal(CORBA::Float amount,
      CORBA::Environment& IT_env)
    throw (DBError);

  void query(AccountName& accName, CORBA::Float& accBalance,
        CORBA::Environment& IT_env)
    throw (DBError);

private:
  // Account number used as a key in SQL statements to access the
  // account's data.
  CORBA::Long m_accountNumber;
};
```

You must include the header file (`TransBank.hh`) generated by the IDL compiler. The code defines the `TransAccountBOAImpl` class from which `TransAccount_i` is derived. The implementation class must provide virtual function definitions for the functions specified in the interface. It can also define additional member functions, data, constructors, and destructors.

The file `TransBank_i.cc` contains C++ code that implements the functions of the `TransAccount_i` class. This code is standard C++ code not shown here.

The functions that actually access the database are defined in the `db_bank.h` file as follows:

```
/* C code */
#define DB_ACC_NAME_LEN  20
#define DB_SUCCESS ((char*)0)
char* db_create_account(char* accName, float accBalance,
                        long* accNumber);
char* db_lookup_account(long accNumber);
char* db_query_account(long accNumber, char accName[],
                       float* accBalance);
char* db_make_lodgement(long accNumber, float amount);
char* db_num_accounts(int* num_accounts);
```

```
char* db_get_accounts(long accounts[], int max_accounts);
void db_free_error(char* error);
```

These are declarations for functions that provide access to the database. Access is done with embedded SQL and each function returns a status string indicating the result of the SQL code.

For example, this is the `db_lookup_account()` function for Oracle:

```
/* Check that an account exists. */
char* db_lookup_account(long accNumber)
{
  EXEC SQL BEGIN DECLARE SECTION;
  long db_accNum;
  EXEC SQL END DECLARE SECTION;

  /* Fill out the database variable. */
  db_accNum = accNumber;

  /* Just check that the account exists. */


  EXEC SQL
    SELECT ACC_NUM
      FROM ACCOUNTS
        WHERE ACC_NUM = :db_accNum;

  /* Check for any errors. */
  if (sqlca.sqlcode != 0 ) {
    return get_sql_error();

  } else {
    return DB_SUCCESS;
  }
}
```

On success, the value `DB_SUCCESS` is returned; on failure, the string contains a text description of the error. These functions are implemented as standard embedded SQL code.

Writing an OrbixOTS Client

Applications use transaction processing to ensure that data remains correct, consistent, and persistent. Transaction processing in an object-oriented distributed environment enables distributed objects to meet the same requirements. This section describes how to write an OrbixOTS client application that manages a transaction. In addition to performing tasks that are specific to your application, OrbixOTS client applications must perform the following basic steps:

1. Initialize the underlying OrbixOTS services for the client.
2. Do a transaction that includes:
   i. Begin a transaction.
   ii. Perform server requests within a transaction.
   iii. End a transaction.
3. Terminate the client application.

Each of these steps are described in turn in the following subsections.

Initializing a Client

The following code demonstrates the initialization programming required before a client begins a transaction.

```
    #include <iostream.h>
    #include <stdlib.h>
    #include <ctype.h>
1   #include <OrbixOTS.hh>
    #include "TransBank.hh"

    static void newAccount(TransBank_var bank);
    static void queryAccount(TransBank_var bank);
    static void doLodgement(TransBank_var bank);
    static void doWithdrawal(TransBank_var bank);
    static void doTransfer(TransBank_var bank);
    static void displayAccounts(TransBank_var bank);

    main(int argc, char* argv[])
    {
      TransBank_var bank; // pointer to the bank object used in demo
```

```
   ...
  CORBA::ORB orb = CORBA::ORB_init()
2 OrbixOTS::Client_var ots = OrbixOTS::Client::IT_create();
3 ots->init();
4 CORBA::Object_var obj =
     orb->resolve_initial_references("TransactionCurrent");
  CosTransactions::Current_ptr current =
     CosTransactions::Current::_narrow(obj);
5 current->set_timeout(30);
   ...
```

The code is described as follows:

1. Applications must include the appropriate header files to define the data types, classes, functions, macros, and constructs used in OrbixOTS and the runtime environment. OrbixOTS provides the `OrbixOTS.hh` header file for use with C++ applications.
2. Create client pseudo object using the function `OrbixOTS::Client::IT_create()`.
3. Initialize the client pseudo object using the function `OrbixOTS::Client::init()`. This initializes all of the necessary underlying OrbixOTS components and services.
4. Obtain a reference to the OTS Current object.
5. OrbixOTS transactions have a finite (and configurable) timeout. If the transaction is not completed within this time, it is automatically rolled back. In addition, some XA interfaces implement a timeout, so that those transactional objects that use XA resource managers can have their work automatically rolled back if the transaction is not completed within the timeout period.

After the typical Orbix client is initialized during execution, it uses an instance of the client stub class, called a client proxy object, to bind to a remote object. The client stub includes member functions for the operations defined in the interface. Once a proxy object is bound to a remote object, calling a function on the proxy object invokes the corresponding function on the remote object.

The client stub also defines a member function for each operation defined in the interface. If the interface is defined in the IDL file as transactional, the functions must be called within the scope of a transaction; otherwise, an exception will be thrown.

Doing a Transaction

After initialization, the client can begin a distributed transaction and make remote calls within the transaction. The following code shows a typical use of the `Current` class member functions to begin and end a transaction. This is an example of the *indirect-implicit* model of programming. The *indirect-implicit* model is the preferred way to manage a transaction because it allows OrbixOTS to manage a transaction in a consistent manner. The *direct-explicit* programming model is very flexible, but it requires more complex and careful programming.

```
void doTransfer(TransBank_var bank)
{
  TransAccount_var srcAccount;
  TransAccount_var destAccount;
  CORBA::Long      srcAccNumber;
  CORBA::Long      destAccNumber;
  CORBA::Float     amount;

  // Do input
  ...
  try {
    // Create a transaction
1   current->begin();

    // Lookup the accounts
    srcAccount = bank->lookupAccount(srcAccNumber);
    destAccount = bank->lookupAccount(destAccNumber);

    // Perform the transfer
2   srcAccount->makeWithdrawal(amount);
    destAccount->makeDeposit(amount);

    // Commit the transaction.
3   current->commit(TRUE);

    cout << " Done." << endl;
    cout << " Transferred " << amount
         << " from account " << srcAccNumber
         << " to account " << destAccNumber << endl;

  } catch (CORBA::TRANSACTION_ROLLEDBACK) {
    cerr << " Unable transfer (transaction rolledback)" << endl;
```

```
    } catch (const DBError ex) {
4    // Call rollback to disassociate transaction from thread.
     current->rollback();
    }

    ... // additional exceptions caught
```

The code is described as follows:

1. Clients can begin a transaction by calling the function `CosTransactions::Current::begin()`. The function does not return a value. The `Current` object can be used to manage different concurrent transactions, one per calling thread.

   Use code such as the following to obtain an instance of the current object:

   ```
   CORBA::ORB_var orb = ...
   CORBA::Object_var obj =
      orb->resolve_initial_references
          ("TransactionCurrent");
   CosTransactions: :Current ::_narrow(obj);
   ```

2. The application-specific functions `lookupAccount()`, `makeWithdrawal()`, `makeDeposit()`, and `balance()` execute within the scope of the transaction. If a call to any of these functions throws an exception (either explicitly or as a result of a communications failure, for example), the exception can be caught at the client and the transaction is rolled back.
3. Call the `CosTransactions::Current::commit()` function to commit the current transaction. This call ends the transaction and starts the two-phase commit processing. The transaction is committed only if all of the participants in the transaction agree to commit. This particular example has only one participating server.

   The association between the transaction and the client process ends when the client calls this function or the `CosTransactions::Current::rollback()` function.
4. Call the `CosTransactions::Current::rollback()` function to roll back the current transaction.

Terminating a Client

When a client finishes all transaction related activities it must shutdown OrbixOTS before exiting. This is done using the `OrbixOTS::Client::shutdown()` operation:

```
OrbixOTS::Client_var ots = ...
ots->shutdown();
```

This operation must be used to ensure all the underlying OrbixOTS services are terminated cleanly. All outstanding transactions in progress are completed (either committed or rolled back).

Alternatively the client can use the operation `OrbixOTS::Client::exit()` which shuts down OrbixOTS and exits. The operation takes one argument, which is an integer status value that is returned to the calling environment.

Completing an Application

A `Makefile` is provided to build the TransBank application. However, this section explains some details about compiling and linking server and client portions of an OrbixOTS application. This section ends by showing how to run the TransBank application.

Compiling and Linking a Server

A simple server application is made up of the following:

- A source file for the server (`Server.cc`) that listens for remote requests; this file must include the header file defining the implementation class for the server interface (`TransBank.hh`).
- A source file that implements the functions for the server interface (`TransBank_i.cc`).
- A server stub file generated by the IDL compiler (`TransBankS.cc`).
- Other application files that implement the interface (`db_bank.h`, `oracle_bank.pc`).

Build the server application by compiling the source file for the server, the source file for the implementation class, and the server stub file and linking them with the appropriate OrbixOTS libraries.

Server applications must link with the following libraries:

- `EncinaServerOrbix`
- `EncServer_nodce`
- `Encina_nodce`
- Orbix multithreaded library `orbixmt`

Before an OrbixOTS server runs, you must specify a name for the server. If the server is started dynamically via the Orbix daemon, you specify the server name by registering the server with the Orbix implementation repository. If the server is started manually (CORBA persistent server), you must specify the server name in the code by using the `serverName` attribute of the `OrbixOTS::Server` pseudo object.

Compiling and Linking a Client

A simple client application is made up of the following:

- A source file for the client that initiates remote requests (`SimpleClient.cc`). This file must include the IDL-generated header file containing the client class definition corresponding to the interface name (`TransBank.hh`).
- A client stub file generated by the IDL compiler (`TransBankC.cc`).

Build the client application by compiling the source file for the client and the client stub file, and linking them with the appropriate OrbixOTS libraries. Note that the files with `.cc` and `.hh` extensions denote C++ source and header files. Client applications must link with the following libraries:

- `EncinaClientOrbix`
- `Encina_nodce`
- Orbix multithreaded library `orbixmt`

Before you run a client application, the server must be registered with the Orbix daemon.

Running the TransBank Application

After you build the TransBank application, follow these steps at the command prompt to run it. This example uses the Oracle version of the application and assumes that a local Oracle installation exists. You run other versions in a similar manner.

1. Create the Oracle tables:

   ```
   % sqlplus scott/tiger @initdb.sql
   ```

   This creates two tables ACCOUNTS and ACCOUNT_NUMBER and populates the ACCOUNTS table with some bank accounts. You can view the two accounts numbered 1002 and 1003.

   ```
   % sqlplus scott/tiger
   SQL> select * from ACCOUNTS where ACC_NUM = 1002
   or ACC_NUM = 1003;

      ACC_NUM ACC_NAME             ACC_BALANCE
   ---------- -------------------- -----------
         1003 Linda                        400
         1002 John                         300
   ```

2. Create the OrbixOTS transaction log:

   ```
   otsmklog ots.log
   ```

   This creates the file ots.log, which is used by OrbixOTS to keep track of the progress of transactions. Note that in a production system a raw device should be used instead of an ordinary file.

3. Register the server with Orbix:

   ```
   % putit TransBank/oracle -persistent
   ```

4. Run the server:

   ```
   % oraclesrv &
   ```

   Note that starting the server the first time may be a slow process because the transaction log needs to be initialized.

5. Run the client and transfer 50 units from account 1002 to account 1003 (user input is in **bold**):

   ```
   %  ./clients/simpleclt oracle

    ** OrbixOTS TransBank Demo
    ** SimpleClient/implicit
   ```

```
* New Account      [n]
* Query Account    [q]
* Make Lodgement   [l]
* Make Withdrawal  [w]
* Transfer         [t]
* Display Accounts [d]
* Exit             [e]

Enter choice: t

TRANSFER
Enter source account number      : 1002
Enter destination account number : 1003
Enter amount to transfer         : 50

Done.
Transferred 50 from account 1002 to account 1003
```

6. Examine the database to verify that the transfer was successful.

```
% sqlplus scott/tiger
SQL> select * from ACCOUNTS where ACC_NUM = 1002
or ACC_NUM = 1003;

   ACC_NUM ACC_NAME             ACC_BALANCE
---------- -------------------- -----------
      1003 Linda                        450
      1002 John                         250
```

4

Programming with the Java Classes

OrbixOTS provides support for Java clients and servers using OrbixWeb. This allows Java clients to create, commit, and rollback transactions and to invoke operations on OrbixOTS Java servers.

This chapter examines the architecture of Java OrbixOTS and describes how to perform distributed transaction operations in Java with Orbix OTS. It also describes the steps involved in building a distributed transactional Java application.

This chapter assumes that you are familiar with specifying transactional interfaces using IDL (see Chapter “Getting Started Programming OrbixOTS” on page 35) and that you are familiar with creating simple distributed client server applications with OrbixWeb professional.

Architecture

Servers and clients are implemented as objects in OrbixOTS Java applications. OrbixOTS for Java provides a client and server implementation in the `IE.Iona.OrbixWeb.CosTransactions` package that allows clients and servers to initialize OTS and participate in distributed transactions.

The architecture of a Java OrbixOTS application is shown in Figure 4.1. Java clients must make use of one or more OrbixOTS C++ servers both to create transactions and to coordinate distributed transactions. The steps are:

1. The Java client uses an OrbixOTS "factory" server to create and terminate transactions. This can be done using both the direct mode (using the `TransactionFactory` and `Terminator` interfaces) and the indirect mode (using the `Current` interface).
2. Once a transaction has been created, the Java client can invoke operations on objects in an OrbixOTS Java server.
3. When the Java client commits or rolls-back the transaction, the factory server coordinates the 2PC protocol which will involve the recoverable server.

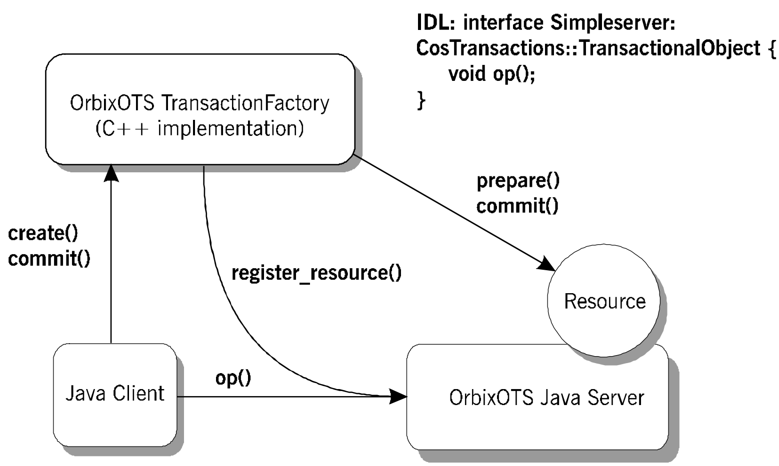


Figure 4.1: *Architecture for Java Applications*

Unlike the scenario when SimpleServer is implemented in C++, the separation between the factory server and the Java server is physical—they cannot be the same object because OrbixOTS Java servers do not support a local implementation of the `TransactionFactory` interface.

Specifying Transactional Classes

You define interfaces for objects in OrbixOTS applications in a similar manner as for Orbix applications. Objects that participate in transactions or make transactional requests on other objects are called *transactional objects*. You use the CORBA Interface Definition Language (IDL) to specify interfaces to transactional objects. The operations defined by an object's interface are used to communicate between the client and server.

You use the OrbixWeb IDL compiler to generate the Java code classes for each interface.

The following code shows example interface definitions for transactional objects. This `TransBank` application is a simple OrbixOTS application showing the transfer of money between two bank accounts and a query operation to retrieve the account's name and current balance:

```
    //IDL code
1   #include <OrbixOTS.idl>
2   exception DBError { string reason; };
    const long AccountNameLen = 20;
    typedef string<AccountNameLen> AccountName;

3   interface TransAccount : CosTransactions::TransactionalObject {
      void makeLodgement(in float amount)
        raises (DBError);

      void makeWithdrawal(in float amount)
        raises (DBError);

      void query(out AccountName accName, out float accBalance)
        raises (DBError);
    };

4   interface TransBank : CosTransactions::TransactionalObject {
      typedef sequence<long> AccountNumSeq;
      TransAccount newAccount(in AccountName accName, in float
              accBalance,
              out long accNumber)
        raises (DBError);

      TransAccount lookupAccount(in long accNumber)
```

```
    raises (DBError);

  void getAllAccounts(out AccountNumSeq accounts)
    raises (DBError);
};
```

The code is described as follows:

1. The interface file for a transactional object must include `OrbixOTS.idl`, the IDL file that defines the OMG OTS interfaces.
2. The exception `DBError` is used to indicate some sort of failure in the backend. All operations can raise a `DBError`. It contains a single string that represents a textual description of the error.
3. You generally make an object transactional by specifying that its interface is derived from the class `CosTransactions::TransactionalObject`.
4. A description of a simple transactional bank. This interface allows new accounts to be created and existing accounts to be looked up. All accounts are identified by a unique account number. There is also an operation to retrieve a list of all accounts in the bank.

Writing a Java Server

This section describes the basic steps involved in writing a Java server. You can implement a normal Java transactional server by following these steps:

1. Create a server instance.
2. Create an implementation object.
3. Perform recovery for any resource that you implement (optional).
4. Listen for requests.
5. Terminate the server.

OrbixOTS for Java provides a JAR file called `OrbixOTS.jar` with all the classes required for programming with transactions. It contains the IDL compiler generated stub code for the Object Transaction Service (OTS) and the Object Concurrency Control Service (OCCS). The corresponding classes must be imported into your code in the normal way to make them available by name. The following code sample demonstrates the steps required in creating a server instance;

```
1   import IE.Iona.OrbixWeb.CosTransactions.Server;
    public static void main(...) {
       ORB orb    = ORB.init(...);
2      Server ots = Server.IT_Create();
3      ots.init();

4   //do recovery now
       // process events
       ...

5      ots.shutdown();
    }
```

The following comments refer to numbered lines in the code sample above:

1. Import the `Server` class from the package `IE.Iona.OrbixWeb.CosTransactions` to make it accessible by name. All the OrbixOTS proprietary interfaces are in the `IE.*` packages and the standard interfaces can be found in the corresponding `org.omg.*` packages.
2. Create a non-initialized instance of the `Server` class using the static `Server.IT_create()` method. Only one instance of this class is permitted per ORB. In this case the `Server` instance is associated with the default `ORB _Corba.Orbix`. Another variant of `IT_create()` takes an ORB instance as a parameter.
3. Initialize the Java OTS Server instance. This installs the required interceptors for transactional context propagation and creates an instance of the transaction current interface. This step must be completed before you attempt transactional operations.
4. If the server implemented a recoverable resource then it should do recovery of that resource at this stage; before you make the server available to clients. See Chapter 6, "Writing a Recoverable Resource" for more details on writing recoverable servers.
5. The server is about to exit because event processing has returned in the main thread. The OrbixOTS Java server instance is shut down in this example. The `shutdown()`operation rolls back any outstanding transactions in the server.

Writing a Transactional Java Client

This section describes the basic steps involved in writing a Java client. You can implement a normal Java transactional client by following these steps:

1. Create a client instance.
2. Obtain a reference to the TransactionCurrent.
3. Perform transactions.
4. Terminate the client.

The following code sample demonstrates the steps required in creating an OrbixOTS for a Java transactional client.

```
1    import IE.Iona.OrbixWeb.CosTransactions.Client;
     import org.omg.CosTransactions.Current;
     import org.omg.CosTransactions.CurrentHelper;

     public static void main(...) {
        ORB orb    = ORB.init(...);
2       Client ots = Client.IT_Create();
3       ots.init();

        Object current_object = null;
4       current_object =
              orb.resolve_initial_references("TransactionCurrent");
        Current current = CurrentHelper.narrow(current_object);

5       current.begin();
        // do some transactional work
                     ...
6       current.commit(false);

7       ots.shutdown();
     }
```

The following comments refer to numbered lines in the code sample above:

1. Import the appropriate packages to make classes easily accessible by name.

2. Create a non-initialized instance of the `Client` class using the static `Client.IT_create()` method. Only one instance of this class is permitted per ORB. In this case the client instance is associated with the default `ORB _Corba.Orbix`.
3. Initialize the Java OTS `Client` instance. This installs the required interceptors for transactional context propagation and creates an instance of the transaction current interface. This step must be completed before attempting transactional operations.
4. Obtain a reference to the `TransactionCurrent` pseudo object through `resolve_initial_references()`. This is the CORBA standard way to obtain such a reference. The instance returned is a CORBA object, and so must be narrowed to the `CosTransaction` current type using the helper class.
5. Begin a transaction using `TransactionCurrent`. This is the simplest way to create a transaction. The client then invokes operations on transactional servers. In this implicit/indirect mode the transaction is automatically propagated to the servers by the OTS.
6. Commit the transaction, when all transactional operations are complete. `commit()` has a `false` parameter, indicating that heuristic outcomes are not of interest for this transaction.
7. Terminate the OTS client before exiting. This ensures that all outstanding transactions (if any) are rolled back.

The *OrbixWeb Programmer's Guide* contains more information on `resolve_initial_references()`.

Building and Running a Java Server/Client

You can build and run a transactional Java application in the same way as a non-transactional OrbixWeb application with two exceptions:

1. You must start a `TransactionFactory` (`otstf` - see Appendix B, “The OrbixOTS Transaction Factory”).
2. You must ensure that the OrbixOTS for Java supplied JAR file (`OrbixOTS.jar`) is in your classpath.

There is no need to compile the CosTransactions.idl file as the stubs and skeletons are already in the supplied JAR file.

Use the -jP command-line argument when using the OrbixWeb IDL compiler to generate code for your own Idl interfaces. This specifies the package prefix for the CosTransactions interface. The generated code contained in the supplied JAR file places standard interfaces in the org.omg.* package, therefore, it is necessary to specify that this is the case to the IDL compiler. For example:

```
idl <additional options> -jPCosTransacitons=org.omg <new
transactional interface>.idl
```

This ensures that references to TransactionalObject in your code refer to the stubs/skeletons in the supplied JAR file.

Part III

Advanced Programming

5

Controlling Transactions

This chapter covers a number of programming topics in OrbixOTS transaction programming.

Topic include the following:

- An overview of a do-it-yourself style of transaction programming that includes direct transaction context management, explicit transaction propagation, and manual resource manager registration.
- Using the direct model of transaction context management.
- Using the explicit model of transaction propagation.
- Suspending and resuming transactions.
- Nested transactions.
- Threading transactions.

An Overview of Transaction Programming Models

When programming with OrbixOTS, you can adopt one of two transaction programming models:

"all-in-OrbixOTS" Programming Model

The all-in-OrbixOTS model is associated with indirect context management, implicit transaction propagation, and automatic (via XA) resource manager registration. This model has many advantages, including:

- Uses the XA interface or native database support.
- Uses a linked OrbixOTS library.
- Recovery is automatic.
- It is easy to upgrade an interface by simply inheriting from the `TransactionalObject`.

The features of the all-in-OrbixOTS model are shown in the example in "Getting Started Programming OrbixOTS" on page 35.

"do-it-yourself" Programming Model

The do-it-yourself (DIY) model uses direct context management, explicit propagation, and manual resource registration. The advantages of this model include:

- It is possible to have parts of an application not linked with the OrbixOTS library.
- Improved efficiency when applications use multiple databases. For example, the all-in-OrbixOTS model may lead to unnecessary overhead.
- Tunable and flexible resource manager registration.

Later sections in this chapter describe how to use features of the do-it-yourself style. These include using the `TransactionFactory` class for direct transaction context management and using function parameters for explicit transaction propagation.

Using Direct Context Management

OrbixOTS provides two interfaces for creating transactions: the `Current` pseudo object, or the `TransactionFactory`. Use of the `Current` interface is simpler, but it does require that the process be linked with an OrbixOTS library. Internally, the `Current` interface may call the `TransactionFactory`, but this is not necessarily the case. Because of this potential relationship, the OMG OTS specification labels the use of `TransactionFactory` as *direct context management*; the use of `Current` is referred to as *indirect context management*.

Creating Transactions

There is a significant difference between these two styles of transaction creation. This becomes more apparent in discussions of the two styles of transaction propagation: explicit and implicit. (See “Using Explicit Transaction Propagation” on page 67.)

Normally, server objects are created before the server begins listening for requests from clients. Server objects can also be created dynamically; you can use a factory object to create server objects while the server is listening for requests. A factory object is designed to create other objects that are managed by the server.

The `create()` function for the `TransactionFactory` as follows:

```
// Get a reference to a Transaction Factory
CosTransactions::TransactionFactory_var factory =
   CosTransactions::TransactionFactory::
    _bind(“TransactionFactory:SomeServer”, “SomeHost”);

// Create a transaction
CosTransactions::Control_var myControl =
   factory->create(60);
```

In this example, a `TransactionFactory` proxy is bound to an object exposed by an OrbixOTS Transaction Manager identified by the server name, `SomeServer`, on which the create function is subsequently called (60 indicates a transaction timeout of 60 seconds). An object of type `Control` is returned; this represents the created transaction.

Ending Transactions

There are two ways for the transaction to end: either indirectly, using the `Current` interface, or directly, using the `Terminator` interface. There are also two types of completion: commit and rollback, both possible via either interface. The following code shows how to directly terminate a transaction by rolling it back.

```
// rollback the transaction
CosTransactions::Terminator_var myTerminator =
     myControl->get_terminator();
myTerminator->rollback();
```

Transactions can be rolled back by the runtime system or by any participant in a distributed transaction. Communications or data access failures are the most common cause of runtime system aborts.

A remote server object may instead simply mark the transaction for rollback by calling `rollback_only()` on `Current` (indirect) or on `Coordinator` (direct). You typically use `rollback_only()` if your server is not the originator of the transaction. This does not actually rollback the transaction, but it ensures that even if the originating server calls `commit()`, the only possible outcome for the transaction is a rollback. However, this function is rarely needed because OrbixOTS permits the server to call the `rollback()` function directly.

Make sure that your application ends each transaction once, and only once, by either committing or rolling back the transaction. This is particularly important if your application uses nested transactions. For example, if a manager function aborts a nested transaction instead of raising an exception, the current thread is disassociated from the nested transaction and associated with the parent transaction. In addition, execution of the statements in the try block enclosing the nested transaction continues until an exception is thrown. If no exception is thrown before the `Current::commit()` function at the end of the try block is invoked, the function attempts to commit the parent transaction. To ensure that your application behaves as expected, you must manage the transaction context of the current thread carefully.

Using Explicit Transaction Propagation

The OrbixOTS interface defines classes for two transaction propagation modes: implicit and explicit. In the *implicit mode*, the client implicitly passes the transaction context that defines a transaction to an object by associating the context with the calling thread. This model is used in the example in “Getting Started Programming OrbixOTS” on page 35. In the *explicit mode*, the transaction context must be passed explicitly to an object as a parameter in a function call.

The implicit mode provides a simpler interface for coding transactional applications than the explicit mode. Because most applications use the preferred implicit mode, the primary focus of this guide is on using this mode. This section provides only a brief introduction to the explicit mode.

In the explicit mode, the remote object’s IDL simply includes an input parameter of type `Control` in functions that involve transactional updates. The `Control` associated with a transaction begun using the `Current` class can be obtained by calling `get_control()` on `Current`. The server object then uses the passed object to register its interest in transaction completion. The following example shows a fictional `explicitOTSServer` interface with one transactional function, `doUpdate()`.

```
// IDL
// Explicit interface example
interface explicitOTSServer
{
   ...
   void doUpdate(in CosTransactions::Control ctrl,in short value);
   ...
}
```

The explicit mode requires that the interface designer know which functions may need to be performed in the context of a transaction. Considerable repercussions can occur if an existing interface is to be made transactional, as many functions may have to be changed to accommodate an extra parameter. On the other hand, the explicit interface allows individual functions to be made transactional, and has the advantage that neither the transaction receiver nor propagator need be linked with an OrbixOTS library.

The implicit mode does not change the signatures of existing functions, but it does require that all functions of a given interface be made transactional, and that the relevant processes be linked with an OrbixOTS library to implement the `Current` interface and the transaction propagation functionality.

Suspending and Resuming Transactions

You can suspend a transaction by invoking the `Current::suspend()` function in the context of the current transaction. The function returns a pointer of type `Control_ptr`, which is a pointer to a `Control` class instance. The `Control` instance represents the transaction context associated with the current thread. Note that the resources that a transaction is accessing remain locked while the transaction is suspended.

To resume the suspended transaction, call the `Current::resume()` function, passing it the pointer returned when the transaction was suspended. The following code shows an example of suspending and resuming a transaction.

```
CosTransactions::Control_var control;

try  {
      current->begin();
      account->debit(amount);
      control =  current->suspend();
      //  do some nontransactional work
      ...
      current->resume(control);
      current->commit();
      }
catch(...)  {
      current->rollback();
      cout  <<  "An  exception  was  caught."  <<  endl;
}
```

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

The `resume()` operation can only be called on two occasions:

- Following a previous call to `suspend()`. Once `resume()` has been called, subsequent calls to `resume()` will fail, raising the `CosTransactions::InvalidControl` exception. This means that the `resume()` operation cannot be used to create several threads that participate in the same transaction because only one of the threads can successfully call `resume()`. The reason for this is that OrbixOTS implements "checked XA" behaviour that prevents transactions from being committed while there are outstanding threads running in the transaction. After a transaction has been resumed, you can make a new sequence of suspend and resume calls. Multiple threads in a transaction are permitted in OrbixOTS for C++, but you must use the `TranPthread` class described in "Threads and Transactions" on page 72.
- `resume()` can also be called following a call to `TransactionFactory::recreate()`. See "Explicit Propagation" on page 131 for details on using this operation.

Nested Transactions

This section provides an introduction to nested transaction and how they are created in both the indirect and direct approaches. It also describes some miscellaneous operations for getting information about transactions and testing relationships between transactions. The additional operations provided by the `Coordinator` and `Current` interfaces are also described.

OrbixOTS fully supports nested transactions. In the nested transaction model, the work done by a single transaction can be broken down into a series of sub-transactions. These sub-transactions can have their own sub-transactions and so on. The advantage of nested transactions is that the failure of a sub-transaction does not cause the whole transaction to fail. Thus the application can decide to repeat the work in another sub-transaction or to take alternative action.

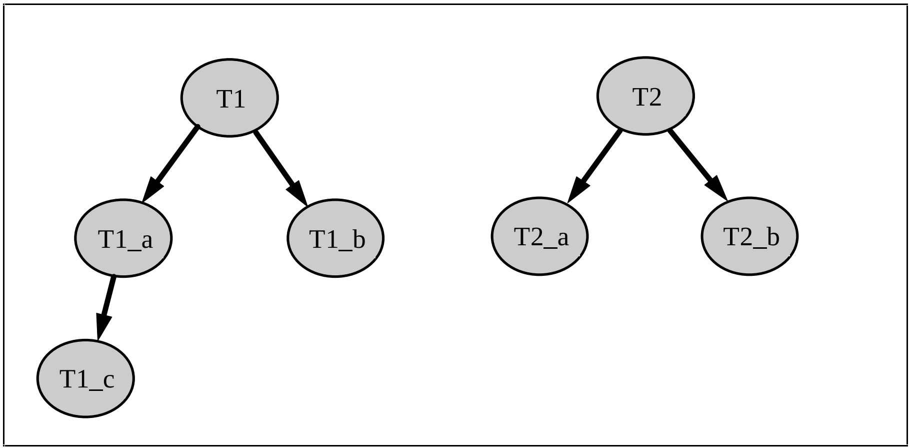


Figure 5.1: *Two Transaction Families*

The nested transaction model introduces the concept of transaction families. Two such families are shown in Figure 5.1. There are two top-level transactions T1 and T2. Transaction T1 has two child transactions T1_a and T1_b. Likewise, the child transactions of T2 are T2_a and T2_b. The sub-transaction T1_a also has a child transaction T1_c. Each child transaction has a single parent transaction. The parent of T1_c is T1_a and the parent of T2_b is T2.

The way in which sub-transactions are created depends on whether the indirect or direct approach is used. The indirect approach (using the Current interface) is the simplest. Here, sub-transactions are created by making nested calls to Current::begin(). For example, the following code first creates a top-level transaction (assuming there is no current transaction) and then creates a sub-transaction:

```
// Create top-level transaction.
current->begin();
…
// Create nested transaction.
current->begin();
…
```

```
if (…) {
    // Commit current nested transaction.
    current->commit(1);
} else {
    // Rollback current nested transaction.
    current->rollback();
}
// Commit top-level transaction.
current->commit(1);
```

This code also shows the sub-transactions being either committed or rolled back. If the sub-transaction is rolled back, all work done by the sub-transaction is undone. However, the top-level transaction continues and, when it commits all of its work (excluding the work done by the sub-transaction), is made permanent.

In the direct approach top-level transactions are created using a transaction factory. However, the `create_subtransaction()`operation is used to create 1subtransactions, supported by the `Coordinator` interface.

Example code for the direct approach is as follows:

```
CosTransactions::TransactionFactory_var factory =
   CosTransactions::TransactionFactory::
    _bind("TransactionFactory:SomeServer", "SomeHost");
…
// Create nested transaction.
CosTransactions::Control_var c2;
CosTransactions::Coordinator_var coord;
coord = c1->get_coordinator();
c2 = coord->create_subtransaction();
…
CosTransactions::Terminator_var t2;
t2 = c2->get_terminator();
if (…) {
    // Commit nested transaction.
    t2->commit(1);
} else {
    // Rollback nested transaction.
    t2->rollback();
}
// Commit top-level transaction.
CosTransactions::Terminator_var t1;
t1 = c1->get_terminator();
```

```
t1->commit(1);
```

Threads and Transactions

Some problems that an application must solve are best done using multiple threads to exploit the available concurrency. You can also break transactions into several sub-tasks that can be executed in parallel. There are two simple ways in which concurrency can be introduced to a transaction:

- Creating a top-level transaction that consists of several concurrent threads.
- Creating a top-level transaction that consists of several concurrent nested transactions (each of these nested transactions can in turn be composed of several concurrent nested transactions).

These concurrent transaction models are supported in OrbixOTS by the `TranPthread` class. This class allows you to start threads that can either join an existing transaction or run in an new top-level or nested transaction. The `TranPthread` class is declared as follows:

```
class TranPthread
{
   public:
      void
      Create(
         void* (*start_func)(void *),
         void* arg,
         int start_new_tran = 0
      );
      void
      Background(
         void* (*start_func)(void *),
         void* arg,
         int start_new_tran = 0
      );
      void*
      Join();
      };
```

A thread is created by invoking the `Create()` operation on an instance of this class. `start_func` is a pointer to a function that is the entry point for the new thread. This function takes a single parameter of type `void*` and returns a `void*`. The value passed to this function is the value of the second argument to the `Create()` operation, `arg`. The parameter `start_new_tran` indicates whether a new transaction is to be created: a zero value (default) means a new transaction is not created and the thread participates in the current transaction, if any; a non-zero value means the thread executes within a new transaction.

The `Join()` operation waits for the thread to exit and returns the return value of the thread's start function. You must use `Join()` when creating threads that participate in an existing transaction. `Join()`ensures that the threads have completed before the transaction can be committed.

The `Background()` operation is similar to `Create()` except that the threads created are detached and the `Join()` operation cannot be used. As it is not possible to determine when the thread has completed, do not use the `Background()` operation to create threads that participate in an existing transaction.

As an example of how to use the `TranPthread` class, the following code creates a transaction and then creates ten threads that participate in the transaction. Note that the `Join()` operation is used to wait for all threads to complete before the transaction is allowed to commit.

```
// Start function for threads.
void*
thread_main(
   void* arg
)
{
   // Do work on behalf of the current transaction.
   // ...
   CosTransactions::Current_var current = ...
   return 0;
}
void
main(
   int argc,
   char** argv
)
{
   // Application initialization
```

```
// ...
CosTransactions::Current_var current = ...
TranPthread thread[10];
// Create a transaction.
current->begin();
// Create 10 threads, all of which participate
// in the current transaction.
int i;
for (i = 0; i < 10; i++)
{
   thread[i].Create(thread_main, 0);
}
// Wait for threads to finish
for (i = 0; i < 10; i++)
{
   thread[i].Join();
}
// Commit the transaction.
current->commit(1);
}
```

Table 5.2 summarizes the effect of the `start_new_tran` parameter of the `Create()` operation on the new thread, depending on whether there is a current transaction or not.

Current Transaction ?	**start_new_tran parameter**	**Effect on New Thread**
No	0	Runs in no transaction.
No	1	Runs in new top-level transaction.
Yes	0	Runs in current transaction.
Yes	1	Runs in new nested transaction.

Table 5.2: *Effect of the* `start_new_tran` *Parameter*

Miscellaneous Operations

The `Coordinator` interface provides several useful operations relating to getting information about transactions and the relationships between transactions. Some of the operations are also supported by the `Current` interface.

Transaction Status

The operation `Coordinator::get_status()` returns the current status of a transaction.

The values returned by this operation and their meaning are shown in Table 5.3.

Status	Meaning
`StatusMarkedRollback`	The transaction has been marked to be rolled back.
`StatusRolledBack`	The transaction has completed rolling back.
`StatusActive`	The transaction is active. This is the case after the transaction has started and before the start of the commit protocol or before the transaction has rolled back.
`StatusNoTransaction`	There is no transaction.
`StatusRollingBack`	The transaction is in the process of being rolled back.
`StatusCommitted`	The transaction has completed its commit protocol.
`StatusPrepared`	The transaction has completed the first phase of its commit protocol.
`StatusUnknown`	The exact state of the transaction is unknown at this point.

Table 5.3: *Transaction Status Values*

Status	Meaning
StatusCommitting	The transaction is in the process of committing.
StatusPreparing	The transaction is in the process of the first phase of its commit protocol.

Table 5.3: *Transaction Status Values*

Below is example code showing how to obtain the status of a transaction:

```
CosTransactions::Coordinator_var coord = …
CosTransactions::Status status;
status = coord->get_status();
if (status == CosTransactions::StatusActive) {
    //
} else if (status == CosTransactions::StatusNoTransaction) {
    //
} else if
```

There are two additional status operations for use within transaction families. The `get_top_level_status()` operation returns the status of the top-level transaction in a transaction family. The status of a transaction's parent can be obtained by using the operation `get_parent_status()`.

The `get_status()` operation is also supported by the `Current` interface.

Transaction Relationship Operations

There are several operations that test the relationship between two transactions. Each of these operations takes as a parameter a reference for the coordinator of a transaction.

`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 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.
`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 a child transaction is a descendant of its parent.

For example, the following code tests if the transaction represented by the coordinator `c1` is an ancestor of the transaction represented by the coordinator object `c2`.

```
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
}
```

To illustrate these relationship operations, Table 5.4 shows the results of some relationship tests between the transactions shown.

Transactions	Same?	Related?	Ancestor?	Descendant?
T1 and T1	Yes	Yes	Yes	Yes
T1 and T2	No	No	No	No

Table 5.4: *Relationship between Transactions*

Transactions	Same?	Related?	Ancestor?	Descendant?
T1 and T1_a	No	Yes	Yes	No
T1 and T1_c	No	Yes	Yes	No
T1_a and T1	No	Yes	No	Yes
T1_a and T1_b	No	Yes	No	No
T1_c and T2_b	No	No	No	No

Table 5.4: *Relationship between Transactions*

Transaction Names

A string representation of a transaction is obtained from the operation `get_transaction_name()`. This can be used for debugging:

```
CosTransactions::Coordinator_var coord = …
CORBA::String_var name;
name = coord->get_transaction_name();
cout << "Transaction name is " << name << endl;
```

The `Current` interface also supports the `get_transaction_name()` operation.

Hash Functions

There are some situations where it is necessary to maintain data on a per transaction basis. The `is_same_transaction()` operation may be used to compare two transactions, but for efficiency the `Coordinator` interface provides two hash operations.

The `hash_transaction()` operation returns a hash code for the transaction represented by the target `Coordinator` object. `Coordinator` objects for the same transaction always return the same hash code. Hash codes are uniformly distributed over the range of a CORBA unsigned long type.

```
CosTransactions::Coordinator_var coord =
CORBA::Ulong hashCode;
hashCode = coord->hash_transaction();
```

Note that hash codes are not guaranteed to be unique. The `hash_transaction()` operation should be used in conjunction with the `is_same_transaction()` operation when mapping from a transaction to the transaction specific data.

The second hash operation is `hash_top_level_tran()`, which returns a hash code for the top-level transaction within a transaction family.

6

Writing a Recoverable Resource

OrbixOTS provides a set of interfaces that support the implementation of recoverable resources, as opposed to supporting integration of XA-compliant resource managers.

Introduction

This chapter describes how a recoverable resource can participate in the two-phase-commit (2PC) protocol of a transaction and provides guidelines for recovering from failure. Adding support for nested transactions, synchronisation, and heuristic outcomes is also covered.

Recoverable Objects

A recoverable resource may be many things, ranging from a simple object or set of objects to a large relational or object-oriented database. For illustration we assume here that a recoverable resource is a single object representing an account in a bank. However, the guidelines presented here are applicable to all recoverable resources, whatever their nature. The term recoverable object used in this text can refer to any recoverable resource. (Recoverable object is used instead of recoverable resource to prevent confusion with the resource object used to participate in the 2PC of a transaction.)

The IDL for our bank account might be as follows:

```
interface Account : CosTransactions::TransactionalObject {
   void lodge(in float amount);
   void withdraw(in float amount);
   float balance();
};
```

Here we are using the `TransactionalObject` interface to provide implicit propagation of transaction contexts. Explicit propagation can be used by including a reference to a `Control` object as a parameter in each operation.

Recoverable Servers

Similar to an OrbixOTS server that uses an XA resource manager, a server that supports recoverable objects requires a transaction log to record the progress of transactions. Additional work is required to support recovery after a failure.

Providing access to a transaction log is done by setting the attributes of the `OrbixOTS::Server` pseudo object. Thus a local log may be specified using the following code:

```
ots->logDevice("OTS.log");
ots->restartFile("r1");
ots->mirrorRestartFile("r2");
```

Alternatively, you can use the transaction log belonging to an existing OrbixOTS server registered under the name `TM`, by using the following code:

```
ots->logServer("TM");
```

Note, when using a log server in this way, you should not set the attributes `logDevice`, `restartFile`, or `mirrorRestartFile`. The log server must be running on the local host; using a log server on another host is not permitted.

The `OrbixOTS::Server` pseudo object provides the operation `recoverable()` to indicate that the server is recoverable and requires a transaction log. This operation is passed a reference to a sub-class of `OrbixOTS::Restart` which provides a call-back operation used during the recovery phase.

The following code declares a restart class called `Restart_i` and redefines the `recovery()` call-back operation:

```
class Restart_i : public virtual OrbixOTS::Restart {
public:
    // Implementation specific members omitted.
    // Redefinition of the recovery call-back operation.
    void recovery();
};
```

Now the server can be registered as recoverable using an instance of `Restart_i`:

```
OrbixOTS::Restart_var restart = new Restart_i();
ots->recoverable(restart);
```

Later when the `OrbixOTS::Server::init()` operation is invoked, recovery processing is initiated and the `Restart_i::recovery()` operation will be called. Notice that this means recovery processing is done before `OrbixOTS::Server::impl_is_ready()` is called and therefore before the server can process invocations.

The Data Log

The transaction log maintained by OrbixOTS records and stores the progress of transactions. To implement a recoverable resource, some mechanism is also required to store information on modifications made to the resources so that these modifications may be reapplied after a failure. Thus a server supporting recoverable resources requires a stable data log that is logically separate from the transaction log.

Note that the term data log is used here for clarity and does not preclude an implementation using any other suitable stable storage mechanism.

Resource Objects

Support for recoverable objects is provided primarily by the interface CosTransactions::Resource. This interface provides a means for a recoverable object to participate in a transaction's 2PC protocol.

The Resource interface is defined as follows:

```
// In module CosTransactions.
interface Resource {
    Vote prepare()
           raises (HeuristicMixed,
                   HeuristicHazard);

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

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

    void commit_one_phase()
           raises (HeuristicHazard);

    void forget();
};
```

The prepare() operation allows the resource to vote in the outcome of the transaction and to prepare for an eventual commit. The commit() and rollback() operations are called when the transaction is committed or rolled-back. A guide to implementing these operations is given in section, "Participating in the 2PC Protocol" on page 87.

Resource objects become participants in a transaction by registering with that transaction. To illustrate this, assume our resource object is implemented by the class Resource_i and is declared as follows (using the BOAImpl approach):

```
class Resource_i : public virtual CosTransactions::ResourceBOAImpl
{
```

```
public:
  // Resource_i specific members omitted.

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

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

  void commit(CORBA::Environment&)
    throw (CosTransactions::NotPrepared,
           CosTransactions::HeuristicRollback,
         CosTransactions::HeuristicMixed,
           CosTransactions::HeuristicHazard);

  void commit_one_phase(CORBA::Environment&)
    throw (CosTransactions::HeuristicHazard);

  void forget(CORBA::Environment&);
};
```

To register an instance of this class, the `Coordinator::register_resource()` operation is invoked, passing the resource object's reference as the parameter. The following code illustrates resource registration by showing part of the implementation of the `deposit()` operation on our `account` interface implemented by the class `Account_i`:

```
void Account_i::deposit(const float amount, CORBA::Environment&)
{
  if (/* transaction not already involved */ ) {
    CosTransactions::Resource_var resource;
    CosTransactions::Control_var control;
    CosTransactions::Coordinator_var coord;
    CosTransactions::RecoveryCoordinator_ptr recCoord;

      // Get a reference to the coordinator for the
      // current transaction
      // (current is a reference to the Current pseudo
      // object).
         resource = new Resource_i();
```

```
            control = current->get_control();
            coord = control->get_coordinator();
         // Register the resource.
            recCoord = coord->register_resource(resource);
    }
     // Perform deposit ...
}
```

The register_resource() operation returns a reference to a recovery coordinator (specified in the CosTransactions::RecoveryCoordinator interface). This has a single operation, replay_completion(), which is used in certain failure situations and is discussed in "Failure and Recovery" on page 91.

A resource object may only be registered once so a test is required to determine whether the current transaction has already accessed the recoverable object. To support this, the CosTransactions::Coordinator interface provides two operations: is_same_transaction() and hash_transaction(). The is_same_transaction() operation takes a coordinator object and returns true if both coordinators represent the same transaction. The hash_transaction() operation returns a uniformly distributed hash code for the transaction to help reduce the number of times is_same_transaction() needs to be called.

During recovery after a server failure, resource objects registered with incomplete transactions need to be recreated so the 2PC protocol can complete. OrbixOTS uses the resource object's marker to associate it with a particular transaction and during recovery the same marker must be used when the resource object is recreated. This requires that the markers used for resource objects be unique across server failures. One approach is to use the string returned by the operation get_transaction_name() (provided by the Current and Coordinator interfaces) in addition to some per-recoverable object unique identifier (such as an account number in our example).

At any given time, a recoverable object may be associated with multiple resource objects–one for each transactions currently accessing the resource. They are managed by OrbixOTS and are deleted when they are no longer required.

Participating in the 2PC Protocol

Once a resource object has been registered with a transaction, it will participate in the 2PC protocol of the transaction. This means it must implement each of the operations in the `Resource` interface. Below is a description of the requirements for these operations and guidelines for a typical implementation.

Note: When OrbixOTS invokes these operations, the current transaction is not available via the `Current` pseudo object. This is because the 2PC protocol operations are normally invoked from a thread not associated with the transaction. If an operation requires access to information about the transaction, the resource object or recoverable object must maintain a reference to the transaction's `Control` or `Coordinator` object.

The prepare() Operation

This operation allows the resource object to vote in the 2PC protocol of the transaction and to prepare the recoverable object for eventual commitment. It is called at most once by OrbixOTS.

Voting is done by returning one of the three values `VoteReadOnly`, `VoteRollback`, or `VoteCommit` which are enumerated in the `CosTransactions::Vote` type:

`VoteReadOnly`

This indicates that the resource object does not want to be further involved in the 2PC protocol. After returning `VoteReadOnly`, the resource object can forget about the transaction.

A typical use of this vote is when the recoverable object associated with the resource object was not modified during the transaction. For example, if `balance()` is the only operation invoked on an account object then the resource might return `VoteReadOnly`.

`VoteRollback`	This indicates that the resource object has decided to rollback the transaction. If one or more resource objects return this value, the transaction will always be rolled-back. After returning `VoteRollback` the resource object will not be further involved in the 2PC protocol and can forget about the transaction.
`VoteCommit`	This indicates that the resource object is prepared to commit its part of the transaction. This does not guarantee that the transaction will eventually commit as it may be rolled-back due to factors such as another resource voting to roll-back or the failure of some other component. A resource object returning `VoteCommit` has a responsibility to ensure that a subsequent invocation on the `commit()` operation will succeed even after the failure of a server.

Typically, if the resource object returns `VoteCommit`, information must be stored in the data log so that after a server failure, the resource object can fulfil its obligations as a participant in the transaction. The actual information stored depends on the recoverable object, but the following is a general guide to what is required:

- The name of the transaction associated with the resource object. This can be obtained using the `get_transaction_name()` operation provided by the `Coordinator` object.
- The marker for the resource.
- The string form of the reference for the recovery coordinator returned when the resource object was registered (obtained using `CORBA::ORB::object_to_string()`).
- Sufficient information to redo any modifications made to the resource object during the transaction. This might be a complete copy of the data, a copy of the modified parts, or a list of the operations that caused the modifications. In the bank account example, we could store the new balance of the account or the fact that the operations `deposit(120)` and `withdraw(50)` were invoked.

Once a resource has been prepared and has returned `VoteCommit`, it may invoke the operation `replay_completion()` on the recovery coordinator object as a hint that the 2PC protocol has not been completed. This is necessary during recovery after a server failure and in other failure scenarios. See section, “Failure and Recovery” on page 91 for details.

The `prepare()` operation may raise one of the exceptions `HeuristicMixed` or `HeuristicHazard`. Most resource objects have no need to raise these exceptions. See section, “Heuristic Outcomes” on page 98 for more about heuristic exceptions.

The rollback() Operation

When a transaction is rolled-back, this operation is invoked on all resource objects registered with the transaction that were not prepared or were prepared and returned `VoteCommit`. A resource object should expect `rollback()` to be invoked multiple times, including after a server failure. If the resource object has forgotten about the transaction, no action is required.

Typically an implementation of `rollback()` does the following:

- Undoes any changes made to the recoverable object associated with the resource. This requires that the resource or recoverable object has some mechanism of undoing modifications; for example, by creating a backup the first time the recoverable object is modified by the transaction.
- Writes an entry to the data log indicating that the transaction has been rolled back. (The name of the transaction or the marker of the resource object may be used for identification purposes.)
- Cleans up all traces of the transaction from the resource object and the recoverable object.

The `rollback()` operation may raise one of the exceptions `HeuristicCommit`, `HeuristicMixed`, or `HeuristicHazard`. The former is raised when the resource actually wants to commit the transaction; the latter two are not normally required for most resource objects.

The commit() Operation

After the prepare phase of the 2PC protocol, the transaction is committed if all resources registered with the transactions returned either `VoteCommit` or `VoteReadOnly`, and no external factors caused the transaction to be rolled-back. During the commit phase, the `commit()` operation is invoked on resources that returned `VoteCommit`. A resource object should expect `commit()` to be invoked multiple times, including after a server failure. If the resource object has forgotten about the transaction, no action is required.

Typically an implementation of `commit()` does the following:

- Makes permanent any modifications made to the recoverable object associated with the resource.
- Writes an entry to the log indicating that the transaction has been committed. (The name of the transaction or the marker of the resource object may be used for identification purposes.)
- Cleans up all traces of the transaction from the resource object and the recoverable object.

If the resource object was not prepared (that is, the `prepare()` operation was not invoked), then the exception `NotPrepared` should be raised. In addition, the `commit()` operation may raise one of the exceptions `HeuristicRollback`, `HeuristicMixed`, or `HeuristicHazard`. The former is raised when the resource actually wants to rollback the transaction; the latter two are not normally required for most resources.

The commit_one_phase() Operation

The OTS specification optionally allows an implementation to invoke this operation if there is only one resource object registered with the transaction. There is no prepare phase. Currently OrbixOTS does not implement this option, so `commit_one_phase()` is never invoked. However, this option may be provided in a future release so an implementation should be provided. A resource object should expect `commit_one_phase()` to be invoked multiple times, including after a server failure. If the resource has forgotten about the transaction, no action is required.

Typically an implementation of `commit_one_phase()` does the following:

- Makes permanent any modification made to the recoverable object associated with the resource.
- Cleans up all traces of the transaction from the resource object and the recoverable object.

If the resource object cannot commit the modifications, the standard system exception `TRANSACTION_ROLLEDBACK` should be raised. In addition the `commit_one_phase()` operation may raise the exception `HeuristicHazard`.

The forget() Operation

If a resource object raises a heuristic exception, it must remember the exception raised so that subsequent calls to `commit()`, `commit_one_phase()`, and `rollback()` return a consistent outcome. This must also survive server failures. The information should therefore be recorded in the data log. The `forget()` operation is invoked when knowledge of heuristic exceptions is no longer required.

A typical implementation of `forget()` cleans up all traces of the transaction from the resource object and the recoverable object.

Failure and Recovery

A participant in a transaction must be able to recover from failures. Once a resource object returns `VoteCommit` from its `prepare()` operation, it has an obligation to see that the 2PC protocol is completed. The mechanism to do this is provided by the `replay_completion()` operation of the interface `RecoveryCoordinator`.

Recall that when a resource object is registered with a transaction (see "Resource Objects" on page 84), the `register_resource()` operation returns a reference to a recovery coordinator for that resource. Any resource that has been prepared, should invoke this operation if the 2PC protocol has not been completed. (For example, the `commit()` or `rollback()` operations have not been called.)

Note: The `replay_completion()` operation is non-blocking and does not force the coordinator to complete the transaction. It is only treated as a hint.

Remote Server Failure

Assume that the resource is registered with a coordinator in a remote server and the decision has been made to rollback the transaction but this has not propagated to the local server. If the remote server fails, there will be no record of the transaction after a restart. The OTS uses presumed rollback semantics, so the `rollback()` operation will not be called. This is an optimisation that allows a coordinator not to log anything before the commit decision. If there is no record of the transaction at restart, the transaction is presumed to have been rolled-back.

By invoking the `replay_completion()` operation, the resource object can determine the correct outcome in this case. This also needs to be done after the failure of the local server.

Local Server Failure

If a recoverable server fails, it is necessary to perform recovery for those resources whose associated transaction has an unknown outcome. This typically occurs when the server crashes after the resource objects have been prepared, but before the commit or rollback decision has been propagated to all resource objects.

Recall that when making a server recoverable ("Recoverable Servers" on page 82), the `OrbixOTS::Server::recoverable()` operation is called and passed the reference to a restart object. When a recoverable server is restarted, OrbixOTS processes the transaction log, determines those transactions that require completion, and recreates the recovery coordinators for those transactions. When this is complete, the `recovery()` operation is invoked on the `restart` object.

When the `recovery()` operation returns, OrbixOTS expects that all resource objects for incomplete transactions are recreated and in a state to receive invocations on their `commit()` or `rollback()` operations. If you follow the guidelines presented in the section, "Participating in the 2PC Protocol" on page 87, then this can be done by processing the data log and finding those resource objects for which there is a prepare record but no commit or rollback record. Then for each resource object the following is done:

- The resource object is recreated using its original maker. The resource object's marker is obtained from the data log.

- The recoverable object associated with the resource object is brought back to the state it was in during the prepare phase of the transaction. In our bank account example, this can be done by reading the current state of the account and applying the modification information stored in the data log.
- Rebind to the resource object's recovery coordinator. The reference for the recovery coordinator is obtained from the data log. If during the rebind the recovery coordinator does not exist, then it can be presumed that the transaction has been rolled-back.

The following is outline code for binding to the recovery coordinator and completing the transaction:

```
try {
  // Rebind to the object reference (ref is the stringified
  // reference).
  CORBA::ORB_var orb = ...
  CORBA::Object_var obj =
    orb->string_to_object(ref) ;
  // Narrow the object reference.
  CosTransactions::RecoveryCoordinator_ptr recCoord =
    CosTransactions::RecoveryCoordinator::_narrow(obj);
  // Restart the completion of the 2PC protocol.
  recCoord->replay_completion(resource);
} catch (CORBA::OBJECT_NOT_EXIST ex) {
  // No recovery coordinator so assume the transaction has
  //rolled-back.
  resource->rollback();
} catch (CosTransactions::NotPrepared ex) {
  // Resource was not prepared so rollback.
  resource->rollback();
} catch (...) {
  // ...
}
```

Nested Transactions

OrbixOTS fully supports nested transactions (also known as sub-transactions) that provide a means of isolating failure. A transaction of several nested transactions that can independently fail without causing the whole transaction to be rolled-back can achieve this. A nested transaction itself can be composed of several nested transactions. The effects of a nested transaction are only made durable if all ancestor transactions (including the top-level transaction) commit.

The `CosTransactions::Resource` object provides a means for a recoverable object to participate in the 2PC protocol of a top-level transaction. Because nested transactions do not require a 2PC protocol (that is, there is no prepare phase), an alternative interface is required. A specialization of the `Resource` interface called `CosTransactions::SubtransactionAwareResource` is used. The interface is declared as follows:

```
// IDL (in module CosTransactions)
interface SubtransactionAwareResource : Resource
{
   void
   commit_subtransaction(
      in Coordinator parent
   );
   void
   rollback_subtransaction();
};
```

You can register an object supporting the `SubtransactionAwareResource` interface with a nested transaction using either the `register_subtran_aware()` operation or the `register_resource()` operation. As with objects supporting the `Resource` interface, the object can only be registered with a single transaction.

The commit_subtransaction() Operation

The `commit_subtransaction()` operation is called when the nested transaction is committed. `commit_subtransaction()` accepts a reference to the coordinator object of the parent transaction as a parameter. The resource object usually just cleans up all traces of the transaction from the resource and recoverable objects. Any modifications made by the nested transaction are not made durable at this state. However, modifications should be made available to the parent transaction so they can be made durable when the top-level transaction commits.

The rollback_subtransaction() Operation

This operation is invoked when the nested transaction rolls-back. In this case, any modifications made by the nested transaction should be undone.

Registering SubtransactionAwareResource Objects

You can invoke the `register_subtran_aware()` operation on a nested transaction's coordinator object to register an object supporting the `SubtransactionAwareResource` interface with the transaction. For example, assuming the class `SubTranResourceImpl` supports the `SubtransactionAwareResource` interface, the following C++ code registers an instance of the object as a participant in the current transaction:

```
// C++
try
{
   CosTransactions::Current_var current = …
   CosTransactions::Control_var control;
   CosTransactions::Coordinator_var coord;

   control = current->get_control();
   coord = control->get_coordinator();

   CosTransactions::SubtransactionAwareResource_var resource;
   resource = new SubTranResourceImpl();
   // Register the resource object to be notified when the
   // sub-transaction completes.
   coord->register_subtran_aware(resource);
}
catch (CosTransactions::NotSubtransaction)
{
   // Current transaction is not a nested transaction.
   // …
}
catch (CosTransactions::Inactive)
{
   // Current transaction is inactive.
   // …
}
catch (…)
```

```
{
   // …
}
```

If the transaction is not a nested transaction, `register_subtran_aware()` raises the `CosTransactions::NotSubtransaction` exception. Also, if the transaction is not currently active, the exception `CosTransactions::Inactive` is raised.

Since the `SubtransactionAwareResource` interface is a specialization of the `Resource` interface, the operation `register_resource()` can also be used. In this case, the object is notified when the nested transaction completes *and* when the top-level transaction commits. For example:

```
// C++
// Register the resource object to be notified when the
// sub-transaction completes and when the top-level enclosing
// transaction completes.
CosTransactions::RecoveryCoordinator_var recCoord;
recCoord = register_resource(resource);
```

When using the `register_resource()` operation, the operations specific to the `Resource` interface (for example `prepare()`, `commit()` and `rollback()`) are only called when the top-level enclosing transaction completes. If a nested transaction commits, the effects of the transaction may subsequently be undone if the top-level transaction rolls back.

Concurrency

If your server permits concurrent or interleaved transactions (that is, one of the serialization modes `concurrent` or `serializeRequests` is used), then some form of synchronization to the recoverable object is required so that the isolation property can be ensured. OrbixOTS provides an implementation of the OMG Object Concurrency Control Service (OCCS) which may be used for synchronization. This section discusses the requirements for using OCCS with a recoverable object.

Using the OCCS puts extra requirements on both the implementation of recoverable objects and resource objects.

A full description of the OCCS is given in Chapter 7 "Concurrency Control" on page 103.

Requirements for Recoverable Objects

The number of lock-sets required by a recoverable object and the modes in which they are acquired will vary depending on the nature of the resource. For clarity, we just consider our bank account object which requires a single lock-set and standard read/write locking.

During initialization of the recoverable object the lock-set is created. This is done using a `LockSetFactory` object whihc is obtained by the `get_lockset_factory()` operation provided by the `OrbixOTS::Server` class. This is illustrated with the following code (assuming `m_lock` is a `LockSet` reference):

```
OrbixOTS::Server_var ots = ...
CosConcurrencyControl::LockSetFactory_var factory =
   ots->get_lockset_factory();

m_Lock = factory->create ();
```

Then at the start of each operation a lock must be acquired in the appropriate mode. For example, the `balance()` operation must acquire a read lock:

```
float Account_i::balance(CORBA::Environment&)
{
    // Acquire read lock
    m_Lock->lock(CosConcurrencyControl::read);

    // Register a resource object (if first access by a
    // transaction).

    float bal = ...
    return bal;
}
```

Note that the lock is not released when the operation returns. This is necessary to ensure other transactions do not see any intermediate results before the transaction completes.

Note: Using lock-sets in the implicit mode only provides synchronization for transactions. If your recoverable object permits concurrency within transactions, then additional synchronization is required.

Requirements for Resource Objects

Any locks held on a recoverable object need to be dropped when the transaction completes. This is done by invoking the `drop_locks()` operation on the lock coordinator. The lock coordinator is obtained by invoking the `get_coordinator()` operation on the lock set object. This is illustrated with the following code:

```
CosTransactions::Coordinator_var coord =
CosConcurrencyControl::LockCoordinator_var lockCoord;

// Get the lock coordinator from the lock set object.
lockCoord = m_Lock->get_coordinator(coord);

// Drop all locks.
lockCoord->drop_locks();
```

Locks should be dropped when a transaction commits and when a transaction rolls back.

Heuristic Outcomes

Heuristic outcomes arise when a resource unilaterally decides to commit or rollback its part of the transaction, possibly conflicting with the eventual outcome decided by the transaction's coordinator. For example, after a failure a resource may make a heuristic decision after a timeout period to free up access to resources. Heuristic outcomes are reported by raising one of the heuristic exceptions, which will be reported to the transaction's originator. (Provided the report-heuristics parameter passed to the `commit()` operation is true.)

There are four heuristic exceptions:

`HeuristicRollback`	This may be raised in the `commit()` operation to indicate that all updates to the recoverable object have been rolled-back.
`HeuristicCommit`	This may be raised in the `rollback()` operation to indicate that all updates to the recoverable object have been committed.
`HeuristicMixed`	This indicates that some updates have been committed while others have been rolled-back.
`HeuristicHazard`	This indicates that a heuristic decision has been made but it is not known which updates have been committed or rolled-back.

A resource object that makes a heuristic decision is obligated to remember the decision so that subsequent calls to either `commit()` or `rollback()` have consistent results. The decision must survive server failures so it must be stored in the data log. The `forget()` operation is called when the resource object no longer needs to remember the heuristic decision.

Heuristic outcomes may also arise when a transaction is forced to either commit or rollback by an administrator using the `otsadmin` tool.

Resource Object Lifecycle

This section describes the possible invocation sequences that OTS can make on a resource object in order to summarize the responsibilities of a resource object. At the end of each invocation sequence, the resource object is no longer involved in the transaction and system resources used by the resource object can be released.

Operations marked with a **+** can be invoked one or more times. This can happen, for example, if a failure occurs before the OTS has received the response to the invocation. For example, in invocation sequence **3,** the OTS invokes the `commit()` operation again if it does not receive the response to the first `commit()` operation (for example, due to a communications failure or an application crash). This continues until the OTS retrieves a valid response.

1. rollback()

This occurs when the transaction is rolled back before the resource object participates in the transaction's commit protocol. This can happen, for example, if the transaction is explicitly rolled-back, the transaction times-out, or another resource object voted to rollback the transaction.

2. prepare() → VoteReadOnly | VoteCommit

In this case the resource object returns either `VoteReadOnly` or `VoteCommit` in response to OTS invoking `prepare()`. These return values mean that the resource is no longer involved in the transaction.

3. prepare() →VoteCommit, commit() +

The resource object returns `VoteCommit` from the `prepare()` operation indicating that the resource has taken the necessary steps to eventually commit its part of the transaction. The OTS coordinator has collected the votes from all other resources and made the decision to commit the transaction.

4. prepare() → VoteCommit, commit() + → raise heuristics, forget() +

This sequence is the same as sequence **3** except that the resource, before receiving the `commit()` invocation, decides to rollback the transaction. The OTS coordinator decides to commit the transaction, and when the OTS eventually invokes the `commit()` operation, the resource responds by raising one of the heuristic exceptions `HeuristicRollback`, `HeuristicMixed`, or `HeuristicHazard`. Finally the OTS invokes the `forget()` operation indicating that the resource object is no longer involved in the transaction.

5. prepare() → VoteCommit, rollback() +

The resource object returns `VoteCommit` from the `prepare()` operation, but for some reason the OTS coordinator has decided to rollback the transaction. This can occur, for example, if another resource object returned `VoteRollback`, or because of some other failure.

6. prepare() → VoteCommit, rollback() + → raise heuristics, forget() +

This sequence is the same as sequence **5** except that the resource, before receiving the `rollback()` invocation, decides to commit the transaction. The OTS coordinator decides to rollback the transaction, and when the OTS eventually invokes the `rollback()` operation, the resource object responds by raising one of the heuristic exceptions `HeuristicCommit`, `HeuristicMixed`, or `HeuristicHazard`. Finally the OTS invokes the `forget()`operation indicating that the resource object is no longer involved in the transaction.

7. commit_one_phase() +

The resource object is the only resource registered with the transaction and the OTS coordinator has decided to use the one-phase-commit (1PC) protocol.

8. commit_one_phase() + → raise heuristics, forget() +

This sequence is the same as sequence **7** except the resource object raises the heuristic exception `HeuristicHazard`. The OTS then invokes the `forget()` operation indicating that the resource object is no longer involved in the transaction.

9. commit() → raises NotPrepared

This sequence indicates an OTS commit protocol error. A foreign OTS coordinator has invoked the `commit()` operation before the `prepare()` operation. The resource object responds by raising the `NotPrepared` exception.

10. rollback() + heuristics

This is a resource protocol error. The `rollback()` operation is invoked and the resource raises a heuristic exception. Heuristic exceptions can be raised by the `rollback()` operation only if the resource object was previously prepared.

11. prepare() → raise heuristics

In this case the `prepare()` operation raises one of the heuristic exceptions `HeuristicMixed` or `HeuristicHazard`. Normally a resource object never needs to raise heuristic exceptions from the `prepare()` operation. This situation is provided in the OTS specifications for an implementation technique called *interposition*.

Interposition allows a distributed transaction to be represented as a tree of transactions with one superior transaction (the root of the tree) and several subordinate transactions. Each subordinate transaction registers a resource object with its parent. Interposition allows the 2PC protocol to be spread over a number of servers rather than being the sole responsibility of a single server, and so prevents a single OTS server from becoming a bottleneck in the system.

An interposed resource object must be able to raise heuristic exceptions in its `prepare()` operation, because one of its subordinate resource objects can raise a heuristic exception during the `rollback()` operation. OrbixOTS uses the interposition technique for handling foreign OTS transactions; for native transactions a similar but more efficient technique is used. Interposition also explains why the `commit()` and `rollback()` operations can raise the `HeuristicMixed` and `HeuristicHazard` exceptions.

7

Concurrency Control

This chapter describes the Object Concurrency Control Service (OCCS) that is provided with OrbixOTS to control access to shared resources by concurrent transactions. Examples of how to use the OCCS C++ mapping are included.

OrbixOTS includes an implementation of the OMG Object Concurrency Control Service (OCCS). This can be used to control concurrent transactions as they access a shared set of resources. Though the OCCS is a separate service, it is tightly integrated with the transaction service. The following sections describe the OCCS and demonstrate how it is used.

Note that XA resource managers provide their own concurrency control and the OCCS is typically not required. The OCCS is useful when using the `Resource` interface to implement recoverable resources.

Locks and Lock Sets

The OCCS uses locks to control concurrent transactions. Before a transaction can access a shared resource a lock must be acquired on behalf of the transaction. Several lock modes are supported to increase the level of concurrency. If a transaction tries to acquire a lock in a mode that conflicts with a lock held by another transaction, the request is either denied or blocked until the conflict is resolved.

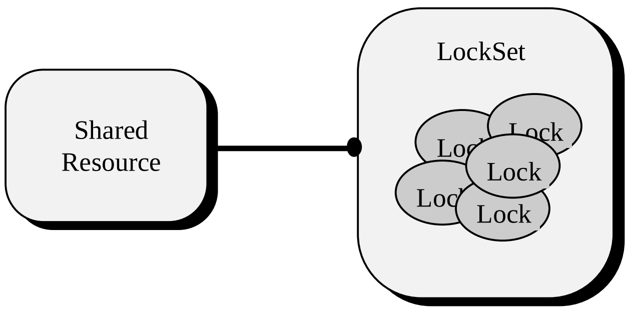


Figure 7.1: *Associating Lock Sets and Resources*

A *lock set* is a collection of locks that is associated with a resource, as shown in Figure 7.1. This association is made by the application and reflects the granularity of resources. For example, a resource could be a single object or a collection of objects. The former permits more concurrency but requires more locks, while the latter has fewer locks but greatly reduces the degree of concurrency.

Implicit and Explicit Lock Sets

Similar to the way in which transaction contexts can be propagated from a client to a server implicitly or explicitly, the OCCS provides *implicit* and *explicit* lock sets. With *implicit* lock sets, all operations are performed on behalf of the current transaction. With *explicit* lock sets, the identifier of the transaction, in the form of a reference to a coordinator object, is passed as a parameter to the operations.

The OCCS also allows implicit lock sets to be used outside of a transaction. Here, the requests are made on behalf of the current thread of control.

Lock Modes

The OCCS supports five different lock modes: read, write, upgrade, intention-read, and intention-write. Table 7.2 shows the conflict matrix for each mode (where ● indicates a conflict). A conflict occurs when a transaction requests a lock and at least one unrelated transaction holds a lock in a conflicting mode. Requests to acquire a lock that result in a conflict will either fail or cause the request to block.

	Requested Mode				
Granted Mode	**IR**	**R**	**U**	**IW**	**W**
Intention Read (IR)					●
Read (R)				●	●
Upgrade (U)			●	●	●
Intention Write (IW)		●	●		●
Write (W)	●	●	●	●	●

Table 7.2: *Lock Mode Conflict Matrix*

Standard multiple-readers/single-writer transactions are supported with read and write locks. The upgrade lock is used to overcome a common deadlock scenario. Intention read and write locks are used to support locking hierarchies of resources. These lock modes are discussed in more detail in the sections that follow.

Read/Write Locking

The OCCS supports conventional read/write locking which allows multiple readers but only a single writer. Transactions that want to read a resource must acquire a read lock, which will succeed only if there are no other transactions holding a write lock on the resource. Transactions that want to update a resource must acquire a write lock, which will succeed only if there are no other transactions holding either a read or a write lock on the resource.

Standard read/write locking can easily lead to deadlock when two or more transactions attempt to first read a resource and then later update the same resource. This is illustrated below where two transactions `T1` and `T2` acquire read and write locks on a resource `x`.

T1	T2
x.lock(R)	
	x.lock(R)
x.lock(W)	
BLOCKS!	
	x.lock(W)
	BLOCKS!

Due to the order in which each transaction acquires the locks and the order in which the transactions are interleaved, a deadlock situation arises. Each transaction is attempting to acquire a write lock which is conflicting with the read lock held by the other transaction.

	Requested Mode		
Granted Mode	**R**	**U**	**W**
Read (R)			●
Upgrade (U)		●	●
Write (W)	●	●	●

Table 7.3: *Read/Write/Upgrade Conflict Matrix*

To overcome this problem the OCCS supports upgrade locks. An upgrade lock is similar to a read lock except that it conflicts with itself. Table 7.3 shows the conflict matrix for read, write and upgrade locks. The resulting scenario is illustrated as follows:

T1	T2
x.lock(U)	
	x.lock(U)
	BLOCKS!
x.lock(W)	
release locks	**UNBLOCKS**
	x.lock(W)

Here, each transaction acquires an upgrade lock in anticipation that it will eventually want to acquire a write lock. Since an upgrade lock conflicts with itself, the transaction `T2` is blocked trying to acquire the upgrade lock and `T1` proceeds to acquire a write lock. When `T1` releases its locks, `T2` is granted the upgrade lock and can then acquire the write lock. Note that an upgrade lock does not prevent other transactions from acquiring read locks and reading the resource.

Hierarchical Locking

Many resources are hierarchical in nature. Consider the directory/file hierarchy in file systems and the database/table/row hierarchy in relational databases. The hierarchical nature of these resources may be exploited to reduce the number of locks that must be acquired for certain operations. To simplify the discussion consider the two-level hierarchy shown in Figure 7.4 on page 108, where there is a parent node **P** with 100 child nodes **C1**…**C100**.

Consider the following four transactions that want to perform certain operations:

`T1`: Update **C1**

`T2`: Update **C2**

`T3`: Read **C3**

`T4`: Read all children (**C1**…**C100**)

Using conventional locking, the first three transactions would acquire a read or write lock on the child node being accessed. Transaction T4 would have to acquire a read lock on the parent node and a read lock on each of the child nodes. In this example, T4 would acquire 101 locks but in a real database there might be thousands of records that need to be locked.

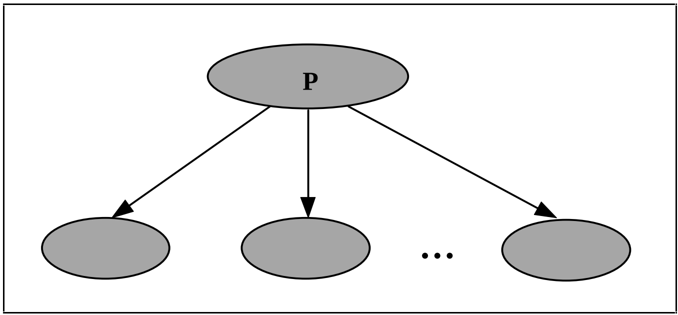


Figure 7.4: *Hierarchical resources*

A better solution is to have multiple granularity locks so that a read lock could be acquired on all child nodes. Here, T4 could just acquire a read lock on the parent node **P**. However, this still allows T1 and T2 write access to the child nodes **C1** and **C2**, so these transactions would have to acquire a write lock on **P**. This naïve solution severely restricts concurrency, since locks are effectively held at the highest level. In a database this would mean acquiring read and write locks on the database itself!

The correct solution is to use intention locks, which provide variable granularity locks suitable for hierarchical resources. There are two types of intention locks: intention-read and intention-write locks.

Table 7.5 shows the conflict matrix for read, write and intention locks without upgrade locks.

	Requested Mode			
Granted Mode	**IR**	**R**	**IW**	**W**
Intention Read (IR)				●
Read (R)			●	●
Intention Write (IW)		●		●
Write (W)	●	●	●	●

Table 7.5: *Intention Lock Conflict Matrix*

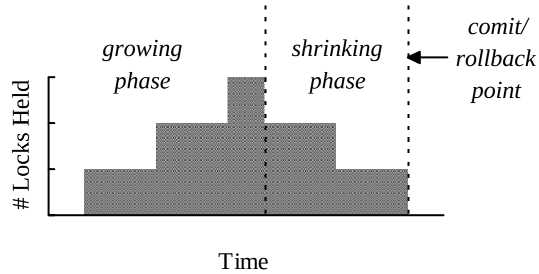
When using intention locks to access a hierarchy, the order in which locks are acquired is always from the top down, as shown in Figure 7.6. Transaction T1 first acquires an intention-write lock in the parent node **P** and then acquires a write lock on the child node **C1**. Similarly, T2 acquires an intention-write lock on **P** and a write lock on **C2**. Both transactions are granted access since they are working on different child nodes and intention-write locks do not conflict. Transaction T3 acquires an intention-read lock on **P** and a read lock on **C3**. Again there is no conflict, since all three transaction are accessing different child nodes and intention-read locks do not conflict with intention-write locks. Finally, T4 attempts to acquire a read lock on **P**, which is equivalent to acquiring read locks on all child nodes. This causes a conflict because a read lock conflicts with intention-write locks. When transactions T1 and T2 complete and drop their locks, T4 will be granted the read lock.

T1	T2	T3	T4
P.lock(IW)			
C1.lock(W)			
	P.lock(IW)		
	C2.lock(W)		
		P.lock(IR)	
		C3.lock(R)	
			P.lock(R)
			BLOCKS!

Figure 7.6: *Hierarchical Locking using Intention Locks*

Two-Phase Locking

When several transaction are run concurrently, the effect must be the same as running the transactions in some serial order. This is known as the *serializability* property. When using the OCCS (or any other concurrency control mechanism that uses locks) there is a simple technique that must be followed to ensure serializability, known as two-phase-locking (2PL).

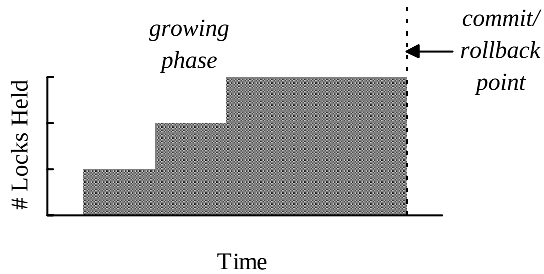


(a) Standard-2PL

Figure 7.7: *Standard Locking*

Figure 7.7 shows how 2PL works. There are two phases: the growing phase and the shrinking phase. All locks are acquired during the growing phase and no locks may be released. As soon as one lock is released the shrinking phase starts. In this second phase, locks can only be released and no new locks can be acquired.

A simpler variation on standard-2PL is strict-2PL which is shown in Figure 7.8. Here all locks are released when the transaction commits (or rolls-back) and no locks are released during the transaction. This is supported in the OCCS with a lock coordinator object that can release all locks held by a transaction. Strict-2PL decreases the level of concurrency between transactions, because locks are held for longer times.



(b) Strict-2PL

Figure 7.8: *Strict Two Phase Locking*

Note that using standard-2PL can weaken the isolation property. Consider a transaction that acquires a write lock on a resource, modifies the resource and then releases the write lock. Another transaction can then read the modified resource and view the intermediate results of an incomplete transaction. For this reason, strict-2PL should be used in preference to standard-2PL unless your application can tolerate the weaker isolation levels.

Multiple Possession Semantics

The OCCS locking model provides multiple possession semantics. This means that a transaction may hold multiple locks in a lock set at any one time. In addition, a transaction may hold several locks in the same mode. Effectively a count is maintained per lock mode for each transaction holding locks in a lock set.

	Lock Set									
	Transaction T1					Transaction T2				
Operation	IR	R	U	IW	W	IR	R	U	IW	W
	-	-	-	-	-	-	-	-	-	-
T1: lock(IR)	1	-	-	-	-	-	-	-	-	-
T1: lock(W)	1	-	-	-	1	-	-	-	-	-
T1: release(W)	1	-	-	-	-	-	-	-	-	-
T2: lock(R) x2	1	-	-	-	-	-	2	-	-	-
T1: lock(IR)	2	-	-	-	-	-	2	-	-	-
T2: lock(U)	2	-	-	-	-	-	2	1	-	-
T1: lock(R) x3	2	3	-	-	-	-	2	1	-	-
T1: unlock(IR)	1	3	-	-	-	-	2	1	-	-
T1: unlock(R) x2	1	1	-	-	-	-	2	1	-	-
T2: lock(W) - **denied**	1	1	-	-	-	-	2	1	-	-
T1: unlock(R)	1	-	-	-	-	-	2	1	-	-
T2: lock(IW)	1	-	-	-	-	-	2	1	1	-
T2: drop locks	1	-	-	-	-	-	-	-	-	-

Table 7.9: *Multiple Possession Semantics*

To illustrate multiple possession semantics Table 7.9 shows the internals of a lock set as two transactions acquire and release locks over a short period.

Note: * x2 means the operation is repeat.ed

Note the following points:

1. Transaction `T1` starts by acquiring an intention read lock and a write lock. This is permitted because conflicts only occur between unrelated transactions.
2. When transaction `T1` acquires the write lock, its intention read lock is not released.
3. Transaction `T2` acquires two read locks and transaction `T1` acquires another intention read lock. This increases the count of locks held for these transactions.
4. When `T1` unlocks a single intention read lock, the lock set still contains one intention read lock for `T1` because its count is decreased to 1. When `T1` unlocks its final read lock, the lock is released and its count is decreased to 0.
5. Transactions `T2`'s attempt to acquire a write lock is denied since this conflicts with the read and intention read lock held by `T1`.
6. When `T2` drops its locks, all locks held by `T1` are released.

Using the OCCS

The OCCS, like the transaction service, is implemented as a library and not as an external server program. The IDL for the OCCS is contained in the `CosConcurrencyControl` module and a C++ implementation for the IDL interfaces is available in all OrbixOTS servers. Within IDL files, the `CosConcurrencyControl` module may be accessed by including the file `OrbixOTS.idl`. Within server source files the C++ mapping is accessible by including the file `OrbixOTS.hh`.

Lock Modes and Exceptions

The enumeration type `lock_mode` defines the five lock modes, as shown in Figure 7.10. There is one exception named `LockNotHeld`, which is used when a request to release a lock is made by a transaction that does not hold the lock.

```
// IDL (module CosConcurrencyControl)
enum lock_mode {
  read,
  write,
  upgrade,
  intention_read,
  intention_write
};

exception LockNotHeld{};
```

Figure 7.10: *Lock Modes and Exceptions*

Implicit Lock Sets

The interface `LockSet` in Figure 7.11 is used for implicit lock sets. Operations are provided to acquire and release locks on a lock set object on behalf of the current transaction. There is also an operation to get a reference to the transaction's lock coordinator so that all locks held by the transaction may be dropped when the transaction completes. Implicit lock sets are created using a lock set factory; see "Creating Lock Set Objects" on page 119.

```
// IDL (module CosConcurrencyControl)
interface LockSet {

  void lock(in lock_mode mode);

  boolean try_lock(in lock_mode mode);

  void unlock(in lock_mode mode)
    raises(LockNotHeld);

  void change_mode(in lock_mode held_mode,
                   in lock_mode new_mode)
    raises(LockNotHeld);

  LockCoordinator get_coordinator(in CosTransactions::Coordinator which);
};
```

Figure 7.11: *IDL for Implicit Lock Sets*

The operation `lock()` acquires a single lock in a specific mode. If the lock mode conflicts with another lock held by another unrelated transaction, the operation blocks until the conflict is resolved or until the requesting transaction rolls back.

The following code illustrates acquiring a read lock for the current transaction:

```
CosConcurrencyControl::LockSet_ptr lockset = ...
lockset->lock(CosConcurrencyControl::read);
```

A `lock()` operation that blocks causes the request to be added to a queue. When the conflict is resolved, requests on the queue are serviced in a first-in first-out (FIFO) order.

If blocking when there is a conflict is unacceptable, the operation `try_lock()` can be used. This attempts to acquire a single lock in a specific mode, but if there is a conflict, a value of `FALSE` is returned. A return value of `TRUE` means that the lock was successfully acquired. For example, the following code attempts to acquire an upgrade lock:

```
CosConcurrencyControl::LockSet_ptr lockset = ...
lockset->try_lock(CosConcurrencyControl::upgrade);
```

The operation unlock() releases a single lock in a specific mode. Note that because a transaction may hold several locks in the same mode, calling unlock() does not always release the lock. If the transaction does not hold a lock in the specified mode, the exception LockNotHeld is raised. The following code releases a single write lock on behalf of the current transaction:

```
CosConcurrencyControl::LockSet_ptr lockset = ...
try {
  lockset->unlock(CosConcurrencyControl::write);
}
catch (CosConcurrencyControl::LockNotHeld) {
...
}
```

Releasing all locks held by the current transaction is done by invoking an operation on the transaction's lock coordinator. The get_coordinator() operation is used to obtain a reference to the lock coordinator. See "Dropping Locks" on page 120 for details.

Lastly, the operation change_mode() is provided to change the mode of a single lock. Both the original mode and the new mode are specified, and if the transaction does not hold a lock in the original mode the exception LockNotHeld is raised. For example, to change an upgrade lock to a write lock the following code may be used:

```
CosConcurrencyControl::LockSet_ptr lockset = ...
try {
  lockset->change_mode(CosConcurrencyControl::upgrade,
                       CosConcurrencyControl::write);
}
catch (CosConcurrencyControl::LockNotHeld) {
...
}
```

Explicit Lock Sets

Explicit lock sets are supported by the `TransactionalLockSet` interface shown in Figure 7.12. This provides the same operations as the interface `LockSet`, except the operations `lock()`, `try_lock()`, `unlock()` and `change_mode()` all take an extra parameter that is a reference to the transaction coordinator on whose behalf the operations are performed. Explicit lock sets are created using a lock set factory; refer to "Creating Lock Set Objects" on page 119 for details.

```
// IDL (module CosConcurrencyControl)
interface TransactionalLockSet {

  void lock(in CosTransactions::Coordinator which,
            in lock_mode mode);

  boolean try_lock(in CosTransactions::Coordinator which,
                   in lock_mode mode);

  void unlock(in CosTransactions::Coordinator which,
              in lock_mode mode)
    raises(LockNotHeld);

  void change_mode(in CosTransactions::Coordinator which,
                   in lock_mode held_mode,
                   in lock_mode new_mode)
    raises(LockNotHeld);

  LockCoordinator get_coordinator(in CosTransactions::Coordinator which);
};
```

Figure 7.12: *IDL for Explicit Lock Sets*

The following code shows how an intention read lock is acquired on an explicit lock set:

```
CosConcurrencyControl::TransactionalLockSet_ptr lockset = ...
CosTransactions::Coordinator_ptr coord = ...
lockset->lock(coord, CosConcurrencyControl::intention_read);
```

Creating Lock Set Objects

Implicit and explicit lock sets are created using a lock set factory provided by the `LockSetFactory` interface, shown in Figure 7.13. Two operations are supported: `create()` returns a reference to a new implicit lock set object, and `create_transactional()` returns a reference to a new explicit lock set object.

```
// IDL (module CosConcurrencyControl)
interface LockSetFactory {

  LockSet create();

  TransactionalLockSet create_transactional();
  …
};
```

Figure 7.13: *IDL for Lock Set Factory*

Each OrbixOTS server has a lock set factory object which can be obtained using the `get_lockset_factory()` operation provided by the `OrbixOTS::Server` class. Creating a lock set object involves first obtaining the lock set factory reference and invoking either `create()` or `create_transactional()`. This is illustrated with the following code:

```
OrbixOTS::Server_var ots = ...
CosConcurrencyControl::LockSetFactory_var Factory =
    ots->get_lockset_factory();

// Create an implicit lock set object.
CosConcurrencyControl::LockSet_ptr lockset =
    factory->create();

// Create an explicit lock set object.
CosConcurrencyControl::LockSet_ptr lockset2 =
    factory->create_transactional();
```

Dropping Locks

The LockCoordinator interface shown in Figure 7.14 provides a means of dropping all locks held by a transaction. This is useful when using strict-2PL, where it is necessary to drop all locks when a transaction completes. Refer to "Two-Phase Locking" on page 110 for further information on this topic. The lock coordinator object is obtained by invoking the operation get_coordinator() on a lock set object.

```
// IDL (module CosConcurrencyControl)
interface LockCoordinator {
  void drop_locks();
};
```

Figure 7.14: *IDL for Lock Coordinator*

The following code illustrates dropping all locks held by a transaction in an implicit lock set. Note that when calling the get_coordinator() operation, a reference to the transaction's coordinator must be passed as a parameter.

```
CosTransactions::Coordinator_ptr coord = ...
CosConcurrencyControl::LockSet_ptr lockset = ...
CosConcurrencyControl::LockCoordinator_var lockCoord;

// Get the lock coordinator from the lock set object.
lockCoord = lockset->get_coordinator(coord);

// Drop all locks.
lockCoord->drop_locks();
```

Note that with nested transactions, locks should only be released when the sub-transaction rolls back. When a sub-transaction commits, its locks are inherited by the parent transaction.

8

Advanced XA Programming

OrbixOTS allows resources such as databases and messaging systems to be easily integrated if they provide an XA interface. This text discusses the XA interface and shows how an XA resource manager is integrated with OrbixOTS. Other issues are also discussed including concurrency, using explicit propagation, caching data, and support for nested transactions.

Overview of XA

Figure 8.1 on page 122 shows a three tier application in which the OrbixOTS server in the middle makes use of two XA-compliant resource managers. For example, resource manager *A* could be a relational database and resource manager *B* could be a message queue system. The clients can create transactions, invoke operations on the server which may access both of the resource managers and commit the transaction. Because both resource mangers support the XA protocol, the integrity of their data is insured.

To OrbixOTS, the XA protocol consists of ten functions (provided by the resource manger in a library) which are called at certain times. The names of the ten functions and their purpose is given. Refer to the XA specification for complete information.

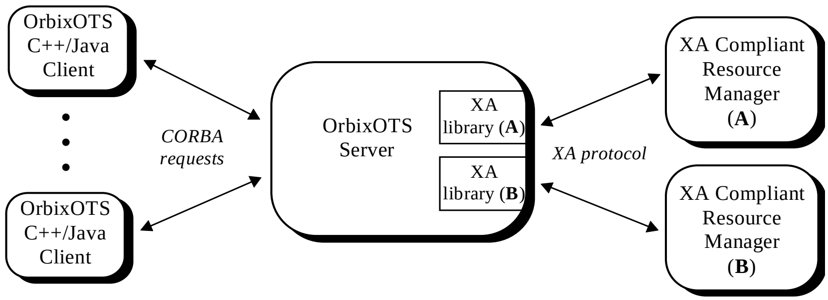


Figure 8.1: *XA Resource Managers and OrbixOTS*

XA Operation	Purpose
`xa_open()`	Opens the connection to the resource manager. This is called during initialization.
`xa_close()`	Closes the connection to the resource manager.
`xa_start()`	Informs the resource manager that a thread or process has started working on behalf of a transaction.
`xa_end()`	Informs the resource manager that a thread or process has finished working on behalf of a transaction.
`xa_rollback()`	Rolls-back modifications to the resource made by a transaction.
`xa_prepare()`	Prepares the resource manager for eventual commitment of a transaction. The resource manager returns its vote.
`xa_commit()`	Commits modifications to the resource made by a transaction.

Table 8.2: *The XA Protocol Functions*

XA Operation	Purpose
xa_recover()	Retrieves the identifiers of transactions for which the resource manager needs to know the final outcome. This is called during initialization.
xa_forget()	Informs the resource manager that a heuristic decision may be forgotten.
xa_complete()	Completes an asynchronous call.

Table 8.2: *The XA Protocol Functions*

These XA functions are called automatically by OrbixOTS when the implicit/indirect programming model is used.

Integrating an XA Resource Manager

Integrating an XA resource manager with an OrbixOTS server is done via the `OrbixOTS::Server::register_xa_rm()` operation:

```
CORBA::Long register_xa_rm(const xa_switch_t* xasw,
                const char* openString,
                const char* closeString,
                const CORBA::Boolean isThreadAware
```

This operation must be invoked before the `OrbixOTS::Server::init()` operation is invoked. It returns an integer that is the local resource manager identifier.

The parameters to the operation are shown in Table 8.3.

Parameter	Description
xasw	A pointer to a variable of type `xa_switch_t` which contains pointers to the resource manager's XA functions. Refer to your resource manager documentation for the name of this variable.

Table 8.3: *The* `register_xa_rm()`*Parameters*

Parameter	Description
openString	A string used to initialize the connection to the resource manager. This is passed as a parameter to the xa_open() function. Refer to your resource manager documentation for the correct value of this string.
closeString	A string used to close the connection to the resource manager. This is passed as a parameter to the xa_close() function. Refer to your resource manager documentation for the correct value of this string.
isThreadAware	A boolean that indicates whether the XA library supplied by the resource manager is thread-safe. See "Single Association versus Multiple Associations" on page 128 for a discussion of this parameter.

Table 8.3: *The* register_xa_rm()*Parameters*

The xa_switch_t structure looks like this:

```
struct xid_t {
  long formatID;
  long gtrid_length;
  long bqual_length;
  char data[128];
};

typedef struct xid_t XID;
```

```
struct xa_switch_t {
     char name[32];
     long flags;
     long version;
     int (*xa_open_entry)(char*, int, long);
     int (*xa_close_entry)(char*, int, long);
     int (*xa_start_entry)(XID*, int, long);
     int (*xa_end_entry)(XID*, int, long);
     int (*xa_rollback_entry)(XID*, int, long);
     int (*xa_prepare_entry)(XID*, int, long);
     int (*xa_commit_entry)(XID*, int, long);
     int (*xa_recover_entry)(XID*, long, int, long);
     int (*xa_forget_entry)(XID*, int, long);
     int (*xa_complete_entry)(int*,int*,int,long);
};
```

Use of the `register_xa_rm()` operation is illustrated in the following code, which integrates an Oracle database with OrbixOTS:

```
const char* openString = "Oracle_XA+Acc=P/scott/tiger+SesTm=60";
const char* closeString = "";
OrbixOTS::Server_var ots = ...
CORBA::Long rm_id;

rm_id = ots->register_xa_rm(&xaosw, openString, closeString, 0);
```

The parameters are described as follows:

- The name of the XA switch variable provided by the Oracle XA library is `xaosw`.
- The open string consists of the connection name, the account user (scott), and password (tiger) and a session timeout of 60 seconds.
- The close string is empty.
- The XA library is not thread-safe.

Note: For automatic management of an XA resource manager, implicit propagation must be used. Explicit propagation can be used, but this requires extra programming. See “Explicit Propagation” on page 131.

Concurrency Issues

There are a number of issues involving concurrency that need special attention when integrating XA resource managers. These are the use of resource manager locks, the server's concurrency mode, and thread-aware XA libraries.

Resource Manager Locks

Each resource manager is responsible for synchronizing access to its data. Typically this means that locks are acquired when data is accessed, and these locks are only released when the transaction holding the locks completes (either commits or rolls-back). This can lead to deadlock if locks are acquired out of sequence (see Table 8.4). Applications should follow the resource manager's guidelines to avoid these situations.

Transaction 1	Transaction 2
lock resource A	
	lock resource B
lock resource B **BLOCKS!**	
	lock resource A **BLOCKS!**

Table 8.4: *Deadlock with Resource Manager Locks*

Unless a cache is being used (see "Synchronizing Cache Data" on page 132) the application itself does not normally have to provide concurrency control for accessing resource manager data.

Concurrency Modes

OrbixOTS servers support three different concurrency modes. These are:

`serializeRequestsAndTransactions`

> This is the most conservative mode and hence the default. Once a transaction enters the server, that transaction has exclusive access to the server until the transaction completes (either commits or rolls back). In addition, concurrent requests from the same transaction are serialized.
>
> Using this mode is simple because neither requests nor transactions are interleaved at the server. Hence there is no need for any concurrency control within the server.

`serializeRequests`

> This mode falls half way between `concurrent` and `serializeRequestsAndTransactions`. Requests are serialized but transactions may be interleaved.

`concurrent`

> This is the most liberal of the modes. Both requests and transactions can be interleaved in the server. This mode is implemented using a pool of threads that are used to dispatch requests.
>
> Using this mode requires careful programming because code that accesses the resource manager (for example, embedded SQL or a propriety application programming interface) must be thread-safe.

The concurrency mode is specified as a parameter to the `OrbixOTS::Server::impl_is_ready()` operation. For example, a fully concurrent OrbixOTS server uses code such as the following:

```
OrbixOTS::Server_var ots = ...
...
ots->impl_is_ready(OrbixOTS::Server::concurrent);
```

Single Association versus Multiple Associations

A thread becomes associated with a resource manager when `xa_start()` is called and the association continues until `xa_end()` is called. In addition, calls to functions such as `xa_prepare()`, `xa_commit()`, and `xa_rollback()` cause the current thread to be associated with the resource manager for the duration of the function call. An XA library that permits multiple threads to be associated at any one time with the resource manager is said to support *multiple associations*; otherwise only a *single association* is supported.

When registering an XA resource manager with OrbixOTS, the `isThreadAware` parameter to the `register_xa_rm()` operation indicates whether the XA library supports multiple associations. A value of `1` (true) means the XA library is thread-aware and thus supports multiple associations. A value of `0` (false) means the XA library is not thread-aware and only supports a single association.

If the `isThreadAware` parameter is `0` (false), OrbixOTS uses a lock to serialize access to the XA library. This lock is acquired when `xa_start()` is called, and released with `xa_end()` is called. Thus the XA lock is held for the duration of a request, which effectively serializes all requests.

Using the concurrency modes `concurrent` or `serializeRequests` with a single association XA library can easily lead to deadlock.

Consider two transactions trying to update a resource manager (see Table 8.5).

Transaction 1	Transaction 2
acquire XA lock	
acquire RM lock update RM	
release XA lock	
	acquire XA lock
	acquire RM lock **BLOCKS!**
acquire XA lock **BLOCKS!**	

Table 8.5: *Deadlock with Single Association XA Library*

The first transaction, transaction 1, acquires a resource manager lock and updates some data. The second transaction, transaction 2, tries to acquire the same resource manager lock but fails because this lock is already held by transaction 1. This causes transaction 2 to block while holding the XA lock. Then transaction 1 tries to update the data again, but is blocked while trying to acquire the XA lock. Further, other transactions will also be blocked even if they are not accessing the resource manager. Eventually the resource manager may timeout its lock and cause transaction 2 to be rolled back.

This situation does not occur with the `serializeRequestsAndTransactions` concurrency mode because transaction 2 would not be allowed to acquire the XA lock until transaction 1 completed.

The following grid shows the recommended concurrency modes to use with single and multiple associations:

Associations	**serializeRequests & serializeRequestsAndTransactions**	**concurrent**
Single	recommended	not recommended
Multiple	not recommended	recommended

Table 8.6: *Concurrency Modes and Associations*

Some of the differences between single association and multiple associations are illustrated in Table 8.7. This shows the XA functions being called during initialization, servicing of two concurrent requests and the 2PC protocol. For the single association library, the xa_open() function is called once and the two requests are serialized. The multiple associations library calls xa_open() each time a new thread accesses the resource manager and the two requests are interleaved.

Activity	**Single Association**	**Multiple Associations**			
		Thread 1	Thread 2	Thread 3	Thread 4
1. Server initialization	`xa_open()` `xa_recover()`	`xa_open()` `xa_recover()`			

2. Two concurrent invocations	`xa_start()`
	`xa_end()`
	`xa_start()`
	`xa_end()`
			`xa_open()`
			`xa_start()`
			
			`xa_end()`
				
				`xa_open()`
				`xa_start()`
				`xa_end()`	

3. 2PC commit protocol	`xa_prepare()`
	`xa_commit()`
					`xa_open()`
					`xa_prepare()`
					`xa_commit()`

Table 8.7: *Single Association Versus Multiple Associations*

Explicit Propagation

When using implicit propagation, the thread performing the invocation on the server side is associated with the current transaction. With explicit propagation, there is no such association which means that OrbixOTS cannot automatically make the required XA function calls. Thus, using explicit propagation requires extra work on the server side to create the association between the current thread and the transaction. The following code sample shows the steps required:

```
void TransAccount_i::makeLodgement(CORBA::Float amount,
CosTransactions::Control_ptr control)
{
  // Get coordinator object from control object.
  CosTransactions::Coordinator_var coord;
1 coord = control->get_coordinator();

  // Get the transaction propagation context.
  CosTransactions::PropagationContext_var context;
2 context = coord->get_txcontext();

  // Get a reference to the local transaction factory.
   OrbixOTS::Server_var ots = ...
3  CosTransactions::TransactionFactory_var factory =
      ots->get_transaction_factory();

  // Recreate the transaction locally.
4 control = factory->recreate(context);

  // Associate the current thread with the transaction
  // Assume current is a reference to the Current pseudo object
5 current->resume(control);

6 // Perform deposit(lodgement) operation as normal.
  ...

  // Disassociate the current thread from the transaction.
7 CosTransactions::Control_var control2;
  control2=current->suspend();

}
```

The steps are explained as follows:

1. Use the control object (passed as a parameter) to get the coordinator object.
2. The coordinator object is then used to get the transaction's propagation context using the `get_txcontext()` operation.
3. Get a reference to the local transaction factory using the `get_transaction_factory()` operation provided by the `OrbixOTS::Server` class.
4. Recreate the transaction locally using the `recreate()` operation provided by the transaction factory.
5. Associate the new local transaction with the current thread using the `resume()` operation provided by the `Current` pseudo object.
6. Perform the operation as normal. All accesses to the XA resource manager will take place in the context of the transaction.
7. Disassociate the current thread from the transaction using the `suspend()` operation provided by the `Current` pseudo object.

This approach is not a recommended for typical situations because extra remote invocations are required, as well as the extra coding.

Synchronizing Cache Data

In the discussion of using XA resource managers, we have assumed that the server application always contacts the resource manager each time it needs to access any data. However, it is likely that applications will cache data in the server to increase performance. This raises the problem of what to do when the transaction is committed. When an XA resource manager is involved, the 2PC protocol only involves the resource manager; any data in the cache is ignored.

To solve this problem, OrbixOTS supports the interface `CosTransactions::Synchronization`. An object implementing the `Synchronization` interface is registered with a transaction coordinator and is invoked both before and after the 2PC protocol. This is the interface:

```
// In CosTransactions module.
interface Synchronization : TransactionalObject {
    void before_completion();
    void after_completion(in Status s);
```

```
};
```

The before_completion() Operation

This operation is invoked before the 2PC protocol is started, at the coordinator with which the synchronization object is registered. This means that it is invoked before any XA resource managers have been prepared.

An implementation may flush all cache data to the resource manager so that when the 2PC protocol commences, the data in the resource manager is correct.

Raising a system exception will cause the transaction to be rolled back. The transaction may also be rolled back by invoking one of the operations `rollback()` or `rollback_only()` on the `Current` pseudo object.

The after_completion() Operation

This operation is invoked after all commit and rollback responses have been received by the coordinator with which the synchronization object is registered. It is passed the current status of the transaction so the operation can know whether the transaction has committed or rolled-back.

An implementation can use this operation to release locks held on the cache. Raising a system exception in this operation has no effect on the outcome of the transaction.

Registering a Synchronization Object

A synchronization object is registered using the `register_synchronization()` operation provided by the `Coordinator` interface. Assuming the class `Synchronization_i` implements the `Synchronization` interface, the following code may be used:

```
// Get a reference to the coordinator for the current transaction.
CosTransactions::Current_var current = ...
CosTransactions::Control_var control =
    current->get_control();
CosTransactions::Coordinator_var coord =
    control->get_coordinator();
```

```
// Create a synchronization object and register it with
// the transaction in the code example.
CosTransactions::Synchronization_var sync =
   new Synchronization_i(...);
coord->register_synchronization (sync);
```

The `register_synchronization()` operation raises the `Inactive` exception if the transaction has already been prepared. Note that a synchronization object must only be registered once for a given transaction. Thus the application code should maintain a list of currently active transactions, and only register a new synchronization object the first time a transaction accesses the cache.

Concurrency Issues

Using a cache with an XA resource manager may require that the application deal with concurrent transactions. This arises if the serialization modes `concurrent` or `serializeRequests` are used. Two possible strategies are:

- Synchronize access to the cache using some locking mechanism. The OCCS implementation provided with OrbixOTS may be used here.
- Provide a cache for each transaction. For this, the application must map between transactions and caches (using the `is_same_transaction()` and `hash_transaction()` operations provided by the `Coordinator` interface, for example), and check when a new transaction is involved so a new cache can be created.

Nested Transactions

The X/Open DTP model does not support the notion of nested transactions. However, OrbixOTS supports the use of nested transactions with XA resource managers with some restrictions. Each transaction in a family must be mapped to XA style transactions, and there are four different models to choose from. Setting the mapping model is done with the `tmxa_SetNestingModel()` operation provided by the Encina toolkit. (This and other `tmxa` functions call Encina directly.)

```
#include <tmxa/tmxa.h>
#include <tran/tran.h>
```

```
tmxa_status_t tmxa_SetNestingModel(tran_tid_t tid, int scope,
                                   int nestingModel)
```

The return value is `TMXA_SUCCESS` if the operation succeeded or an error code if the operation failed. The first two parameters (`tid` and `scope`) must have the values `TRAN_TID_NUL` and `TMXA_NEW_TOP_LEVEL_TIDS` respectively. The `nestingModel` parameter specifies the mapping model and can be one of the following four values:

`TMXA_DIFFERENT_GTRID`

Each transaction in a family is mapped to a unique global XA transaction. This is the default model.

`TMXA_SAME_XID`

All transactions in a family are mapped to the same global XA transaction.

`TMXA_DIFFERENT_BQUAL_INDEPENDENT`

All transactions in a family are mapped to the same global XA transaction, but each transaction has a different branch qualifier. Each branch may commit or rollback independently.

`TMXA_DIFFERENT_BQUAL_LINKED`

All transactions in a family are mapped to the same global XA transaction, but each transaction has a different branch qualifier. All branches share the same outcome.

Each model effects the ACID properties of transactions and the choice of model will depend on the nature of the application.

Note: Using nested transactions with XA resource managers should be approached with caution, and only after the consequences of doing so in your application have been studied in detail.

One-Phase-Commit Optimization

The X/Open XA specification includes a one-phase-commit (1PC) protocol as an optimization. This means that only one registered XA resource manager is involved in committing a transaction. By default OrbixOTS does not use this optimized protocol, but under certain conditions it can be turned on in a server using the `tmxa_SetUsesOnlyLocalXaWork()` function provided as part of the Encina toolkit:

```
#include <tmxa/tmxa.h>
tmxa_status_t tmxa_SetUsesOnlyLocalXaWork(
                tran_tid_t tid,
                int scope,
                int onlyLocalXaFlag);
```

This function returns `TMXA_SUCCESS` if it is successful; otherwise it returns one of the following Encina errors:

```
TMXA_INVALID_PARAM
TMXA_NO_MORE_MEMORY,
TMXA_NOT_INITIALIZED
TMXA_TOO_MANY_TIDS
```

There are two conditions that must be met before this function can be used:

1. There are no `CosTransaction::Resource` objects registered with transactions.
2. All XA resource managers involved in the transaction are registered with a single OrbixOTS server.

The `tmxa_SetUsesOnlyLocalXaWork()` function can be used even if a server has several registered XA resource managers. The 1PC/XA optimization only takes effect if all but one of the resource managers returns `XA_RDONLY` (indicating a read-only transaction) from the `xa_prepare()` function.

The following code shows how to turn on the 1PC/XA optimization, which should be done after initializing the OTS:

```
tmxa_status_t status;
status = tmxa_SetUsesOnlyLocalXaWork(0,
              TMXA_NEW_TOP_LEVEL_TIDS,
              TMXA_ONLY_LOCAL_XA_WORK);
if (status != TMXA_SUCCESS)
{  // Error, function failed…
}
```

This turns on the IPC/XA optimization for all new top-level transactions. Passing a value of TMXA_NOT_ONLY_LOCAL_XA_WORK for the third parameter turns off the IPC/XA optimization. This function can also be used for a specific transaction. For example, the following code turns off the IPC/XA optimization for the current transaction:

```
CosTransactions::Current_var current = …
CosTransactions::Control_var control;
control = current->get_control();
// Get Encina transaction identifier
tran_tid_t = control->id();
tmxa_status_t status;
status = tmxa_SetUsesOnlyLocalXaWork(
                tid,
                TMXA_THIS_TID,
                TMXA_NOT_ONLY_LOCAL_XA_WORK);
```

This can be used, for example, if the application wants to register a resource object with the transaction.

Other Issues

Two final issues to mention include resource manager APIs and database cursors.

Resource Manager APIs

There are some restrictions on the use of a resource manager's API when integrating it with OrbixOTS.

All calls to create and terminate transactions must be done using OrbixOTS APIs, and not using the resource manager's API. For example, transactions can be created using the `begin()` operation provided by the `Current` pseudo object but never using an embedded `SQL BEGIN` statement.

Any connections to the resource manager are established during the `xa_open()` function using the value of the `openString` parameter to the operation `register_xa_rm()`. Thus, resource manager APIs that establish connections must not be used.

If an administrator uses manual intervention to force incompleted transactions to commit or rollback, this must be done using the OrbixOTS `otsadmin` tool and never using tools provided by the resource manager. This is to ensure that OrbixOTS has a consistent view of the state of its transactions.

Database Cursors

Due to the way OrbixOTS uses the `xa_start()` and `xa_end()` functions, database cursors cannot be used across invocation boundaries. For example, once `xa_end()` is called, any open cursors are invalidated.

Part IV

Programmer's Reference

9

OrbixOTS Reference Overview

Figure 9.1 illustrates the IDL interfaces defined by the OTS specification, with an indication of the entities that use them.

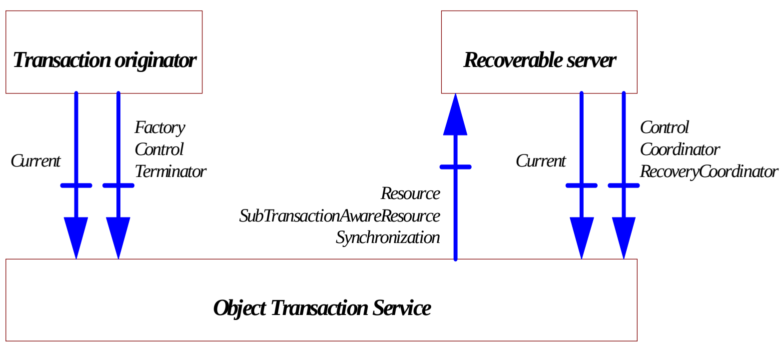


Figure 9.1: *OTS IDL Interfaces*

The transaction originator is the component of the system that needs to begin and complete transactions, as well as invoke recoverable servers. These are processes that contain objects whose state changes need to be managed atomically with distributed transactions.

Interfaces

The text following describes each interface in turn. Each interface is defined in a module called `CosTransactions`.

Current

This pseudo-interface allows a transaction client to begin and complete transactions. It also provides operations for suspending and resuming transactions by which a thread can associate and disassociate itself from active transactions. Use of the `Current` pseudo-object can be seen as an indirect way of accessing the "real" transactional interfaces, in the following sections.

Control

Instances of this interface represent the transaction. It is simply an encapsulation of two other objects which provide operations for transaction manipulation: a `Coordinator` and a `Terminator`. Two operations are supported that return references to these contained objects.

Coordinator

This interface provides a variety of operations for obtaining information about the transaction. It also exposes the `rollback_only()` operation, by which the transaction may be marked for rollback, but not actually rolled back. The main function of `Coordinator` is to allow a recoverable object to register a `Resource` (or `SubtransactionAwareResource`) to be called back on transaction completion.

Terminator

The Terminator object associated with a transaction provides two operations to complete the transaction: `commit()` and `rollback()`.

Resource

The Resource interface is called by the OTS on transaction completion. It exposes operations supporting a two-phase commit protocol: `prepare()`, `commit()`, `rollback()`, `forget()`, and `commit_one_phase()`.

TransactionalObject

This empty interface is used by the OTS to determine if the transaction context should be implicitly transferred to a remote object. If the remote object inherits from `TransactionalObject` then the OTS transparently "piggy-backs" the transaction information to be extracted by the OTS library at the other end.

TransactionFactory

This interface serves as a transaction (or, more specifically, `Control`) creation factory.

SubtransactionAwareResource

This is similar to a Resource, in that it is implemented by the user of the OTS, and is called back on transaction completion—however it is specific to completion of a nested transaction.

RecoveryCoordinator

A reference to a `RecoveryCoordinator` is returned to a transactional object when a `Resource` is registered with the `Coordinator`. The server should save this reference as it can be used to resolve transactions that are in doubt. After the transaction is prepared, the server can call `replay_completion()` on this object as a hint to the coordinator that `commit()` or `rollback()` have not been called yet.

Synchronization

This callback object is implemented by the OTS user, and is registered with the `Coordinator` in exactly the same fashion as a `Resource` object. The OTS informs it of transaction completion, as for a `Resource`. However the operations it implements do not involve a two-phase commit; instead the two operations `before_completion()` and `after_completion()` are called before and after the two-phase commit process. Synchronization objects are intended for use with caching systems to inform them when to flush the cache to a more permanent store, and can drive the release of locks acquired through an OCCS interface.

Java Classes

OrbixOTS provides a set of Java classes for use with the Orbix Java Edition object request broker. Orbix Java Edition allows you to build distributed applications in the Java language. The OrbixOTS Java interfaces allow you to write Java client and server applications that begin and control transactions by using the Java-language implementation of the OTS IDL.

10

The Classes Client, Restart, and Server

There are three interfaces in the OrbixOTS module: Client, Restart and Server. The Client class initializes clients. The Server class initializes server applications, manages server objects, and registers various resources. OrbixOTS also provides a Restart class for restarting servers.

The `Client` interface is used in C++ OrbixOTS clients, and provides the following functionality:

- Initialization
- Termination
- Setting transaction policies on objects and interfaces

The `Server` and `Server` interfaces are used in C++ OrbixOTS servers and provides the following functionality:

- Initialization
- Termination
- Specification of local or remote logging
- Integration of XA-compliant resource managers
- Recovery support

- Setting transaction policies on objects and interfaces
- Getting references to local transaction factory and lock set factory objects

The header file provided with OrbixOTS that defines client and server functionality is `OrbixOTS.hh`. The following is the pseudo IDL code for the OrbixOTS module:

```
module OrbixOTS {
  interface Client {
    void init();
    void exit(in long status);
  };
  interface Restart {
    void recovery();
  };
  interface Server {
    attribute string serverName;
    attribute string logDevice;
    attribute string restartFile;
    attribute string mirrorRestartFile;
    attribute string logServer;
    long register_xa_rm(in any xasw,
             in string openString,
             in string closeString,
             in boolean isThreadAware);
    void recoverable(in Restart obj);
    void init();
    enum ConcurrencyMode {
      concurrent,
      serializeRequests,
      serializeRequestsAndTransactions
    };
    void impl_is_ready(in ConcurrencyMode mode);
    void exit(in long status);
  };
};
```

The OrbixOTS classes provide the following functionality:

`OrbixOTS::Client`	This class initializes and terminates client applications.

`OrbixOTS::Restart`	This class performs recovery for servers that are registered as recoverable.
`OrbixOTS::Server`	This class initializes and terminates server applications, registers resources, and registers servers as recoverable.

OrbixOTS::Client Class

Synopsis

The CORBA specification defines a standard set of `Client` methods and IONA adds a number of IONA-specific methods.

CORBA

```
class OrbixOTS::Client : public CORBA::IT_PseudoIDL {
public:
   Client();
   virtual ~Client();
   static Client_ptr IT_create(
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
   static void init(
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
   static void exit(const CORBA::Long status = 0,
      CORBA::Environment& IT_env=CORBA::IT_chooseDefaultEnv());
};
```

IONA

```
enum TransactionPolicy {
                transactionRequired,
                transactionAllowed,
        };

static void setObjectTransactionPolicy(
                    CORBA::Object_ptr obj,
                   TransactionPolicy policy = transactionRequired);
static void setInterfaceTransactionPolicy(
                    const char *interface,
                   TransactionPolicy policy = transactionRequired);
static TransactionPolicy
            setDefaultTransactionPolicy(TransactionPolicy policy);
static TransactionPolicy
             getDefaultTransactionPolicy();
```

Description The `OrbixOTS::Client` class represents a client application. This class encapsulates functions for initializing and terminating a client application. Because the member functions are static functions, it is not necessary to create an instance of the class to use in initializing or terminating a client application.

Class Members

```
OrbixOTS::Client::exit()
OrbixOTS::Client::getDefaultTransactionPolicy()
OrbixOTS::Client::init()
OrbixOTS::Client::IT_Create()
OrbixOTS::Client::setDefaultTransactionPolicy()
OrbixOTS::Client::setInterfaceTransactionPolicy()
OrbixOTS::Client::setObjectTransactionPolicy()
OrbixOTS::Client::shutdown()
```

See Also "`OrbixOTS::Server Class`" on page 153

OrbixOTS::Client::shutdown()

Synopsis

```
static void shutdown ()
   throw (CORBA::SystemException);
```

Description Shuts down the OTS and rolls back all outstanding transactions.

See Also "`OrbixOTS::Client::exit()`" on page 148

OrbixOTS::Client::exit()

Synopsis

```
static void exit(const CORBA::Long status = 0,
   CORBA::Environment& IT_env=CORBA::IT_chooseDefaultEnv());
```

Parameters The `status` parameter specifies the exit status for the client application. If no value is specified, an exit status of `0` (zero), indicating successful termination, is used by default.

Description The `exit()` function terminates a client application. A value can be specified to indicate the exit status for the termination. If any transactions are in progress when the `exit()` function is called, all outstanding transactions are aborted before the client application is terminated. The standard exception `CORBA::SystemException` may be thrown.

See Also “OrbixOTS::Client::init()” on page 149

OrbixOTS::Client::getDefaultTransactionPolicy()

Synopsis

```
static TransactionPolicy
        getDefaultTransactionPolicy();
```

Description The `getDefaultTransactionPolicy()` function gets the current default `TransactionPolicy`.

Returns The current default `TransactionPolicy`.

Notes IONA-specific.

See Also “OrbixOTS::Client::setDefaultTransactionPolicy()” on page 150

“OrbixOTS::Client::setInterfaceTransactionPolicy()” on page 150

“OrbixOTS::Client::setObjectTransactionPolicy()” on page 151

OrbixOTS::Client::init()

Synopsis

```
static void init(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description The `init()` function initializes the client application and the underlying services. A fatal error is generated if any errors occur during the initialization of the underlying components. The standard exception `CORBA::SystemException` may be thrown.

See Also “OrbixOTS::Client::exit()” on page 148

OrbixOTS::Client::IT_create()

Synopsis

```
void Client_ptr IT_create()
   throw (CORBA::SystemException)
```

Description Creates an instance of a `Client` pseudo-object. `IT_create()` should be used in preference to the C++ operator new but only when there is no (suitable) compliant way to obtain a pseudo-object reference. This will ensure memory management consistency. `IT_create()` returns a pointer to a new `Client` pseudo object.

OrbixOTS::Client::setDefaultTransactionPolicy()

Synopsis

```
static TransactionPolicy
            setDefaultTransactionPolicy(TransactionPolicy policy);
```

Parameters The `policy` parameter specifies the current default `TransactionPolicy`.

Description The `setDefaultTransactionPolicy()` function sets the default `TransactionPolicy`. The default transaction policy is `TransactionRequired`, in which case both the client and server throw a `TRANSACTION_REQUIRED` exception if an invocation on a transactional object is outside the scope of a transaction. You may also choose to change the default transaction policy to `TransactionAllowed`. In this case all transactional objects can process requests outside the scope of a transaction. All newly-created objects take on the default behaviour unless they implement an interface that has a particular policy selected. In the case where the default is changed, `TransactionRequired` semantics need to be explicitly set for individual interfaces of objects.

Returns The previous `TransactionPolicy`.

Notes IONA-specific.

See Also "OrbixOTS::Client::getDefaultTransactionPolicy()" on page 149

"OrbixOTS::Client::setInterfaceTransactionPolicy()" on page 150

"OrbixOTS::Client::setObjectTransactionPolicy()" on page 151

OrbixOTS::Client::setInterfaceTransactionPolicy()

Synopsis

```
static void setInterfaceTransactionPolicy(
                    const char *interface,
                  TransactionPolicy policy = transactionRequired);
```

Parameters

`interface`	The interface to treat as transactional.
`policy`	The `TransactionPolicy` for this transactional interface.

Description The `setInterfaceTransactionPolicy()` function marks an interface as transactional and specifies the transaction policy for this transactional interface. Objects that support this interface are treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface. The `interface` parameter is the CORBA repository identifier for the interface that is of the form "IDL:X:1.0".

Notes IONA-specific.

See Also "OrbixOTS::Client::getDefaultTransactionPolicy()" on page 149

"OrbixOTS::Client::setDefaultTransactionPolicy()" on page 150

"OrbixOTS::Client::setObjectTransactionPolicy()" on page 151

OrbixOTS::Client::setObjectTransactionPolicy()

Synopsis

```
static void setObjectTransactionPolicy(
                    CORBA::Object_ptr obj,
                   TransactionPolicy policy = transactionRequired);
```

Parameters

`obj`	The object to treat as transactional.
`policy`	The `TransactionPolicy` for this transactional object.

Description The `setObjectTransactionPolicy()` function marks an object as transactional and specifies the transaction policy for this transactional object. This object is treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface.

Notes IONA-specific.

See Also "OrbixOTS::Client::getDefaultTransactionPolicy()" on page 149

"OrbixOTS::Client::setDefaultTransactionPolicy()" on page 150

"OrbixOTS::Client::setInterfaceTransactionPolicy()" on page 150

OrbixOTS::Restart Class

Synopsis

```
class Restart : public CORBA::IT_PseudoIDL {
  public:
    Restart();
    virtual ~Restart();
    static Restart_ptr IT_create(CORBA::Environment& IT_env
                             = CORBA::IT_chooseDefaultEnv());
    virtual void recovery();
  };
```

Description

The `Restart` class is used to encapsulate a callback function registered for a server. A recovery callback is a function that is invoked when the server for which it is registered is restarted. Use the function `OrbixOTS::Server::recoverable()`to register a recovery callback with a server.

The `Restart` class `recovery()` function must be defined in a class derived from the `Restart` class. The `recovery()` function you define must perform the work required by the callback. Any data required by the callback can be encapsulated in the class declaration of the derived class.

If recoverable objects are being used, an instance of the `Restart` class must be registered for the server. The `recovery()` function defined for the `Restart` instance must reregister any `Resource` objects that require recovery during the restart of a server. (Resource objects are reregistered by using the associated `RecoveryCoordinator` instance and the `RecoveryCoordinator::replay_completion()` function provided.)

See Also

"OrbixOTS::Server Class" on page 153

"OrbixOTS::Server::recoverable()" on page 165

OrbixOTS::Restart::IT_create()

Synopsis

```
static Restart_ptr IT_create(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
```

Description Creates an instance of a `Restart` pseudo-object. `IT_create()` should be used in preference to the C++ operator `new` but only when there is no (suitable) compliant way to obtain a pseudo-object reference. This will ensure memory management consistency. `IT_create()` returns a pointer to a new `Restart` pseudo object.

OrbixOTS::Restart::recovery()

Synopsis

```
virtual void recovery();
```

Description A pure virtual operation that is invoked during restart processing on objects passed to the operation `recoverable()`. Classes inheriting from `Restart` must define this operation.

See Also "`OrbixOTS::Server::recoverable()`" on page 165

OrbixOTS::Server Class

Synopsis The `Server` interface is more complex than the `Client` or `Restart` interfaces since it is used to initialize recoverable OrbixOTS servers requiring a logging facility and it also provides functions to integrate XA-compliant resource managers. The following attributes, types, and functions are provided:

`serverName`	An attribute that specifies the name of the server. This only needs to be specified for persistent (manually launched) servers. The value is the same that one would normally be passed to `CORBA::BOA::impl_is_ready()`.
`logDevice`	An attribute that specifies the path for the transaction log used for recoverable servers. The path may refer to either an ordinary file or a raw disk partition.

`restartFile`	An attribute that specifies the path for the restart file that contains information about the log device being used (if any).
`mirrorRestartFile`	An attribute that specifies the path for the mirror restart file that contains information about the log device being used (if any).
`logServer`	An attribute that specifies the name of another recoverable OrbixOTS server which maintains the transaction log for this server.
`register_xa_rm()`	Registers a XA-compliant resource manager. The parameters specify the XA switch structure, the strings passed to `xa_open()`and `xa_close()` calls and an indication of whether the XA library is thread aware. The return value is an identifier for the registered resource manager.
`recoverable()`	Registers the server as recoverable and specifies an object to be invoked during recovery processing.
`init()`	Initializes the OrbixOTS server and underlying Encina components. The initialization is done based on the values of the five attributes described in the preceding paragraphs. If any of the attributes are used, they must be initialized (non-null and not the empty string) before calling `init()`. Recovery processing is initiated during `init()` and any restart objects registered with the `recoverable()` operation are invoked at this time. The `init()` operation is provided to allow recovery processing to be done before `impl_is_ready()` is called. If `init()` is not called, the initialization and recovery processing are done when `impl_is_ready()` is called.
`get_transaction_factory()`	Used to obtain a reference to the local transaction factory.

`get_lockset_factory()`	Used to obtain a reference to the local lockset factory.
`ConcurrencyMode`	An enumerated type that specifies the concurrency mode servicing requests (passed to `impl_is_ready()`). There are three values: `concurrent` means that all requests are processed concurrently (using a thread pool); `serializeRequests` means that only one request is be serviced at a time; `serializeRequestsAndTransactions` means that only one request is serviced at a time and only one transaction is serviced at a time.
`impl_is_ready()`	Used to indicate that the server is ready to service requests. This operation should be called instead of the operation `CORBA::BOA::impl_is_ready()`. If the `init()` operation was not previously called, then the initialization as described for the `init()` operation is performed. The parameter passed specifies the concurrency mode as described in the section on the `ConcurrencyMode` type.
`shutdown()`	Shuts down the OTS.
`exit()`	Terminates the OrbixOTS server. The status value passed is returned to the execution environment.

There are two types of OrbixOTS servers: recoverable and non-recoverable. A recoverable server is any server that manages its own recoverable data using the `CosTransactions Resource` interface, integrates with an XA-compliant resource manager (RM), or acts as a transaction coordinator. A non-recoverable server just explicitly propagates transaction contexts and object references. A server is made recoverable by calling one or both of the operations `recoverable()` or `register_xa_rm()`. The `recoverable()` operation takes a reference to an object which is invoked when recovery processing is initiated. The operation `register_xa_rm()` is used to register an XA-compliant resource manager.

The five attributes are used to control the transaction log associated with a recoverable server. Setting these attributes must be done before the `init()` method is invoked. Setting an attribute after the `init()` method is called has no effect.

The `serverName` attribute is used to specify the name of the Orbix server (the value that would normally be passed to `CORBA::BOA::impl_is_ready()`. This attribute must be set for persistent servers. If set for automatically activated servers, it must match the name under which the server is registered.

The remaining four attributes are used to provide information about the transaction log used for recoverable servers. When using a local log, the path to the device (which may be an ordinary file or raw disk partition) is specified with the `logDevice` attribute. Information about the log is maintained in two restart files the paths for which are specified in the attributes `restartFile` and `mirrorRestartFile`. The first time a log is used, the `logDevice` attribute must be specified; in subsequent uses this attribute may be omitted. A server may use the transaction log of another OrbixOTS server on the same host. This is done by specifying the name of the server in the `logServer` attribute. If this is done, none of the other three attributes may be specified.

CORBA

```
class Server : public CORBA::IT_PseudoIDL {
public:
   Server();
   virtual ~Server();
   static Server_ptr IT_create(
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
   void serverName(const char* serverName,
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
   char* serverName(
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
   void logDevice(const char* logDevice,
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
   char* logDevice(
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
   void restartFile(const char* restartFile,
      CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
      throw (CORBA::SystemException);
```

```
    char* restartFile(
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    void mirrorRestartFile(const char* mirrorRestartFile,
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    char* mirrorRestartFile(
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    void logServer(const char* logServer,
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    char* logServer(
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    CORBA::Long register_xa_rm(const xa_switch_t* xasw,
           const char* openString,
           const char* closeString,
           const CORBA::Boolean isThreadAware)
       throw (CORBA::SystemException);
    void recoverable(Restart_ptr restart,
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
    void init(
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
       throw (CORBA::SystemException);
    CosTransactions::transactionFactory_ptr
       get_transaction_factory();
    CosConcurrencyControl::LockSetFactory_ptr
       get_lockset_factory();

    enum ConcurrencyMode {
       concurrent,
       serializeRequests,
       serializeRequestsAndTransactions
    };

    void impl_is_ready(const ConcurrencyMode mode =
          serializeRequestsAndTransactions);
    void exit(const long status,
       CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
    };
```

IONA

```
enum TransactionPolicy {
```

```
                 transactionRequired,
                 transactionAllowed,
        };

static void setObjectTransactionPolicy(
                    CORBA::Object_ptr obj,
                  TransactionPolicy policy = transactionRequired);
static void setInterfaceTransactionPolicy(
                    const char *interface,
                  TransactionPolicy policy = transactionRequired);
static TransactionPolicy
            setDefaultTransactionPolicy(TransactionPolicy policy);
static TransactionPolicy
             getDefaultTransactionPolicy();
```

Description The `OrbixOTS::Server` class represents a server application. An instance of this class can be used to initialize and terminate a server application, register resource managers and recovery services with a server, and cause a server to start listening for requests.

Only one instance of the `OrbixOTS::Server` class is permitted per process. Creating more than one instance in a server causes a fatal error.

Constructor `Server();`

This constructor creates a `Server` object.

Destructors `virtual ~Server();`

This destructor destroys a `Server` object and frees any memory that was allocated for it.

Class Members

```
OrbixOTS::Server::ConcurrencyMode enumeration
OrbixOTS::Server::exit();
OrbixOTS::Server::getDefaultTransactionPolicy();
OrbixOTS::Server::get_lockset_factory();
OrbixOTS::Server::get_transaction_factory();
OrbixOTS::Server::impl_is_ready();
OrbixOTS::Server::init();
OrbixOTS::Server::IT_create();
OrbixOTS::Server::logDevice();
OrbixOTS::Server::logServer();
OrbixOTS::Server::mirrorRestartFile();
OrbixOTS::Server::recover();
```

```
OrbixOTS::Server::recoverable();
OrbixOTS::Server::register_xa_rm();
OrbixOTS::Server::restartFile();
OrbixOTS::Server::serverName();
OrbixOTS::Server::setDefaultTransactionPolicy()
OrbixOTS::Server::setInterfaceTransactionPolicy()
OrbixOTS::Server::setObjectTransactionPolicy()
OrbixOTS::Client::shutdown()
```

See Also "OrbixOTS::Client Class" on page 147

OrbixOTS::Server::ConcurrencyMode Enumeration

Synopsis

```
enum ConcurrencyMode {concurrent, serializeRequests,
   serializeRequestsAndTransactions
};
```

Constants

`concurrent`

This indicates that requests and transactions are concurrent and that there is no restriction on server access. A pool of threads is created to handle requests. This is the default concurrency mode.

`serializeRequests`

This indicates that requests are not concurrent but handled one at a time at the server.

`serializeRequestsAndTransactions`

All requests and transactions are serialized so only one request and only one transaction may be processed by a server at any one time. Once a request for a transaction is processed by a server, no requests for other transactions may be processed until the first transaction completes. This mode is only available when used with implicit context propagation (that is, for invocations on objects supporting the TransactionalObject interface.)

Description The `ConcurrencyMode` enumerated type defines the concurrency modes supported by an OrbixOTS server. A value of this type is passed to the operation `impl_is_ready()`. If requests are serialized, only one request is allowed to be executing actively within the server. Requests from other transactions are blocked until the active request returns. The `interface` parameter is the full CORBA interface identifier for the interface. That is, for an interface called "X" the identifier should be of the form "IDL:X:1.0".

See Also "`OrbixOTS::Server::impl_is_ready()`" on page 161

OrbixOTS::Server::shutdown()

Synopsis

```
static void shutdown()
   throw (CORBA::SystemException);
```

Description Shuts down the OTS and rolls back all outstanding transactions.

See Also "`OrbixOTS::Server::exit()`" on page 160

OrbixOTS::Server::exit()

Synopsis

```
void exit(const long status,
   CORBA::Environment& IT_env=CORBA::IT_chooseDefaultEnv()););
```

Parameters The `status` parameter specifies the exit status for the server application.

Description The `exit()` function terminates a server and exits. The server is shut down forcibly, and neither the `OrbixOTS::Server::impl_is_ready()` function called in the main thread nor the `exit()` function returns. Any outstanding transactions are aborted. A value can be specified to indicate the exit status.

See Also "`OrbixOTS::Server::impl_is_ready()`" on page 161

OrbixOTS::Server::getDefaultTransactionPolicy()

Synopsis

```
static TransactionPolicy getDefaultTransactionPolicy();
```

Description The `getDefaultTransactionPolicy()` function gets the current default `TransactionPolicy`.

Returns The current default `TransactionPolicy`.

Notes IONA-specific.

See Also "`OrbixOTS::Server::setDefaultTransactionPolicy()`" on page 168

“OrbixOTS::Server::setInterfaceTransactionPolicy()” on page 168

“OrbixOTS::Server::setObjectTransactionPolicy()” on page 169

OrbixOTS::Server::get_transaction_factory()

Synopsis

```
static CosTransactions::TransactionsFactory_ptr
                                   get_transaction_factory();
```

Description Returns a reference to the local transaction factory.

Returns The current default `TransactionPolicy`.

Notes IONA-specific.

OrbixOTS::Server::get_lockset_factory()

Synopsis

```
static CosConcurrencyControl::LockSetFactory_ptr
                                           get_lockset_factory();
```

Description Returns a reference to the local transaction factory.

Returns The current default `TransactionPolicy`.

Notes IONA-specific.

OrbixOTS::Server::impl_is_ready()

Synopsis

```
void impl_is_ready(CORBA::ULong timeout, const ConcurrencyMode
mode = serializeRequestsAndTransactions)
```

Parameters The parameter `timeout` specifies the timeout in milliseconds that is passed down to the `CORBA.Orbix.impl_is_ready()` call.

Description The `impl_is_ready()` function causes a server to begin accepting requests from clients. This function also initializes OrbixOTS components and XA-compliant resource managers, and it registers exported objects and interfaces with underlying services if initialization was not performed previously with the `OrbixOTS::Server::init()` function.

This is an alternative to `CORBA::Orbix.impl_is_ready()` to prepare an OrbixOTS server to receive invocations.

The concurrency mode to be supported by the server can be specified with the mode parameter. The concurrency mode determines whether transactions are serialized at the server. By default, both are serialized. See the `OrbixOTS::Server::ConcurrencyMode` enumeration type for more information.

The default thread pool size (the number of threads available to handle concurrent requests) is 5. You can override this by specifying a value for the `ENCINA_TPOOL_SIZE` environment variable.

Note that calling the `Server::impl_is_ready()` function is optional. The application can call the Orbix `impl_is_ready()` function directly, but no thread pool is created to handle concurrent requests for thread-aware servers. `Server::impl_is_ready()` automatically calls the Orbix `impl_is_ready()` function. The `Server::impl_is_ready()` function blocks until `CORBA::Orbix.impl_is_ready()` returns. If the `timeout` parameter is specified this `timeout` is passes to `CORBA.Orbix.impl_is_ready()`. Otherwise the default Orbix timeout of `CORBA::Orbix.DEFAULT_TIMEOUT` is passed. This default timeout value can be changed through the configuration variable `OrbixOTS.OTS_ORBIX_LISTEN_TIMEOUT`, specified in milliseconds. This is true for both automatically and persistently launched servers.

Any exceptions thrown by Orbix `impl_is_ready()` are not caught by the `Server::impl_is_ready()` function.

See Also

"`OrbixOTS::Server::ConcurrencyMode Enumeration`" on page 159

"`OrbixOTS::Server::exit()`" on page 160

"`OrbixOTS::Server::init()`" on page 162

OrbixOTS::Server::init()

Synopsis

```
void init(
    CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
    throw (CORBA::SystemException);
```

Description

The `init()` function initializes OrbixOTS components and XA-compliant resource managers. The `init()` function is optional and makes it possible to perform application-specific initialization after the underlying services are initialized but before the server begins listening. After the `init()` function is called, transactions can be created and outgoing transactional requests can be made.

The values of the attributes are used to perform this initialization (only attributes that are non-null and are not the empty string are used). If the server has been registered as recoverable using the `recoverable()` operation, the `recovery()` operation is invoked on the restart object passed to `recoverable()` (only if the reference is non-null).

The standard exception `CORBA::SystemException` may be thrown.

See Also "`OrbixOTS::Server::impl_is_ready()`" on page 161

`Server` class constructor

OrbixOTS::Server::IT_create()

Synopsis

```
static Server_ptr IT_create(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
```

Description Creates an instance of a `Server` pseudo-object. `IT_create()` should be used in preference to the C++ operator `new` but only when there is no (suitable) compliant way to obtain a pseudo-object reference. This will ensure memory management consistency. `IT_create()` returns a pointer to a new `Server` pseudo object.

OrbixOTS::Server::logDevice()

Synopsis

```
void logDevice(const char* logDevice,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
char* logDevice(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description These functions set and get the value of the log device path.

When setting the path, it must be done before `OrbixOTS::Server::init()` is called, and the `logDevice` parameter must refer to an existing ordinary file or raw disk partition which should be of non-zero length (the recommended size is 8Mbytes).

If the `logDevice` attribute was not previously set, a call to get the value returns an empty string.

The standard exception `CORBA::SystemException` may be thrown.

See Also "`OrbixOTS::Server::init()`" on page 162

OrbixOTS::Server::logServer()

Synopsis

```
void logServer(const char* logServer,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
char* logServer(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description Sets or gets the value of the log server name.

When setting the name, it must be done before `OrbixOTS::Server::init()` is called.

If the `logServer` attribute was not previously set, a call to get the value returns an empty string.

The standard exception `CORBA::SystemException` may be thrown.

See Also "`OrbixOTS::Server::init()`" on page 162

OrbixOTS::Server::mirrorRestartFile()

Synopsis

```
void restartFile(const char* restartFile,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
char* restartFile(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description Sets or gets the value of the mirror restart file path attribute of the `Server` pseudo-object.

When setting the mirror restart file path, it must be done before `OrbixOTS::Server::init()` is called. Setting this value requires that the function `restartFile()` must also be called to set the restart file path. For cold starts, the file specified in the path must not exist; for re-starts the file must exist.

If the mirror restart file path attribute was not previously set, a call to get the value returns an empty string.

The standard exception `CORBA::SystemException` may be thrown.

See Also "`OrbixOTS::Server::init()`" on page 162

"`OrbixOTS::Server::restartFile()`" on page 167

OrbixOTS::Server::recoverable()

Synopsis

```
void recoverable(Restart_ptr restart,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv());
```

Parameters The `restart` parameter specifies a reference to a callback object to be called during server restart. It is an instance of a subclass of the `Restart` class.

Description The `recoverable()` function makes a server recoverable by registering the appropriate recovery services. Recovery services must be registered before the server is initialized with either of the functions `OrbixOTS::Server::init()` or `OrbixOTS::Server::impl_is_ready()`.

An instance of a class derived from the `OrbixOTS::Restart` class must be passed as the parameter. The derived class must define a callback function (invoked when the server restarts) that recreates any resources that the server requires or exports.

If the application server calls the `OrbixOTS::Server::register_xa_rm()` function, it does not need to call the `recoverable()` function to make a server recoverable because the server becomes recoverable automatically. See the `OrbixOTS::Server::register_xa_rm()` function for information on registering XA-compliant resource managers.

See Also "`OrbixOTS::Server::init()`" on page 162

"OrbixOTS::Server::impl_is_ready()" on page 161

"OrbixOTS::Server::register_xa_rm()" on page 166

"OrbixOTS::Restart Class" on page 152

OrbixOTS::Server::register_xa_rm()

Synopsis

```
CORBA::Long register_xa_rm(const xa_switch_t* xaSwitch,
      const char* openString,
      const char* closeString,
      const CORBA::Boolean isThreadAware)
   throw (CORBA::SystemException);
```

Parameters

`xaSwitch`	Specifies the XA switch structure used by the resource manager. Refer to the resource manager documentation for more information.
`openString`	Specifies a string of information that is specific to the resource manager and passed in `xa_open()` calls.
`closeString`	Specifies the string containing information specific to the resource manager and passed in `xa_close()` calls.
`isThreadAware`	Specifies whether the resource manager's XA library is thread aware or not. A value of `1` (`true`) means the XA library can handle multiple threads. A value of `0` (`false`) means the XA library cannot handle multiple threads.

Description

The `register_xa_rm()` function registers and opens an XA-compliant resource manager for an OrbixOTS server. The function returns the identifier assigned to the registered resource manager. This also has the effect of registering the server as a recoverable server. An identifier for the resource manager is returned. If used, this function must be called before `OrbixOTS::Server::init()` is called.

The standard exception `CORBA::SystemException` may be thrown.

See Also

"OrbixOTS::Server::init()" on page 162

"OrbixOTS::Server::impl_is_ready()" on page 161

OrbixOTS::Server::restartFile()

Synopsis

```
void restartFile(const char* restartFile,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
char* restartFile(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description

Sets or gets the value of the restart file path attribute of the Server pseudo-object.

For cold starts, the file specified in the path must not exist; for re-starts, the file must exist. Calling this function requires that the function `mirrorRestartFile()` must also be called to set the mirror restart file path. When setting the restart file path, it must be done before `OrbixOTS::Server::init()` is called.

If the `restartFile` attribute was not previously set, a call to get the value returns an empty string.

The standard exception `CORBA::SystemException` may be thrown.

See Also

“`OrbixOTS::Server::init()`” on page 162

“`OrbixOTS::Server::mirrorRestartFile()`” on page 164

OrbixOTS::Server::serverName()

Synopsis

```
void serverName(const char* serverName,
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
char* serverName(
   CORBA::Environment& IT_env = CORBA::IT_chooseDefaultEnv())
   throw (CORBA::SystemException);
```

Description

Sets or gets the value of the server name attribute of the Server pseudo-object.

When setting the server name, it must be done before the function `OrbixOTS::Server::init()` is called. The value used should be the same name that would normally be passed to `CORBA::BOA::impl_is_ready()`. The server name is only necessary for manually launched servers and must be exactly the server name with which the server was registered.

If the serverName attribute was not previously set, a call to get the value returns an empty string.

The standard exception CORBA::SystemException may be thrown.

See Also "OrbixOTS::Server::init()" on page 162

OrbixOTS::Server::setDefaultTransactionPolicy()

Synopsis

```
static TransactionPolicy
            setDefaultTransactionPolicy(TransactionPolicy policy);
```

Parameters The policy parameter specifies the current default TransactionPolicy.

Description The setDefaultTransactionPolicy() function sets the default TransactionPolicy. The default transaction policy is TransactionRequired, in which case both the client and server throw a TRANSACTION_REQUIRED exception if an invocation on a transactional object is outside the scope of a transaction. You may also choose to change the default transaction policy to TransactionAllowed. In this case all transactional objects can process requests outside the scope of a transaction. All newly-created objects take on the default behaviour unless they implement an interface that has a particular policy selected. In the case where the default is changed, TransactionRequired semantics need to be explicitly set for individual interfaces of objects.

Returns The previous TransactionPolicy.

Notes IONA-specific.

See Also "OrbixOTS::Server::getDefaultTransactionPolicy()" on page 160

"OrbixOTS::Server::setInterfaceTransactionPolicy()" on page 168

"OrbixOTS::Server::setObjectTransactionPolicy()" on page 169

OrbixOTS::Server::setInterfaceTransactionPolicy()

Synopsis

```
static void setInterfaceTransactionPolicy(
   const char *interface,
      TransactionPolicy policy = transactionRequired);
```

Parameters

`interface`	The interface to treat as transactional.
`policy`	The `TransactionPolicy` for this transactional interface.

Description The `setInterfaceTransactionPolicy()` function marks an interface as transactional and specifies the transaction policy for this transactional interface. Objects that support this interface are treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface. The `interface` parameter is the CORBA repository identifier for the interface that is of the form “IDL:X:1.0”.

Notes IONA-specific.

See Also “`OrbixOTS::Server::getDefaultTransactionPolicy()`” on page 160

“`OrbixOTS::Server::setDefaultTransactionPolicy()`” on page 168

“`OrbixOTS::Server::setObjectTransactionPolicy()`” on page 169

OrbixOTS::Server::setObjectTransactionPolicy()

Synopsis

```
static void setObjectTransactionPolicy(
   CORBA::Object_ptr obj,
      TransactionPolicy policy = transactionRequired);
```

Parameters

`obj`	The object to treat as transactional.
`policy`	The `TransactionPolicy` for this transactional object.

Description The `setObjectTransactionPolicy()` function marks an object as transactional and specifies the transaction policy for this transactional object. This object is treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface.

Notes IONA-specific.

See Also “`OrbixOTS::Server::getDefaultTransactionPolicy()`” on page 160

"`OrbixOTS::Server::setInterfaceTransactionPolicy()`" on page 168
"`OrbixOTS::Server::setInterfaceTransactionPolicy()`" on page 168

11

CosTransactions Module

This chapter describes the C++ class implementations of the CosTransactions interfaces. The implementation of OrbixOTS supports the full OMG specification of the CosTransactions module. This contains interfaces that include support for defining transactional interfaces and recoverable resources.

Introduction

The Object Management Group's Object Transaction Service (OMG OTS) defines interfaces that integrate transactions into the distributed object paradigm. The OMG OTS interface allows developers to manage transactions under two different models of transaction propagation: implicit and explicit:

- In the implicit model, the transaction context is associated with the client thread; when client requests are made on transactional objects, the transaction context associated with the thread is propagated to the object implicitly.
- In the explicit model, the transaction context must be passed explicitly when client requests are made on transactional objects in order to propagate the transaction context to the object.

Java implementations for the `CosTransactions` interfaces are described in Chapter 13, "Java Classes".

Keep the following in mind:

- Applications must include the header file `OrbixOTS.hh`.
- All of the OTS classes are nested within the `CosTransactions` class. Therefore, you must prefix `CosTransactions` to the OTS class and function names when using them in your application.
- All of the OTS class functions define one additional final parameter of the type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.
- All of the OTS class functions can throw the `CORBA::SystemException` exception if an object request broker (ORB) error occurs.

Overview of Classes

The OMG OTS classes provide the following functionality:

- Defining transactional interfaces in the CORBA environment:

  `TransactionalObject`
- Managing transactions under the implicit model:

  `Current`
- Managing transactions under the explicit model:

  `TransactionFactory`
  `Control`
  `Coordinator`
  `Terminator`
- Managing recoverable resources in the CORBA environment:

  `RecoveryCoordinator`
  `Resource`
  `SubtransactionAwareResource`
  `Synchronization`
- Reporting system errors:

  `HeuristicCommit`
  `HeuristicHazard`
  `HeuristicMixed`
  `HeuristicRollback`

```
Inactive
InvalidControl
INVALID_TRANSACTION
NoTransaction
NotPrepared
NotSubtransaction
SubtransactionsUnavailable
TRANSACTION_REQUIRED
TRANSACTION_ROLLEDBACK
Unavailable
```

General Data Types

OrbixOTS defines enumerated data types to represent the status of a transaction object during two-phase commit and to indicate a participant's vote on the outcome of a transaction.

Status Enumeration Type

Synopsis

```
enum Status{
   StatusActive,
   StatusMarkedRollback,
   StatusPrepared,
   StatusCommitted,
   StatusRolledBack,
   StatusUnknown,
   StatusNoTransaction,
   StatusPreparing,
   StatusCommitting,
   StatusRollingBack
};
```

Description The `Status` enumerated type defines values that are used to indicate the status of a transaction. Status values are used in both the implicit and explicit models of transaction demarcation defined by the Object Transaction Service (OTS). The `Current::get_status()` function can be called to return the transaction status if the implicit model is used. The `Coordinator::get_status()` function can be called to return the transaction status if the explicit model is used.

Constants

`StatusActive`	Indicates that processing of a transaction is still in progress.
`StatusMarkedRollback`	Indicates that a transaction is marked to be rolled back.
`StatusPrepared`	Indicates that a transaction has been prepared but not completed.
`StatusCommitted`	Indicates that a transaction has been committed and the effects of the transaction have been made permanent.
`StatusRolledBack`	Indicates that a transaction has been rolled back.
`StatusUnknown`	Indicates that the status of a transaction is unknown.
`StatusNoTransaction`	Indicates that a transaction does not exist in the current transaction context.
`StatusPreparing`	Indicates that a transaction is preparing to commit.
`StatusCommitting`	Indicates that a transaction is in the process of committing.
`StatusRollingBack`	Indicates that a transaction is in the process of rolling back.

See Also

"`Coordinator::get_status()`" on page 185

"`Current::get_status()`" on page 197

Vote Enumeration Type

Synopsis

```
enum Vote{
   VoteCommit,
   VoteRollback,
   VoteReadOnly
};
```

Constants

`VoteCommit`	Specifies the value used to indicate a vote to commit a transaction.

`VoteRollback`	Specifies the value used to indicate a vote to abort (rollback) a transaction.
`VoteReadOnly`	Specifies the value used to indicate no vote on the outcome of a transaction.

Description The `Vote` enumerated type defines values for the voting status of transaction participants. The participants in a transaction each vote on the outcome of a transaction during the two-phase commit process. In the prepare phase, a `Resource` object can vote whether to commit or abort a transaction. If a `Resource` has not modified any data as part of the transaction, it can vote `VoteReadOnly` to indicate that its participation does not affect the outcome of the transaction.

See Also “`CosTransactions::Resource Class`” on page 202

General Exceptions

Errors are handled in OrbixOTS by using exceptions. Exceptions provide a way of returning error information back through multiple levels of procedure or function calls, propagating this information until a function or procedure is reached that can respond appropriately to the error.

Each of the following exceptions are implemented as classes. The exceptions are shown here in two tables: one for the OrbixOTS exceptions and another for the system exceptions.

Exception	Description
`HeuristicCommit`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that all updates have been committed. See Also: `Resource class`

Table 11.1: *OrbixOTS Exceptions*

Exception	Description

`HeuristicHazard`	This exception is thrown to report that a heuristic decision has possibly been made by one or more participants in a transaction and the outcome of all participants in the transaction is unknown. See Also:
	`Current::commit()`
	`Resource class`
	`Terminator::commit()`

`HeuristicMixed`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that some updates have been committed and others rolled back. See Also:
	`Current::commit()`
	`Resource class`
	`Terminator::commit()`

`HeuristicRollback`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that all updates have been rolled back. See Also: `Resource class`

`Inactive`	This exception is thrown when a transactional operation is requested for a transaction, but that transaction is already prepared. See Also:
	`Coordinator::create_subtransaction()`
	`Coordinator::register_resource()`
	`Coordinator::register_subtran_aware()`
	`Coordinator::rollback_only()`

`InvalidControl`	This exception is thrown when an invalid `Control` object is used in an attempt to resume a suspended transaction. See Also:
	`Control class`
	`Current::resume()`

Table 11.1: *OrbixOTS Exceptions*

Exception	Description

`NotPrepared`	This exception is thrown when an operation (such as a `commit()`) is requested for a resource, but that resource is not prepared. See Also:
	`RecoveryCoordinator::replay_completion()`
	`Resource class`

`NoTransaction`	This exception is thrown when an operation is requested for the current transaction, but no transaction is associated with the client thread. See Also:
	`Current::commit()`
	`Current::rollback()`
	`Current::rollback_only()`

`NotSubtransaction`	This exception is thrown when an operation that requires a subtransaction is requested for a transaction that is not a subtransaction. See Also:
	`Control::get_parent()`
	`Coordinator::register_subtran_aware()`

`SubtransactionsUnavailable`	This exception is thrown when an attempt is made to create a subtransaction, but the parent transaction is already prepared. See Also:
	`Coordinator::create_subtransaction()`
	`Current::begin()`

`Unavailable`	This exception is thrown when a Terminator or `Coordinator` object cannot be provided by a `Control` object due to environment restrictions. See Also:
	`Control::get_coordinator()`
	`Control::get_terminator()`

Table 11.1: *OrbixOTS Exceptions*

The following table shows the system exceptions that can be thrown:

Exception	Description
`INVALID_TRANSACTION`	This exception is thrown when the transaction context is invalid for a request.
`TRANSACTION_REQUIRED`	This exception is thrown when a null transaction context is associated with the client thread, and a transactional operation is requested.

`TRANSACTION_ROLLEDBACK`	This exception is thrown when a transactional operation (such as a `commit()`) is requested for a transaction that has been rolled back or marked for rollback. See Also:
	`Current::commit()`
	`Terminator::commit()`

Table 11.2: *System Exceptions*

CosTransactions::Control Class

Synopsis

```
class Control {
public:
   Terminator_ptr get_terminator();
   Coordinator_ptr get_coordinator();
   CORBA::Long id();
   Control_ptr get_parent();
   Control_ptr get_top_level();
};
typedef Control *Control_ptr;
class Control_var;
```

Description

The `Control` class enables explicit control of a factory-created transaction; the factory creates a transaction and returns a `Control` instance associated with the transaction. The `Control` object provides access to the `Coordinator` and `Terminator` objects used to manage and complete the transaction.

A `Control` object can be used to propagate a transaction context explicitly. By passing a `Control` object as an argument in a request, the transaction context can be propagated. The `TransactionFactory::create()` function can be used to create a transaction and return the `Control` object associated with it. This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Control`.

A `Control_ptr` type holds a reference to a `Control` object. OrbixOTS also provides a `Control_var` helper class. Both the `Control_ptr` and `Control_var` types hold and manage a reference to a `Control` object.

Class Members

```
Control::get_coordinator()
Control::get_parent()
Control::get_terminator()
Control::get_top_level
Control::id()
```

See Also

"`CosTransactions::Coordinator Class`" on page 182

"`Current::get_control()`" on page 197

"`Coordinator::get_status()`" on page 185

"`CosTransactions::Terminator Class`" on page 209

"`TransactionFactory::create()`" on page 212

"`NoTransaction`" on page 177

"`NotSubtransaction`" on page 177

Control::get_coordinator()

Synopsis

```
Coordinator_ptr get_coordinator()
   throw(CORBA::SystemException, Unavailable);
```

Description

The `get_coordinator()` member function returns the `Coordinator` object for the transaction with which the `Control` object is associated. The returned `Coordinator` object can be used to determine the status of the transaction, the relationship between the associated transaction and other transactions, to create subtransactions, and so on.

The `get_coordinator()` function throws the `Unavailable` exception if the Coordinator associated with the `Control` object is not available.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Control::get_coordinator()`.

See Also "CosTransactions::Coordinator Class" on page 182

"Unavailable" on page 177

Control::get_parent()

Synopsis

```
Control_ptr get_parent()
   throw(CORBA::SystemException, NotSubtransaction);
```

Description The `get_parent()` member function returns the `Control` object for the parent of the transaction with which the `Control` object is associated. If the associated transaction is not a subtransaction, the `NotSubtransaction` exception is thrown.

Notes This function is specific to OrbixOTS and is not a standard CORBA function.

The `Control` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Control::get_parent()`.

See Also "NotSubtransaction" on page 177

Control::get_terminator()

Synopsis

```
Terminator_ptr get_terminator()
   throw(CORBA::SystemException, Unavailable);
```

Description The `get_terminator()` member function returns the Terminator object for the transaction with which the `Control` object is associated. The returned Terminator object can be used to either commit or roll back the transaction.

The `get_terminator()` function throws the `Unavailable` exception if the Terminator associated with the `Control` object is not available.

The `Terminator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Control::get_terminator()`.

See Also "CosTransactions::Terminator Class" on page 209

"Unavailable" on page 177

Control::get_top_level()

Synopsis

```
Control_ptr get_top_level()
   throw(CORBA::SystemException, NotSubtransaction);
```

Description The `get_top_level()` member function returns the `Control` object for the top-level ancestor of the transaction with which the `Control` object is associated. If the associated transaction is not a subtransaction, the `NotSubtransaction` exception is thrown.

Notes This function is specific to OrbixOTS and is not a standard CORBA function.

The `Control` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Control::get_top_level()`.

See Also "`NotSubtransaction`" on page 177

Control::id()

Synopsis

```
CORBA::Long id()
   throw(CORBA::SystemException);
```

Description The `id()` member function returns the transaction identifier for the transaction with which the `Control` object is associated.

Notes This function is specific to OrbixOTS and is not a standard CORBA function.

The `id()` function is an OrbixOTS extension to the OMG OTS interface. The return value can be used to display the identity of the transaction associated with the `Control` object.

The `Control` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Control::id()`.

See Also "`CosTransactions::Control Class`" on page 178

CosTransactions::Coordinator Class

Synopsis

```
class Coordinator {
public:
   CORBA::Long id();
   char *get_transaction_name();
   Status get_status();
   Status get_parent_status();
   Status get_top_level_status();
   CORBA::Boolean is_same_transaction(Coordinator_ptr);
   CORBA::Boolean is_related_transaction(Coordinator_ptr);
   CORBA::Boolean is_ancestor_transaction(Coordinator_ptr);
   CORBA::Boolean is_descendant_transaction(Coordinator_ptr);
   CORBA::Boolean is_top_level_transaction();
   unsigned long hash_transaction();
   unsigned long hash_top_level_tran();
   RecoveryCoordinator register_resource(Resource);
   void register_subtran_aware(SubtransactionAwareResource);
   Control_ptr create_subtransaction();
   void rollback_only();
   PropagationContext* get_txcontext()
};
typedef Coordinator *Coordinator_ptr;
class Coordinator_var;
```

Description

The `Coordinator` class enables explicit control of a factory-created transaction; the factory creates a transaction and returns a `Control` instance associated with the transaction. The `Control::get_coordinator()` function returns the `Coordinator` object used to manage the transaction.

The operations defined by the `Coordinator` class can be used by the participants in a transaction to determine the status of the transaction, determine the relationship of the transaction to other transactions, mark the transaction for rollback, and create subtransactions.

The `Coordinator` class also defines operations for registering resources as participants in a transaction and registering subtransaction-aware resources with a subtransaction.

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Coordinator`.

A `Coordinator_ptr` type holds a reference to a `Coordinator` object. OrbixOTS also provides a `Coordinator_var` helper class. Both the `Coordinator_ptr` and `Coordinator_var` types hold and manage a reference to a `Coordinator` object.

Class Members

```
Coordinator::create_subtransaction()
Coordinator::get_parent_status()
Coordinator::get_status()
Coordinator::get_top_level_status()
Coordinator::get_transaction_name()
Coordinator::get_txcontext()
Coordinator::hash_top_level_tran()
Coordinator::hash_transaction()
Coordinator::is_ancestor_transaction()
Coordinator::is_descendant_transaction()
Coordinator::is_related_transaction()
Coordinator::is_same_transaction()
Coordinator::is_top_level_transaction()
Coordinator::register_resource()
Coordinator::register_subtran_aware()
Coordinator::rollback_only()
```

See Also

"`CosTransactions::Control Class`" on page 178

"`Control::get_coordinator()`" on page 179

"`CosTransactions::Terminator Class`" on page 209

Coordinator::create_subtransaction()

Synopsis

```
Control_ptr create_subtransaction()
   throw(CORBA::SystemException, Inactive,
SubtransactionUnavailable);
```

Description

The `create_subtransaction()` member function creates a new subtransaction for the transaction associated with the `Coordinator` object. A subtransaction is one that is embedded within another transaction; the transaction within which the subtransaction is embedded is referred to as its parent. A transaction that has no parent is a top-level transaction. A subtransaction executes within the scope of its parent transaction and can be used to isolate failures; if a subtransaction fails, only the subtransaction is rolled back. If a subtransaction commits, the effects of the commit are not permanent until the parent transaction commits. If the parent transaction rolls back, the subtransaction is also rolled back.

If the parent transaction is already rolled back when `create_subtransaction()` is called, the `SubtransactionsUnavailable` exception is thrown.

The `create_subtransaction()` function throws the `Inactive` exception if the transaction is already prepared.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::create_subtransaction()`.

Return Values The `create_subtransaction()` function returns the `Control` object associated with the new subtransaction.

See Also "`CosTransactions::Control Class`" on page 178

"`Inactive`" on page 176

"`SubtransactionsUnavailable`" on page 177

Coordinator::get_parent_status()

Synopsis

```
Status get_parent_status()
   throw(CORBA::SystemException);
```

Description The `get_parent_status()` member function returns the status of the parent of the transaction associated with the `Coordinator` object. For more information, see the reference page for the function `Coordinator::create_subtransaction()`.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::get_parent_status()`.

Return Values The status returned indicates which phase of processing the transaction is in. See the reference page for the `Status` type for information about the possible status values. If the transaction associated with the `Coordinator` object is a subtransaction, the status of its parent transaction is returned. If there is no parent transaction, the status of the transaction associated with the `Coordinator` object itself is returned.

See Also “`Coordinator::create_subtransaction()`” on page 183

“`Coordinator::get_status()`” on page 185

“`Coordinator::get_top_level_status()`” on page 185

“`Status Enumeration Type`” on page 173

Coordinator::get_status()

Synopsis

```
Status get_status()
   throw(CORBA::SystemException);
```

Description The `get_status()` member function returns the status of the transaction associated with the `Coordinator` object. The status returned indicates which phase of processing the transaction is in. See the reference page for the Status type for information about the possible status values.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::get_status()`.

See Also “`Coordinator::get_parent_status()`” on page 184

“`Coordinator::get_top_level_status()`” on page 185

“`Status Enumeration Type`” on page 173

Coordinator::get_top_level_status()

Synopsis

```
Status get_top_level_status()
   throw(CORBA::SystemException);
```

Description The `get_top_level_status()` member function returns the status of the top-level ancestor of the transaction associated with the `Coordinator` object. See the reference page for the `Coordinator::create_subtransaction()` function for more information.

The status returned indicates which phase of processing the transaction is in. See the reference page for the `Status` type for information about the possible status values. If the transaction associated with the `Coordinator` object is the top-level transaction, its status is returned.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::get_top_level_status()`.

See Also "`Coordinator::create_subtransaction()`" on page 183

"`Coordinator::get_status()`" on page 185

"`Coordinator::get_parent_status()`" on page 184

"`Status Enumeration Type`" on page 173

Coordinator::get_transaction_name()

Synopsis

```
char *get_transaction_name();
```

Description The `get_transaction_name()` member function returns the name of the transaction associated with the `Coordinator` object.

Notes The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::get_transaction_name()`.

Coordinator::get_txcontext()

Synopsis

```
PropagationContext* Coordinator::get_txcontext()
   throw (CORBA::SystemException, Unavailable);
```

Description Returns the propagation context object which is used to export the current transaction to a new transaction service domain. The exception `Unavailable` is raised if the propagation context is unavailable.

See Also "`Unavailable`" on page 177

"`TransactionFactory::recreate()`" on page 213

Coordinator::hash_top_level_tran()

Synopsis

```
unsigned long hash_top_level_tran()
   throw(CORBA::SystemException);
```

Description

The `hash_top_level_tran()` member function returns a hash code for the top-level ancestor of the transaction associated with the `Coordinator` object. If the transaction associated with the `Coordinator` object is the top-level transaction, its hash code is returned. See the reference page for the `Coordinator::create_subtransaction()` function for more information. The returned hash code is typically used as an index into a table of `Coordinator` objects. The low-order bits of the hash code can be used to hash into a table with a size that is a power of two.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::hash_top_level_tran()`.

See Also

"`Coordinator::create_subtransaction()`" on page 183

"`Coordinator::hash_transaction()`" on page 187

Coordinator::hash_transaction()

Synopsis

```
unsigned long hash_transaction()
   throw(CORBA::SystemException);
```

Description

The `hash_transaction()` member function returns a hash code for the transaction associated with the `Coordinator` object.

The returned hash code is typically used as an index into a table of `Coordinator` objects. The low-order bits of the hash code can be used to hash into a table with a size that is a power of two.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::hash_transaction()`.

See Also

"`Coordinator::hash_top_level_tran()`" on page 187

Coordinator::is_ancestor_transaction()

Synopsis

```
CORBA::Boolean is_ancestor_transaction(
   Coordinator_ptr tc)
   throw(CORBA::SystemException);
```

Parameters The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description The `is_ancestor_transaction()` member function determines whether the transaction associated with the `Coordinator` object is an ancestor of the transaction associated with the coordinator specified in the `tc` parameter. See the reference page for the `Coordinator::create_subtransaction()` function for more information.

The `is_ancestor_transaction()` function returns a value of type `CORBA::Boolean`.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::is_ancestor_transaction()`.

Return Values The `is_ancestor_transaction()` function returns true if the transaction is an ancestor or if the two transactions are the same; otherwise, the function returns false.

See Also

"`Coordinator::is_descendant_transaction()`" on page 188

"`Coordinator::is_related_transaction()`" on page 189

"`Coordinator::is_same_transaction()`" on page 190

"`Coordinator::create_subtransaction()`" on page 183

Coordinator::is_descendant_transaction()

Synopsis

```
CORBA::Boolean is_descendant_transaction(Coordinator_ptr tc)
   throw(CORBA::SystemException);
```

Parameters The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description The is_descendant_transaction() member function determines whether the transaction associated with the Coordinator object is a descendant of the transaction associated with the coordinator specified in the tc parameter. See the reference page for the Coordinator::create_subtransaction() function for more information.

The is_descendant_transaction() function returns a value of type CORBA::Boolean.

The Coordinator class is nested within the CosTransactions class. The full name is CosTransactions::Coordinator::is_descendant_transaction().

Return Values The is_descendant_transaction() function returns true if the transaction is a descendant or if the two transactions are the same; otherwise, the function returns false.

See Also

"Coordinator::is_descendant_transaction()" on page 188

"Coordinator::is_related_transaction()" on page 189

"Coordinator::is_same_transaction()" on page 190

"Coordinator::is_top_level_transaction()" on page 191

"Coordinator::create_subtransaction()" on page 183

Coordinator::is_related_transaction()

Synopsis

```
CORBA::Boolean is_related_transaction(Coordinator_ptr tc)
   throw(CORBA::SystemException);
```

Parameters The tc parameter specifies the coordinator of another transaction to compare with the Coordinator object.

Description The is_related_transaction() member function determines whether the transaction associated with the Coordinator object and the transaction associated with the coordinator specified in the tc parameter have a common ancestor. See the reference page for the Coordinator::create_subtransaction() function for more information.

The is_related_transaction() function returns a value of type CORBA::Boolean.

The Coordinator class is nested within the CosTransactions class. The full name for the function is CosTransactions::Coordinator::is_related_transaction().

Return Values The is_related_transaction() function returns true if both transactions are descendants of the same transaction; otherwise, the function returns false.

See Also "Coordinator::is_descendant_transaction()" on page 188

"Coordinator::is_ancestor_transaction()" on page 188

"Coordinator::is_same_transaction()" on page 190

"Coordinator::is_top_level_transaction()" on page 191

"Coordinator::create_subtransaction()" on page 183

Coordinator::is_same_transaction()

Synopsis

```
CORBA::Boolean is_same_transaction(
   Coordinator_ptr tc)
   throw(CORBA::SystemException);
```

Parameters The tc parameter specifies the coordinator of another transaction to compare with the Coordinator object.

Description The is_same_transaction() member function determines whether the transaction associated with the Coordinator object and the transaction associated with the coordinator specified in the tc parameter are the same transaction.

The is_same_transaction() function returns a value of type CORBA::Boolean.

The Coordinator class is nested within the CosTransactions class. The full name for the function is CosTransactions::Coordinator::is_same_transaction().

Return Values The is_same_transaction() function returns true if the transactions associated with the two Coordinator objects are the same transaction; otherwise, the function returns false.

See Also "Coordinator::is_descendant_transaction()" on page 188

"Coordinator::is_related_transaction()" on page 189

"Coordinator::is_ancestor_transaction()" on page 188

"`Coordinator::is_top_level_transaction()`" on page 191

Coordinator::is_top_level_transaction()

Synopsis

```
CORBA::Boolean is_top_level_transaction()
   throw(CORBA::SystemException);
```

Description

The `is_top_level_transaction()` member function determines whether the transaction associated with a `Coordinator` object is a top-level transaction. See the reference page for the `Coordinator::create_subtransaction()` function for more information.

The `is_top_level_transaction()` function returns a value of type `CORBA::Boolean`.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::is_top_level_transaction()`.

Return Values The `is_top_level_transaction()` function returns true if the transaction is a top-level transaction; otherwise, the function returns false.

See Also

"`Coordinator::is_descendant_transaction()`" on page 188

"`Coordinator::is_related_transaction()`" on page 189

"`Coordinator::is_same_transaction()`" on page 190

"`Coordinator::is_ancestor_transaction()`" on page 188

"`Coordinator::create_subtransaction()`" on page 183

Coordinator::register_resource()

Synopsis

```
RecoveryCoordinator register_resource(
   Resource resource)
   throw(CORBA::SystemException, Inactive);
```

Parameters The `resource` parameter specifies the resource to register as a participant.

Description The `register_resource()` member function registers a specified resource as a participant in the transaction associated with a `Coordinator` object. When the transaction ends, the registered resource must commit or roll back changes made as part of the transaction. Only server applications can register resources. See the reference page for the `Resource` class for more information.

The `register_resource()` function throws the `Inactive` exception if the transaction is prepared. It throws the `CORBA::TRANSACTION_ROLLEDBACK` exception if the transaction is marked for rollback only.

Return Values The `register_resource()` function returns a `RecoveryCoordinator` object that the registered Resource object can use during recovery.

Notes The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::register_resource()`.

See Also "`Inactive`" on page 176

"`CosTransactions::RecoveryCoordinator Class`" on page 201

"`CosTransactions::Resource Class`" on page 202

Coordinator::register_subtran_aware()

Synopsis

```
void register_subtran_aware(
   SubtransactionAwareResource resource)
   throw(CORBA::SystemException, NotSubtransaction, Inactive);
```

Parameters The `resource` parameter specifies the resource to register.

Description The `register_subtran_aware()` member function registers a specified resource with the subtransaction associated with a `Coordinator` object. The resource is registered with the subtransaction only, not as a participant in the top-level transaction. (The `Coordinator::register_resource()` function can be used to register the resource as a participant in the top-level transaction.) Only server applications can register resources.

When the transaction ends, the registered resource must commit or roll back changes made as part of the subtransaction. See the reference page for the `SubtransactionAwareResource` class for more information.

The register_subtran_aware() function throws the NotSubtransaction exception if the transaction associated with the Coordinator object is not a subtransaction. It throws the Inactive exception if the subtransaction or any ancestor of the subtransaction has ended. It throws the CORBA::TRANSACTION_ROLLEDBACK exception if the transaction is marked for rollback only.

Notes The Coordinator class is nested within the CosTransactions class. The full name for the function is CosTransactions::Coordinator::register_subtran_aware().

See Also "Inactive" on page 176

"CosTransactions::RecoveryCoordinator Class" on page 201

"CosTransactions::SubtransactionAwareResource Class" on page 205

Coordinator::register_synchronization()

Synopsis

```
void register_synchronization(Synchronization sync);
   throw(CORBA::SystemException, Inactive);
```

Parameters The sync parameter specifies the synchronization object to register.

Description The register_synchronization() member function registers a specified synchronization object for the transaction associated with a Coordinator object. See the reference page for the Synchronization class for more information.

The register_synchronization() function throws the Inactive exception if the transaction is already prepared. It throws the CORBA::TRANSACTION_ROLLEDBACK exception if the transaction is marked for rollback only.

Notes The Coordinator class is nested within the CosTransactions class. The full name for the function is CosTransactions::Coordinator::register_synchronization().

See Also "Inactive" on page 176

"CosTransactions::RecoveryCoordinator Class" on page 201

"CosTransactions::Synchronization Class" on page 207

Coordinator::rollback_only()

Synopsis

```
void rollback_only()
   throw(CORBA::SystemException, Inactive);
```

Description The `rollback_only()` member function marks the transaction associated with the `Coordinator` object so that the only possible outcome for the transaction is to roll back. The transaction is not rolled back until the participant that created the transaction either commits or aborts the transaction.

OrbixOTS allows the `Terminator::rollback()` function to be called instead of `rollback_only()`. Calling `Terminator::rollback()` rolls back the transaction immediately, preventing unnecessary work from being done between the time the transaction is marked for rollback and the time the transaction is actually rolled back.

The `rollback_only()` function throws the `Inactive` exception if the transaction is already prepared.

The `Coordinator` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Coordinator::rollback_only()`.

See Also "`Inactive`" on page 176

"`Terminator::rollback()`" on page 210

CosTransactions::Current Class

Synopsis

```
class Current {
public:
   static Current_ptr IT_Create();
   static void begin();
   static void commit(CORBA::Boolean);
   static void rollback();
   static void rollback_only();
   static Status get_status();
   static char *get_transaction_name();
   static void set_timeout(unsigned long);
   static Control_ptr get_control();
   static Control_ptr suspend();
   static void resume(Control_ptr);
};
typedef Current *Current_ptr;
```

```
class Current_var;
```

Description The `Current` class represents a transaction that is associated with the calling thread; the thread defines the transaction context. Transaction context is propagated implicitly when the client issues requests.

This class defines member functions for beginning, committing, and aborting a transaction using the implicit model of transaction control. It also defines member functions for suspending and resuming a transaction and retrieving information about a transaction.

A `Current_ptr` type holds a reference to a `Current` object.

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Current`.

The `Current` class conforms to the Orbix approach for defining pseudo objects. The class provides a static function called `IT_create()` that can be used to create `Current` objects instead of using the C++ new function. The class also provides static functions for duplicating and releasing object references called `_duplicate()` and `_release()`.

OrbixOTS also provides a `Current_var` helper class. Both the `Current_ptr` and `Current_var` types hold and manage a reference to a `Current` object. Refer to the Orbix documentation for more information on the use of pseudo objects and object reference types.

Class Members

```
Current::begin()
Current::commit()
Current::get_control()
Current::get_status()
Current::get_transaction_name()
Current::IT_Create()
Current::resume()
Current::rollback()
Current::rollback_only()
Current::set_timeout()
Current::suspend()
```

See Also "`CosTransactions::Control Class`" on page 178

"`Status Enumeration Type`" on page 173

Current::begin()

Synopsis

```
static void begin()
   throw(CORBA::SystemException,SubtransactionsUnavailable);
```

Description The `begin()` member function creates a new transaction and modifies the transaction context of the calling thread to associate the thread with the new transaction. If a parent transaction is associated with the calling thread but is already rolled back, the `SubtransactionsUnavailable` exception is thrown.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::begin()`.

See Also

"`Current::commit()`" on page 196

"`Current::rollback()`" on page 199

"`Current::rollback_only()`" on page 199

"`SubtransactionsUnavailable`" on page 177

Current::commit()

Synopsis

```
static void commit( CORBA::Boolean report_heuristics)
   throw(CORBA::SystemException,
         NoTransaction,
         HeuristicHazard,
         TRANSACTION_ROLLEDBACK);
```

Parameters The `report_heuristics` parameter specifies whether heuristic decisions should be reported for the transaction associated with the calling thread.

Description The `commit()` member function attempts to commit the transaction associated with the calling thread.

If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown. If the `report_heuristics` parameter is true, the `HeuristicMixed` exception is thrown when a heuristic decision has caused inconsistent outcomes and the `HeuristicHazard` exception is thrown when a heuristic decision has possibly caused inconsistent outcomes.

If all the transaction participants do not commit, the `CORBA::TRANSACTION_ROLLEDBACK` system exception is thrown.

The `commit()` function takes a value of type `CORBA::Boolean` as its first argument.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::commit()`.

See Also

"`Current::begin()`" on page 196

"`Current::rollback()`" on page 199

"`Current::rollback_only()`" on page 199

"`HeuristicHazard`" on page 176

"`NoTransaction`" on page 177

"`TRANSACTION_ROLLEDBACK`" on page 178

Current::get_control()

Synopsis

```
static Control_ptr get_control()
   throw(CORBA::SystemException);
```

Description

The `get_control()` member function returns the `Control` object for the transaction associated with the calling thread. If no transaction is associated with the calling thread, a null object reference is returned.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::get_control()`.

See Also

"`Current::resume()`" on page 198

Current::get_status()

Synopsis

```
static Status get_status()
   throw(CORBA::SystemException);
```

Description

The `get_status()` member function returns the status of the transaction associated with the calling thread. If no transaction is associated with the calling thread, the `StatusNoTransaction` value is returned.

The status returned indicates the processing phase of the transaction. See the reference page for the `Status` type for information about the possible status values.

The Current class is nested within the CosTransactions class. The full name for the function is CosTransactions::Current::get_status().

See Also "Status Enumeration Type" on page 173

Current::get_transaction_name()

Synopsis

```
static char *get_transaction_name();
```

Description The get_transaction_name() member function returns the name of the transaction associated with the calling thread. If no transaction is associated with the calling thread, a null string is returned.

See Also "CosTransactions::Current Class" on page 194

Current::IT_Create()

Synopsis

```
static Current_ptr IT_create()
```

Description Creates an instance of a Current pseudo-object. It is recommended that IT_create() should be used in preference to the C++ operator new but only when there is no (suitable) compliant way to obtain a pseudo-object reference. Use of IT_create() in preference to new will ensure memory management consistency.

See Also "CosTransactions::Current Class" on page 194

Current::resume()

Synopsis

```
static void resume( Control_ptr which)
   throw(CORBA::SystemException, InvalidControl);
```

Parameters The which parameter specifies a Control object that represents the transaction context associated with the calling thread.

Description The resume() member function resumes the suspended transaction identified by the which parameter and associated with the calling thread. If the value of the which parameter is a null object reference, the calling thread disassociates from the transaction. If a non-null parameter is invalid, the InvalidControl exception is thrown.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::resume()`.

See Also "`CosTransactions::Current Class`" on page 194

"`Current::get_control()`" on page 197

"`Current::suspend()`" on page 201

"`InvalidControl`" on page 176

Current::rollback()

Synopsis

```
static void rollback()
   throw(CORBA::SystemException, NoTransaction);
```

Description The `rollback()` member function rolls back the transaction associated with the calling thread. If the transaction was started with the `Current::begin()` function, the transaction context for the thread is restored to its state before the transaction was started; otherwise, the transaction context is set to null.

If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::rollback()`.

See Also "`CosTransactions::Current Class`" on page 194

"`Current::begin()`" on page 196

"`Current::rollback_only()`" on page 199

"`NoTransaction`" on page 177

Current::rollback_only()

Synopsis

```
static void rollback_only()
   throw(CORBA::SystemException, NoTransaction);
```

Description The `rollback_only()` member function marks the transaction associated with the calling thread for rollback. The transaction is modified so that the only possible outcome is to roll back the transaction. Any participant in the transaction can mark the transaction for rollback. The transaction is not rolled back until the participant that created the transaction either commits or aborts the transaction.

OrbixOTS allows the `Current::rollback()` function to be called instead of `rollback_only()`. Calling `Current::rollback()` rolls back the transaction immediately, preventing unnecessary work from being done between the time the transaction is marked for rollback and the time the transaction is actually rolled back.

If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::rollback_only()`.

See Also

"`CosTransactions::Current Class`" on page 194

"`Current::rollback()`" on page 199

"`NoTransaction`" on page 177

Current::set_timeout()

Synopsis

```
static void set_timeout( unsigned long seconds)
   throw(CORBA::SystemException);
```

Parameters

The `seconds` parameter specifies the number of seconds that the transaction waits for completion before rolling back.

Description

The `set_timeout()` member function sets a timeout period for the transaction associated with the calling thread. The timeout affects only those transactions begun with the `Current::begin()` function after the timeout is set. The seconds parameter sets the number of seconds from the time the transaction is begun that it waits for completion before being rolled back; if the seconds parameter is zero, no timeout is set for the transaction.

The `Current` class is nested within the `CosTransactions` class. The full name for the function is `CosTransactions::Current::set_timeout()`.

See Also

"`CosTransactions::Current Class`" on page 194

Current::suspend()

Synopsis

```
static Control_ptr suspend()
   throw(CORBA::SystemException);
```

Description The suspend() member function suspends the transaction associated with the calling thread. An identifier for the suspended transaction is returned by the function. This identifier can be passed to the Current::resume() function to resume the suspended transaction.

The Current class is nested within the CosTransactions class. The full name for the function is CosTransactions::Current::suspend().

See Also "CosTransactions::Current Class" on page 194

"Current::resume()" on page 198

CosTransactions::RecoveryCoordinator Class

Synopsis

```
class RecoveryCoordinator {
public:
   Status replay_completion(Resource_ptr);
};
typedef RecoveryCoordinator *RecoveryCoordinator_ptr;
class RecoveryCoordinator_var;
```

Description The RecoveryCoordinator class enables a recoverable object to control the recovery process for an associated resource. A RecoveryCoordinator object can be obtained for a recoverable object via the Coordinator object associated with the recoverable object. The Coordinator::register_resource() function returns a RecoveryCoordinator object.

Notes This class is nested within the CosTransactions class. The full name for the class is CosTransactions::RecoveryCoordinator.

OrbixOTS provides a RecoveryCoordinator_ptr type and a RecoveryCoordinator_var helper class. The RecoveryCoordinator_ptr and RecoveryCoordinator_var types hold and manage a reference to the RecoveryCoordinator object.

Class Members

RecoveryCoordinator::replay_completion()

See Also "CosTransactions::Resource Class" on page 202

"Status Enumeration Type" on page 173

RecoveryCoordinator::replay_completion()

Synopsis

```
Status replay_completion( Resource_ptr resource)
   throw(CORBA::SystemException, NotPrepared);
```

Parameters The `resource` parameter specifies the resource associated with the recovery coordinator.

Description The `replay_completion()` member function notifies the recovery coordinator that the `commit()` or `rollback()` operations have not been performed for the associated resource. Notifying the coordinator that the resource has not completed causes completion to be retried, which is useful in certain failure cases. The function returns the current status of the transaction.

This function can be called only for a resource that is prepared. If the resource is not in the prepared state, the `NotPrepared` exception is thrown.

See Also

"CosTransactions::Resource Class" on page 202

"Status Enumeration Type" on page 173

"CosTransactions::RecoveryCoordinator Class" on page 201

"NotPrepared" on page 177

CosTransactions::Resource Class

Synopsis

```
class Resource {
public:
   virtual Vote prepare();
   virtual void rollback();
   virtual void commit();
   virtual void commit_one_phase();
   virtual void forget();
};
typedef Resource *Resource_ptr;
class Resource_var;
```

Description The `Resource` class represents a recoverable resource, that is, a transaction participant that manages data subject to change within a transaction. The `Resource` class specifies the protocol that must be defined for a recoverable resource. Interfaces that inherit from this class must implement each of the member functions to manage the data appropriately for the recoverable object based on the outcome of the transaction. These functions are invoked by the Transaction Service to execute two-phase commit; the requirements of these functions are described in the following sectons.

To become a participant in a transaction, a `Resource` object must be registered with that transaction. The `Coordinator::register_resource()` function can be used to register a resource for the transaction associated with the `Coordinator` object.

A locking mechanism can be used to coordinate access to shared resources. The Object Concurrency Control Service (OCCS) provides classes that enable multiple clients to access a resource without creating inconsistencies in the resource's data. See the reference page for the `CosConcurrencyControl` class for more information.

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Resource`. OrbixOTS provides a `Resource_ptr` type and a `Resource_var` helper class. The `Resource_ptr` and `Resource_var` types hold and manage a reference to the `Resource` object.

Two-phase Commit

The two-phase commit requires both a `prepare()` and a `commit()` function.

A `prepare()` function must be defined to vote on the outcome of the transaction with which the resource is registered. The transaction service invokes this function as the first phase of a two-phase commit; the return value controls the second phase:

- Returns `VoteReadOnly` if the resource's data is not modified by the transaction. The transaction service does not invoke any other functions on the resource, and the resource can forget all knowledge of the transaction.
- Returns `VoteCommit` if the resource's data is written to stable storage by the transaction and the transaction is prepared. Based on the outcome of other participants in the transaction, the transaction service calls either

commit() or rollback() for the resource. The resource should store a reference to the RecoveryCoordinator object in stable storage to support recovery of the resource.

- Returns VoteRollback for all other situations. The transaction service calls the rollback() function for the resource, and the resource can forget all knowledge of the transaction.

A commit() function must be defined to commit all changes made to the resource as part of the transaction. If the forget() function has already been called, no changes need to be committed. If the resource has not been prepared, the NotPrepared exception must be thrown.

Use the heuristic outcome exceptions to report heuristic decisions related to the resource. The resource must remember heuristic outcomes until the forget() function is called, so that the same outcome can be returned if the transaction service calls commit() again.

One-phase Commit

A commit_one_phase() function must be defined to commit all changes made to the resource as part of the transaction. The transaction service may invoke this function if the resource is the only participant in the transaction. Unlike the commit() function, the commit_one_phase() function does not require that the resource be prepared first. Use the heuristic outcome exceptions to report heuristic decisions related to the resource. The resource must remember heuristic outcomes until the forget() function is called, so that the same outcome can be returned if the transaction service calls commit_one_phase() again.

Rollback Transaction

A rollback() function must be defined to undo all changes made to the resource as part of the transaction. If the forget() function has been called, no changes need to be undone. Use the heuristic outcome exceptions to report heuristic decisions related to the resource. The resource must remember heuristic outcomes until the forget() function is called, so that the same outcome can be returned if the transaction service calls rollback() again.

Forget Transaction

A `forget()` function must be defined to cause the resource to forget all knowledge of the transaction. The transaction service invokes this function if the resource throws a heuristic outcome exception in response to the `commit()` or `rollback()` function.

See Also

"CosTransactions::Synchronization Class" on page 207

"CosTransactions::RecoveryCoordinator Class" on page 201

"Vote Enumeration Type" on page 174

CosTransactions::SubtransactionAwareResource Class

Synopsis

```
class SubtransactionAwareResource : Resource {
public:
   virtual void commit_subtransaction(Coordinator);
   virtual void rollback_subtransaction();
};
typedef SubtransactionAwareResource *SubtransactionAwareResource_ptr;
class SubtransactionAwareResource_var;
```

Description

The `SubtransactionAwareResource` class represents a recoverable resource that makes use of nested transactions. This specialised resource object allows the resource to be notified when a subtransaction for which it is registered either commits or rolls back.

The `SubtransactionAwareResource` class specifies the protocol that must be defined for this type of recoverable resource. Interfaces that inherit from this class must implement each of the member functions to manage the recoverable object's data appropriately based on the outcome of the subtransaction. These functions are invoked by the transaction service; the requirements of these functions are described below.

The `Coordinator::register_subtran_aware()` function can be used to register a resource with the subtransaction associated with the `Coordinator` object. The resource can also register with the top-level transaction by using the `Coordinator::register_resource()` function as well; in this case, the protocol for the Resource class must be defined in addition to the protocol for SubtransactionAwareResource. See the reference page for the Resource class for more information.

Commit Subtransaction

A commit_subtransaction() function must be defined to commit all changes made to the resource as part of the subtransaction. If an ancestor transaction rolls back, the subtransaction's changes are rolled back. The transaction service invokes this function if the resource is registered with a subtransaction and it is committed.

The function must be defined to take a Coordinator object as its only argument. When the transaction service invokes this function, it passes the Coordinator object associated with the parent transaction.

Rollback Subtransaction

A rollback_subtransaction() function must be defined to undo all changes made to the resource as part of the subtransaction. The transaction service invokes this function if the resource is registered with a subtransaction and it is rolled back.

Notes

This class is nested within the CosTransactions class. The full name for the class is CosTransactions::SubtransactionAwareResource.

OrbixOTS provides a SubtransactionAwareResource_ptr type and a SubtransactionAwareResource_var helper class. The SubtransactionAwareResource_ptr and SubtransactionAwareResource_var types hold and manage a reference to the SubtransactionAwareResource object.

See Also

"CosTransactions::Coordinator Class" on page 182

"CosTransactions::Resource Class" on page 202

"Status Enumeration Type" on page 173

CosTransactions::Synchronization Class

Synopsis

```
class Synchronization : TransactionalObject {
public:
   virtual void before_completion();
   virtual void after_completion(Status);
};
```

Description

The `Synchronization` class represents a non-recoverable object that maintains transient state data and is dependent on a recoverable object to ensure that the data is persistent. To make data persistent, a synchronization object moves its data to one or more `Resource` objects registered with the same transaction before the transaction completes.

The `Synchronization` class specifies a protocol that must be defined for this type of object. A synchronization object must be implemented as a class derived from the `Synchronization` class. The derived class must implement each of the member functions to ensure that the data maintained by the nonrecoverable object is made recoverable. The transaction service invokes these functions before and after the registered resources commit; the specific requirements of these functions are described in the following sections.

The `Coordinator::register_synchronization()` function can be used to register a synchronization object with the transaction associated with the `Coordinator` object.

Before Completion

A `before_completion()` function must be defined to move the synchronization object's data to a recoverable object. The transaction service invokes this function prior to the prepare phase of the transaction. The function is invoked only if the synchronization object is registered with a transaction and the transaction attempts to commit.

The only exceptions this function can throw are `CORBA::SystemException` exceptions. Throwing other exceptions can cause the transaction to be marked for rollback only.

After Completion

An `after_completion()` function must be defined to do any necessary processing required by the synchronization object; for example, the function could be used to release locks held by the transaction. The transaction service invokes this function after the outcome of the transaction is complete. The function is invoked only if the synchronization object is registered with a transaction and the transaction has either committed or rolled back.

The function must be defined to take a `Status` value as its only argument. When the transaction service invokes this function, it passes the status of the transaction with which the synchronization object is registered.

The only exceptions this function can throw are `CORBA::SystemException` exceptions. Any exceptions that are thrown have no effect on the commitment of the transaction.

Notes

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Synchronization`.

OrbixOTS provides a `Synchronization_ptr` type and a `Synchronization_var` helper class. The `Synchronization_ptr` and `Synchronization_var` types hold and manage a reference to the `Synchronization` object. Refer to the Orbix documentation for more information on the use of object reference types.

See Also

"`CosTransactions::Coordinator Class`" on page 182

"`Coordinator::register_synchronization()`" on page 193

"`CosTransactions::Resource Class`" on page 202

"`Status Enumeration Type`" on page 173

CosTransactions::Terminator Class

Synopsis

```
class Terminator {
public:
   void commit(CORBA::Boolean);
   void rollback();
};
typedef Terminator *Terminator_ptr;
class Terminator_var;
```

Description The `Terminator` class enables explicit termination of a factory-created transaction. The transaction with which the `Terminator` object is associated can be either committed or rolled back. The `Control::get_terminator()` function can be used to return the `Terminator` object associated with a transaction. A `Terminator_ptr` type holds a reference to a `Terminator` object.

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::Terminator`.

OrbixOTS also provides a `Terminator_var` helper class. Both the `Terminator_ptr` and `Terminator_var` types hold and manage a reference to a `Terminator` object.

Class Members

`Terminator::commit()`
`Terminator::rollback()`

See Also

"`CosTransactions::Coordinator Class`" on page 182

"`Control::get_terminator()`" on page 180

"`CosTransactions::Control Class`" on page 178

"`Status Enumeration Type`" on page 173

Terminator::commit()

Synopsis

```
void commit( CORBA::Boolean report_heuristics)
   throw(CORBA::SystemException,
         HeuristicHazard,
         TRANSACTION_ROLLEDBACK);
```

Parameters The `report_heuristics` parameter specifies whether heuristic decisions should be reported for the commit.

Description The commit() member function attempts to commit the transaction associated with the Terminator object. If the report_heuristics parameter is true, the HeuristicHazard exception is thrown when the participants report that a heuristic decision has possibly been made.

The commit() function takes a value of type CORBA::Boolean as its first argument.

If the transaction has been marked as rollback-only, or if all participants in the transaction do not agree to commit, the transaction is rolled back and the CORBA::TRANSACTION_ROLLEDBACK system exception is thrown.

The commit() function takes a value of type CORBA::Boolean as its first argument.

The Terminator class is nested within the CosTransactions class. The full name for the function is CosTransactions::Terminator::commit().

See Also "CosTransactions::Coordinator Class" on page 182

"HeuristicHazard" on page 176

"CosTransactions::Terminator Class" on page 209

"Terminator::rollback()" on page 210

"CosTransactions::Control Class" on page 178

"TRANSACTION_ROLLEDBACK" on page 178

Terminator::rollback()

Synopsis `void rollback();`

Description The rollback() member function rolls back the transaction associated with the Terminator object.

See Also Coordinator class

Terminator class
Terminator::commit()

CosTransactions::TransactionalObject Base Class

Synopsis

```
class TransactionalObject {};
typedef TransactionalObject *TransactionalObject_ptr;
class TransactionalObject_var;
```

Description

The `TransactionalObject` class is the base class for all transactional objects. If an object's interface is derived from this class, the object behaves transactionally. Requests to a transactional object propagate the transaction context of the current thread to the object; that is, the requested operation is executed within the scope of the transaction. If a request is sent to a transactional object and there is no current transaction, the `CORBA::TRANSACTION_REQUIRED` exception is thrown. If a request is sent to a transactional object and the current transaction has already rolled back, the `CORBA::TRANSACTION_ROLLEDBACK` exception is thrown.

Notes

This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::TransactionalObject`.

OrbixOTS also provides a `TransactionalObject_ptr` type and a `TransactionalObject_var` helper class. The `TransactionalObject_ptr` and `TransactionalObject_var` types hold and manage a reference to a `TransactionalObject` object.

See Also

"`CosTransactions::Control Class`" on page 178

"`CosTransactions::Current Class`" on page 194

CosTransactions::TransactionFactory Class

Synopsis

```
class TransactionFactory {
public:
   Control_ptr create(unsigned long timeout);
   Control_ptr recreate(const PropagationContext& ctx);
};
typedef TransactionFactory *TransactionFactory_ptr;
class TransactionFactory_var;
```

Description The `TransactionFactory` class represents a transaction factory that allows the originator of transactions to begin a new transaction for use with the explicit model of transaction demarcation. Servers provide a default instance of this class. Clients can bind to the default instance by using the standard binding mechanism for the object request broker.

Notes This class is nested within the `CosTransactions` class. The full name for the class is `CosTransactions::TransactionFactory`.

OrbixOTS also provides a `TransactionFactory_ptr` type and a `TransactionFactory_var` helper class. The `TransactionFactory_ptr` and `TransactionFactory_var` types hold and manage a reference to a `TransactionFactory` object.

Class Members

```
TransactionFactory::create()
TransactionFactory::recreate();
```

See Also "`CosTransactions::Control Class`" on page 178

TransactionFactory::create()

Synopsis

```
Control_ptr create(unsigned long timeout)
   throw(CORBA::SystemException);
```

Parameters The `timeout` parameter specifies the number of seconds that the transaction waits to complete before rolling back. If the `timeout` parameter is zero, no timeout is set for the transaction.

Description The `create()` member function creates a new top-level transaction for use with the explicit model of transaction demarcation. A `Control` object is returned for the transaction. The `Control` object can be used to propagate the transaction context. See the reference page for the `Control` class for more information.

See Also "`CosTransactions::TransactionFactory Class`" on page 212

"`CosTransactions::Control Class`" on page 178

TransactionFactory::recreate()

Synopsis

```
Control_ptr TransactionFactory::recreate(
   const PropagationContext& ctx);
```

Description Creates a new representation for an existing transaction defined in the propagation context `ctx`. This is used to import a transaction from another domain. The function returns a control object for the new transaction representation.

See Also "`Coordinator::get_txcontext()`" on page 186

12

Concurrency Control Classes

The Object Management Group Object Concurrency Control Service (OMG OCCS) consists of classes used to mediate concurrent access to resources. It enables multiple clients to coordinate their access to shared resources.

Introduction

The OCCS is used in C++ applications where the database or other resource does not have its own object concurrency control. A client use the OCCS in one of two ways:

- It can obtain locks on behalf of a transaction. In this case, the client typically drops all locks after the transaction completes.
- It can obtain locks on behalf of the current thread, which must be executing outside the scope of a transaction. The client must drop locks individually.

The OCCS is only available for C++ servers and client C++ applications must include the `OrbixOTS.hh` header file.

Lock Sets

A lock set is a collection of locks associated with a single resource. Lock sets are represented by:

- `CosConcurrencyControl::LockSet` objects, for nontransactional clients and clients using the implicit transactional model.
- `CosConcurrencyControl::TransactionalLockSet` objects, for clients using the explicit transactional model.

Clients must associate lock sets with resources. That is, the client must define and maintain the mapping between lock sets and resources and consistently using locking when accessing that resource. Lock sets are created by using functions from the `CosConcurrencyControl::LockSetFactory` class.

Lock Modes

Locks can be obtained in specific modes that determine the degree of concurrent access permitted to locked data. OCCS supports five lock modes, defined by the `CosConcurrencyControl::lock_mode` data type: `read`, `write`, `upgrade`, `intention_read`, and `intention_write`. The following table defines the compatibility between the modes:

	Requested Mode				
Granted Mode	**IR**	**R**	**U**	**IW**	**W**
Intention Read (IR)					
Read (R)					
Upgrade (U)					
Intention Write (IW)					
Write (W)					

Table 12.1: *Lock Compatibility*

The shading indicates when locks conflict.

Lock Duration

Locks held on behalf of a transaction are typically held until the transaction commits or aborts, at which time the locks can be dropped using the `CosConcurrencyControl::LockCoordinator::drop_locks()` function. This function drops all locks held by the transaction. If a transactional client wants to release one or more locks before the transaction completes, it can use the `CosConcurrencyControl::LockSet::unlock()` or `CosConcurrencyControl::TransactionalLockSet::unlock()` functions to do so.

A lock coordinator manages the release of locks held by a transaction. Lock sets that are related share the same lock coordinator. A client can determine the coordinator by using the `CosConcurrencyControl::LockSet::get_coordinator()` or `CosConcurrencyControl::TransactionalLockSet::get_coordinator()` function.

Locks held by threads outside of the scope of a transaction must be explicitly dropped by using the `CosConcurrencyControl::LockSet::unlock()` function.

Overview of the Classes

The OCCS classes provide the following functionality:

- Defining lock sets:

  ```
  CosConcurrencyControl::LockSet
  CosConcurrencyControl::TransactionalLockSet
  ```

- Creating lock sets:

  ```
  CosConcurrencyControl::LockSetFactory
  ```

- Dropping locks held by a transaction:

  ```
  CosConcurrencyControl::LockCoordinator
  ```

Lock Mode Enumeration Data Type

Synopsis

```
enum lock_mode {
   read,
   write,
   upgrade,
```

```
    intention_read,
    intention_write
};
```

Description The lock_mode data type is used to specify the lock mode. Two of the lock modes, intention_read and intention_write, are used to specify intention locks. Intention locks are used when locking hierarchical resources and are typically obtained on the root or ancestors of a desired resource. They provide a way to minimise potential conflicts on lower-level resources without needlessly using locks of coarser granularity.

Constants

read	Specifies a read lock. Other transactions can read the locked data, but none can modify the data while a read lock is held.
write	Specifies a write lock. No other transaction can simultaneously access the locked data while a write lock is held.
upgrade	Specifies an upgrade lock. An upgrade lock is a type of read lock that is used if a transaction needs to read data that it may subsequently need to write. An upgrade lock conflicts with other upgrade locks held on behalf of other transactions. If an upgrade lock is obtained successfully, it indicates that no other upgrade lock is held on that data and prevents any new upgrade locks from being obtained on that data. This type of lock can be used to head off potential deadlocks.
intention_read	Specifies an intention read lock.
intention_write	Specifies an intention write lock.

See Also

"CosConcurrencyControl Base Class" on page 219

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

CosConcurrencyControl Base Class

Synopsis

```
class CosConcurrencyControl {
public:
   enum lock_mode {
         read, write, upgrade,
         intention_read, intention_write
   };
   class LockNotHeld : public CORBA::UserException{...};
   class LockCoordinator {...};
   class LockSet {...};
   class TransactionalLockSet {...};
   class LockSetFactory {...};
   typedef LockCoordinator* LockCoordinator_ptr;
   typedef LockCoordinator* LockCoordinatorRef;
   typedef LockSet* LockSet_ptr;
   typedef TransactionalLockSet* TransactionalLockSet_ptr;
};
```

Description

The `CosConcurrencyControl` class is the base class for Object Concurrency Control Service (OCCS) classes. The Concurrency Control service enables multiple clients to coordinate their access to shared resources. Locks can be held on behalf of a transaction or on behalf of the current thread, which must be executing outside of the scope of a transaction. Transactional clients can use either the implicit or explicit transaction model. The class `CosConcurrencyControl::LockSet` is used by Nontransactional clients and by transactional clients using the implicit transaction model. The `CosConcurrencyControl::TransactionalLockSet` is used by transactional clients using the explicit transaction model.

The `CosConcurrencyControl` class contains the classes used for locking. It also contains the `CosConcurrencyControl::lock_mode` data type and several defined types used by OCCS classes.

Class Members

`CosConcurrencyControl::LockCoordinator` class
`CosConcurrencyControl::LockSet` class
`CosConcurrencyControl::LockSetFactory` class
`CosConcurrencyControl::TransactionalLockSet` class
`CosConcurrencyControl::lock_mode`

CosConcurrencyControl::LockCoordinator Class

Synopsis

```
class ConcurrencyControl::LockCoordinator {
public:
   void drop_locks();
};
```

Description The `LockCoordinator` class represents a lock coordinator. A `LockCoordinator` object is created for each transaction that creates `CosConcurrencyControl::LockSet` or `CosConcurrencyControl::TransactionalLockSet` objects.

Class Members

`CosConcurrencyControl::LockCoordinator::drop_locks()`

See Also “`CosConcurrencyControl::LockSet Class`” on page 221

“`CosConcurrencyControl::TransactionalLockSet Class`” on page 229

LockCoordinator::drop_locks()

Synopsis

```
void drop_locks()
   throw(CORBA::SystemException);
```

Description The `drop_locks()` function releases all locks held by a transaction. A client must call this function after a transaction commits or aborts. For nested transactions, this function must be called only when the nested transaction aborts.

See Also “`CosConcurrencyControl::LockCoordinator Class`” on page 220

“`LockSet::get_coordinator()`” on page 225

“`TransactionalLockSet::get_coordinator()`” on page 233

CosConcurrencyControl::LockSet Class

Synopsis

```
class CosConcurrencyControl::LockSet {
public:
   void lock(CosConcurrencyControl::lock_mode);
   CORBA::Boolean try_lock(CosConcurrencyControl::lock_mode);
   void unlock(CosConcurrencyControl::lock_mode);
   void change_mode(
         CosConcurrencyControl::lock_mode,
         CosConcurrencyControl::lock_mode
   );
   CosConcurrencyControl::LockCoordintor_ptr get_coordinator(
         CosTransactions::Coordinator_ptr
   );
};
```

Description

The `LockSet` class represents a lock set. A lock set is a collection of locks associated with a single resource. Clients must associate a `LockSet` object with a resource.

This `LockSet` class includes functions for acquiring and releasing locks, for changing the lock mode of an existing lock, and for determining the lock coordinator associated with a specific transaction.

`LockSet` objects can used by clients operating in the implicit transactional model. Locks are called and released on behalf of the calling thread or transaction. Clients using the explicit transactional model use `TransactionalLockSet` objects. Transactional clients must release all locks when the transaction commits or aborts by calling the `CosConcurrencyControl::LockCoordinator::drop_locks()` function. Transactional clients can also use the `CosConcurrencyControl::LockSet::unlock()` function to release specific locks.

`LockSet` objects can also be used nontransactionally. Clients use the `CosConcurrencyControl::LockSet::lock()` function or the `CosConcurrencyControl::LockSet::try_lock()` function to obtain a lock on the resource associated with the `LockSet` object. Nontransactional clients must drop locks explicitly by calling the `CosConcurrencyControl::LockSet::unlock()` function.

Lock sets are created by using the `CosConcurrencyControl::LockSetFactory` class. Functions of that class are used to create `CosConcurrencyControl::LockSet` and `CosConcurrencyControl::TransactionalLockSet` objects.

Class Members
`CosConcurrencyControl::LockSet::change_mode`
`CosConcurrencyControl::LockSet::get_coordinator()`
`CosConcurrencyControl::LockSet::lock()`
`CosConcurrencyControl::LockSet::try_lock()`
`CosConcurrencyControl::LockSet::unlock()`

See Also
"Lock Modes" on page 216

"`CosConcurrencyControl::LockSetFactory Class`" on page 226

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

LockSet::lock()

Synopsis

```
void lock(CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException);
```

Parameters The `mode` parameter specifies the lock mode for the acquired lock.

Description The `lock()` function acquires a lock in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client, the operation blocks until the lock is acquired. If the call is on behalf of a transactional client and the transaction is aborted while the call is blocked, the `CORBA::TRANSACTION_ROLLEDBACK` exception is thrown.

This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also
"Lock Modes" on page 216

"`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

"`LockSet::try_lock()`" on page 223

"`LockSet::unlock()`" on page 223

LockSet::try_lock()

Synopsis

```
CORBA::Boolean try_lock(CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException);
```

Parameters The `mode` parameter specifies the lock mode for the acquired lock.

Description The `try_lock()` function attempts to acquire a lock in the specified mode. If a lock is held on the same lock set in an incompatible mode by another client, the function returns false to indicate that the lock can not be acquired. If the function is called on behalf of a transactional client and the transaction is aborted while the function is trying to acquire the lock, the `CORBA::TRANSACTION_ROLLEDBACK` exception is thrown.

This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "Lock Modes" on page 216

"`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

"`LockSet::try_lock()`" on page 223

"`LockSet::unlock()`" on page 223

LockSet::unlock()

Synopsis

```
void unlock(CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException,
         CosConcurrencyControl::LockNotHeld);
```

Parameters The `mode` parameter specifies the lock mode for the dropped lock.

Description The `unlock()` function drops a single lock in the specified mode. (A client can hold multiple locks in the same mode.) Transactional clients must release all locks when the transaction commits or aborts by calling the `CosConcurrencyControl::LockCoordinator::drop_locks()` function. Nontransactional clients must drop locks explicitly by using the `unlock()` function. Transactional clients can also use the `unlock()` function to release specific locks.

If an application attempts to drop a lock that is not held, the Cos:ConcurrencyControl::LockNotHeld exception is thrown.

This function takes one additional parameter of type CORBA::Environment for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "Lock Modes" on page 216

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

"LockSet::try_lock()" on page 223

"LockSet::lock()" on page 222

LockSet::change_mode()

Synopsis

```
void change_mode(
        CosConcurrencyControl::lock_mode held_mode,
        CosConcurrencyControl::lock_mode new_mode)
    throw(CORBA::SystemException,
            CosConcurrencyControl::LockNotHeld);
```

Parameters

held_mode	Specifies the current lock mode.
new_mode	Specifies the new lock mode.

Description The change_mode() function changes the mode of a single lock. If the new mode conflicts with an existing lock mode held by an unrelated client, the function is blocked until the new mode can be granted. If the call is on behalf of a transactional client and the transaction is aborted while the call is blocked, the CORBA::TRANSACTION_ROLLEDBACK exception is thrown. If an application tries to change the mode of a lock that is not held, the CosConcurrencyControl::LockNotHeld exception is thrown. This function takes one additional parameter of type CORBA::Environment for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "Lock Modes" on page 216

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

"LockSet::try_lock()" on page 223

"LockSet::lock()" on page 222

LockSet::get_coordinator()

Synopsis

```
CosConcurrencyControl::LockCoordinator_ptr get_coordinator(
        CosTransactions::Coordinator_ptr which)
    throw(CORBA::SystemException);
```

Parameters The `which` parameter specifies the transaction for which the lock coordinator is to be returned. To return the lock coordinator for the transaction implicitly associated with the current thread, specify a value of `CosTransactions::Coordinator::_nil()`.

Description The `get_coordinator()` function returns the lock coordinator associated with the specified transaction. This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "CosTransactions::Coordinator Class" on page 182

"CosConcurrencyControl::LockCoordinator Class" on page 220

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

CosConcurrencyControl::LockSetFactory Class

Synopsis

```
class ConcurrencyControl::LockSetFactory {
public:
   CosConcurrencyControl::LockSet_ptr create();
   CosConcurrencyControl::LockSet_ptr create_related(
      CosConcurrencyControl::LockSet_ptr);
   CosConcurrencyControl::TransactionalLockSet_ptr
         create_transactional();
   CosConcurrencyControl::TransactionalLockSet_ptr
         create_transactional_related(
            CosConcurrencyControl::TransactionalLockSet_ptr);
};
```

Description The `LockSetFactory` class represents a lock set factory. This class includes functions that are used to create objects of the `LockSet` class and the `TransactionalLockSet` class.

Class Members

`CosConcurrencyControl::LockSetFactory::create()`
`CosConcurrencyControl::LockSetFactory::create_related()`
`CosConcurrencyControl::LockSetFactory::create_transactional()`
`CosConcurrencyControl::LockSetFactory::create_transactional_related()`

See Also "`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

LockSetFactory::create()

Synopsis

```
CosConcurrencyControl::LockSet_ptr create()
   throw(CORBA::SystemException);
```

Description The `create()` function creates a new object of the `LockSet` class and a lock coordinator for that lock set. This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::LockSetFactory Class`" on page 226

"LockSetFactory::create_related()" on page 227

"LockSetFactory::create_transactional()" on page 227

LockSetFactory::create_related()

Synopsis

```
CosConcurrencyControl::LockSet_ptr create_related(
        CosConcurrencyControl::LockSet_ptr which)
    throw(CORBA::SystemException);
```

Parameters The `which` parameter specifies an existing lock set to which the new lock set is to be related.

Description The `create_related()` function creates a new object of the `LockSet` class related to an existing lock set. Related lock sets drop their locks together. This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

WARNING: This function is currently not implemented. Attempting to call this function results in an error.

See Also

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::LockSetFactory Class" on page 226

"LockSetFactory::create()" on page 226

"LockSetFactory::create_transactional()" on page 227

LockSetFactory::create_transactional()

Synopsis

```
CosConcurrencyControl::TransactionalLockSet_ptr
    create_transactional()
        throw(CORBA::SystemException);
```

Description The `create_transactional()` function creates a new object of the `TransactionalLockSet` class and a lock coordinator for that lock set. Transactional lock sets are used by clients using the explicit transactional model.

This function takes one additional parameter of type `CORBA::Environment` for C++ compilers that do not support exception handling. If your compiler supports exceptions, use the parameter's default value.

See Also "`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

"`LockSetFactory::create()`" on page 226

"`LockSetFactory::create_transactional_related()`" on page 228

LockSetFactory::create_transactional_related()

Synopsis

```
CosConcurrencyControl::TransactionalLockSet_ptr
   create_transactional_related(
      CosConcurrencyControl::TransactionalLockSet_ptr which)
         throw(CORBA::SystemException);
```

Parameters The `which` parameter specifies an existing transactional lock set to which the new lock set is to be related.

Description The `create_transactional_related()` function creates a new object of the `TransactionalLockSet` class related to an existing transactional lock set. Related lock sets drop their locks together. Transactional lock sets are used by clients using the explicit transactional model.

WARNING: This function is currently not implemented. Attempting to call this function results in an error.

See Also "`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

"`CosConcurrencyControl::LockSetFactory Class`" on page 226

"`LockSetFactory::create_transactional()`" on page 227

CosConcurrencyControl::TransactionalLockSet Class

Synopsis

```
class ConcurrencyControl::TransactionalLockSet {
public:
   void lock(
      CosTransactions::Coordinator_ptr,
      CosConcurrencyControl::lock_mode);
   CORBA::Boolean try_lock(
      CosTransactions::Coordinator_ptr,
      CosConcurrencyControl::lock_mode);
   void unlock(
      CosTransactions::Coordinator_ptr,
      CosConcurrencyControl::lock_mode);
   void change_mode(
      CosTransactions::Coordinator_ptr,
      CosConcurrencyControl::lock_mode,
      CosConcurrencyControl::lock_mode);
   CosConcurrencyControl::LockCoordintor_ptr get_coordinator(
      CosTransactions::Coordinator_ptr);
};
```

Description

The `TransactionalLockSet` class represents a transactional lock set. A lock set is a collection of locks associated with a single resource. Clients must associate a `TransactionalLockSet` object with a resource. The `TransactionalLockSet` class includes functions for acquiring and releasing locks, for changing the lock mode of an existing lock, and for determining the lock coordinator associated with a specific transaction.

Clients must release all locks when the transaction commits or aborts by using the `CosConcurrencyControl::LockCoordinator::drop_locks()` function.

`TransactionalLockSet` objects can be used by clients that are using the explicit transactions model. The operations provided in the interface operate identically to those in the `LockSet` class. However, functions in the `TransactionalLockSet` class take an additional parameter, which is used to explicitly specify the transaction coordinator.

Lock sets are created by using the `CosConcurrencyControl::LockSetFactory` class. Functions of this class are used to create `CosConcurrencyControl::LockSet` and `CosConcurrencyControl::TransactionalLockSet` objects.

Class Members `CosConcurrencyControl::TransactionalLockSet::change_mode()`
`CosConcurrencyControl::TransactionalLockSet::get_coordinator()`
`CosConcurrencyControl::TransactionalLockSet::lock()`
`CosConcurrencyControl::TransactionalLockSet::try_lock()`

See Also

"`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::LockSetFactory Class`" on page 226

TransactionalLockSet::lock()

Synopsis

```
void lock(
     CosTransactions::Coordinator_ptr which,
     CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException);
```

Parameters

`which`	Specifies the coordinator for the transaction.
`mode`	Specifies the lock mode for the acquired lock.

Description The `lock()` function acquires a lock in the specified mode on behalf of the specified transaction. If a lock is held on the same lock set in an incompatible mode by another client, the operation is blocked until the lock is acquired. If the transaction is aborted, the `CORBA::TRANSACTION_ROLLEDBACK` exception is thrown.

See Also

"`CosTransactions::Coordinator Class`" on page 182

"Lock Modes" on page 216

"`CosConcurrencyControl::LockSet Class`" on page 221

"`CosConcurrencyControl::TransactionalLockSet Class`" on page 229

"`TransactionalLockSet::try_lock()`" on page 231

"`TransactionalLockSet::unlock()`" on page 231

TransactionalLockSet::try_lock()

Synopsis

```
CORBA::Boolean try_lock(
      CosTransactions::Coordinator_ptr which,
      CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException);
```

Parameters

which	Specifies the coordinator for the transaction.
mode	Specifies the lock mode for the acquired lock.

Description The try_lock() function attempts to acquire a lock in the specified mode on behalf of the specified transaction. If a lock is held on the same lock set in an incompatible mode by another client, the function returns false to indicate that the lock could not be acquired. If the transaction is aborted while the function is trying the lock, theCORBA::TRANSACTION_ROLLEDBACK exception is thrown.

See Also

"CosTransactions::Coordinator Class" on page 182

"Lock Modes" on page 216

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

"TransactionalLockSet::lock()" on page 230

"TransactionalLockSet::unlock()" on page 231

TransactionalLockSet::unlock()

Synopsis

```
void unlock(
      CosTransactions::Coordinator_ptr which,
      CosConcurrencyControl::lock_mode mode)
   throw(CORBA::SystemException,
        CosConcurrencyControl::LockNotHeld);
```

Parameters

which	Specifies the coordinator for the transaction.
mode	Specifies the lock mode for the dropped lock.

Description The unlock() function drops a single lock in the specified mode on behalf of the specified transaction. (A client can hold multiple locks in the same mode.) If an application attempts to drop a lock that is not held, the CosConcurrencyControl::LockNotHeld exception is thrown.

See Also "CosTransactions::Coordinator Class" on page 182

"Lock Modes" on page 216

"CosConcurrencyControl::LockSet Class" on page 221

"CosConcurrencyControl::TransactionalLockSet Class" on page 229

"TransactionalLockSet::try_lock()" on page 231

"TransactionalLockSet::lock()" on page 230

TransactionalLockSet::change_mode()

Synopsis

```
void change_mode(
      CosTransactions::Coordinator_ptr which,
      CosConcurrencyControl::lock_mode held_mode,
      CosConcurrencyControl::lock_mode new_mode)
   throw(CORBA::SystemException,
         CosConcurrencyControl::LockNotHeld);
```

Parameters

which	Specifies the coordinator for the transaction.
held_mode	Specifies the current lock mode.
new_mode	Specifies the new lock mode.

Description The change_mode() function changes the mode of a single lock held on behalf of the specified transaction. If the new mode conflicts with an existing mode held by an unrelated client, the function is blocked until the new mode can be granted. If the call is blocked and the transaction is aborted, the CORBA::TRANSACTION_ROLLEDBACK exception is thrown. The Cos:ConcurrencyControl::LockNotHeld exception is thrown if an application tries to change the lock mode of a lock that is not held.

See Also "CosTransactions::Coordinator Class" on page 182

"Lock Modes" on page 216

“CosConcurrencyControl::LockSet Class” on page 221

“CosConcurrencyControl::TransactionalLockSet Class” on page 229

“TransactionalLockSet::try_lock()” on page 231

“TransactionalLockSet::lock()” on page 230

TransactionalLockSet::get_coordinator()

Synopsis

```
CosConcurrencyControl::LockCoordinator_ptr get_coordinator(
      CosTransactions::Coordinator_ptr which)
   throw(CORBA::SystemException);
```

Parameters The `which` parameter specifies the transaction for which the lock coordinator is to be returned. To return the lock coordinator for the transaction implicitly associated with the current thread, specify a value of `CosTransactions::Coordinator::_nil()`.

Description The `get_coordinator()` function returns the lock coordinator associated with the specified transaction.

See Also “CosTransactions::Coordinator Class” on page 182

“CosConcurrencyControl::LockSet Class” on page 221

“CosConcurrencyControl::TransactionalLockSet Class” on page 229

13

Java Classes

OrbixOTS provides a Java implementation of the CORBA OTS interface. It supports the development of distributed transactional applications using the Orbix Java Edition object request broker.

Introduction

The OrbixOTS Java classes consist of two main packages that contain the standard and non-standard interfaces that make up the OTS implementation. The standards based `org.omg.CORBA.CosTransactions` package contains the Java mapping of the OMG OTS IDL interfaces as implemented by the Orbix Java Edition IDL compiler. These interfaces are described in detail in the latter part of this chapter.

The `IE.Iona.OrbixWeb.CosTransactions` package contains the OrbixOTS implementation specific interfaces that are used for initializing and configuring OTS client and server applications.

In order to reference these classes by name in your code, import the classes using the standard syntax, bearing in mind that order is important:

```
import org.omg.CosTransactions.*;
import IE.Iona.OrbixWeb.CosTransacitons.*;
```

All OTS operations can throw `CORBA::SystemExceptions` if an object request broker (ORB) errors occur.

C++ implementations for the `CosTransactions` interfaces are described in Chapter 11, “CosTransactions Module” on page 171.

Overview of the Classes

Package IE.Iona.OrbixWeb.CosTransactions

The classes in this package provide the following functionality:

- `Client`: this class configures, initializes and terminates transactional clients.
- `Server`: this class configures, initializes and terminates transactional servers.
- `TransactionPolicy`: this class defines which OrbixOTS transaction policies are supported.

Package org.omg.CosTransactions

The OMG OTS classes in this package provide the following functionality.

- Defining transactional interfaces in the CORBA environment:

  ```
  TransactionalObject
  ```
- Managing transactions under the implicit model:

  ```
  Current
  ```
- Managing transactions under the explicit model:

  ```
  TransactionFactory
  Control
  Coordinator
  Terminator
  ```
- Managing recoverable resources in the CORBA environment:

  ```
  RecoveryCoordinator
  Resource
  SubtransactionAwareResource
  Synchronization
  ```
- Reporting system errors:

  ```
  HeuristicCommit
  HeuristicHazard
  HeuristicMixed
  HeuristicRollback
  Inactive
  ```

```
InvalidControl
INVALID_TRANSACTION
NoTransaction
NotPrepared
NotSubtransaction
SubtransactionsUnavailable
TRANSACTION_REQUIRED
TRANSACTION_ROLLEDBACK
Unavailable
```

The OtsEnv, Client and Server Classes

Much functionality is shared by the `Client` and `Server` classes through a common abstract base class called `OtsEnv`. This class cannot be instantiated and so cannot be used directly by applications but its inherited members provide the main body of configuration functionality.

OtsEnv Class

Synopsis

```
public abstract class OtsEnv {
   public void init();
   public void exit(int status);
   public void shutdown();
   public void setDefaultFactory(orb.omg.CORBA.Object
               remoteObject);
   public void setInterfaceTransactionPolicy(java.lang.String i,
            IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy p);
   public void setObjectTransactionPolicy(org.omg.CORBA.Object o,
            IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy p);
   public IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy
               getDefaultTransactionPolicy();
   public TransactionPolicy setDefaultTransactionPolicy(
         IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy.
               TransactionPolicy policy);
   public long setGCPeriod(long t);
}
```

Description The class `IE.Iona.OrbixWeb.CosTransactions.OtsEnv` is the super class of both the `Client` and `Server` classes. This class provides implementation details common to both the client and server. Because it is an abstract class it cannot be instantiated, and only exists within a `Client` or `Server` instance.

OtsEnv.init()

Synopsis `public void init();`

Description The `init()` method initializes OrbixOTS components. This method must be called at least once before an application can attempt transactional operations or obtain the `TransactionCurrent` reference from `ORB.resolve_initial_references`. It is responsible for installing the appropriate interceptors for transaction propagation and for registering this service with the current ORB.

OtsEnv.shutdown()

Synopsis `public void shutdown();`

Description The shutdown method shuts the OrbixOTS component down. Any outstanding transactions are rolled back.

OtsEnv.exit()

Synopsis `public void exit(int status);`

Parameters The `status` parameter specifies the exit status for the client application.

Description The exit method shuts down the OrbixOTS component and terminates the application by calling `System.exit()` with the indicated status. Any outstanding transactions are rolled back.

OtsEnv.setDefaultTransactionPolicy()

Synopsis

```
public TransactionPolicy setDefaultTransactionPolicy(
      IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy policy);
```

Parameters The `policy` parameter specifies the current default `TransactionPolicy`.

Description The `setDefaultTransactionPolicy()` function sets the default `TransactionPolicy`.

Returns The previous `TransactionPolicy`.

Notes IONA-specific.

See Also “Status Enumeration Class Type” on page 265

OtsEnv.getDefaultTransactionPolicy()

Synopsis

```
public IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy
              getDefaultTransactionPolicy();
```

Description The `getDefaultTransactionPolicy()` function gets the current default `TransactionPolicy`.

Returns The current default `TransactionPolicy`.

Notes IONA-specific.

See Also “TransactionFactory Class” on page 264

OtsEnv.setInterfaceTransactionPolicy()

Synopsis

```
public void setInterfaceTransactionPolicy(java.lang.String i,
            IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy p);
```

Parameters

`i`	The interface to treat as transactional.
`p`	The `TransactionPolicy` for this transactional interface.

Description The `setInterfaceTransactionPolicy()` function marks an interface as transactional and specifies the transaction policy for this transactional interface. Objects that support this interface are treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface. The `interface` parameter is the CORBA repository identifier for the interface that is of the form "IDL:X:1.0".

Notes IONA-specific.

See Also "Status Enumeration Class Type" on page 265

OtsEnv.setObjectTransactionPolicy()

Synopsis

```
public void setObjectTransactionPolicy(org.omg.CORBA.Object o,
            IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy p);
```

Parameters

`o`	The object to treat as transactional.
`p`	The `TransactionPolicy` for this transactional object.

Description The `setObjectTransactionPolicy()` function marks an object as transactional and specifies the transaction policy for this transactional object. This object is treated as transactional in this process even if the object does not (or is not known to) implement the `CosTransactions::TransactionalObject` CORBA interface.

Notes IONA-specific.

See Also "Status Enumeration Class Type" on page 265

OtsEnv.setDefaultFactory()

Synopsis

```
public void setDefaultFactory(orb.omg.CORBA.Object
               remoteObject);
```

Parameters The `remoteObject` parameter specifies the remote `TransactionFactory` to use. This object is usually a reference to a `CosTransactions.TransactionFactory` object, but it can simply be a reference to any object in an OrbixOTS C++ server. If the specified object is not a `TransactionFactory` reference OrbixOTS uses the Orbix Java Edition bind mechanism to obtain a reference from the object's server. This is possible because each OrbixOTS C++ server supports the TransactionFactory interface.

Description OrbixOTS automatically attempts to use the OrbixOTS standalone transaction factory (`otstf`) as its default factory by resolving the default transaction factory name from the Name Service. You can use this method to specify an alternative default factory. OrbixOTS uses the default transaction factory to create transactions. If the configuration variable `OrbixOTS.OTS_USE_DEFAULT_FACTORY` is set to `FALSE`, OrbixOTS attempts to create the transaction as part of the first transactional request. In other words it uses the transaction factory on the target server. See "Use of otstf by OrbixOTS for Java" on page 281.

OtsEnv.setGCPeriod()

Synopsis `public long setGCPeriod(long t);`

Parameters The parameter `t` specifies the garbage collection period in milliseconds.

Description This method sets the sweep period for the garbage collection thread. The garbage collection thread releases references to dead threads and stale objects. The period defaults to three minutes.

Client Class

Synopsis

```
public class Client {
   public static Client IT_Create();
   public static Client IT_Create(org.omg.CORBA.ORB);
}
```

Description The class `IE.Iona.OrbixWeb.CosTransactions.Client` is used to instantiate an OrbixOTS Java Client. It supports two methods that return an instance of this class. An application that needs to use the Client OrbixOTS Java classes must obtain a Client reference using one of the methods described below and initialize it using the inherited `init()` method before attempting transactional operations. The optional ORB parameter allows an application developer to specify the ORB instance to use. If none is specified the default `ORB`, `_CORBA.Orbix` is used.

See Also "`OtsEnv Class`" on page 237.

Client.IT_create()

Synopsis `public static Client IT_create()`

Description Creates an instance of a `Client` pseudo-object. This instance is associated with the default ORB, `_CORBA.Orbix`.

Returns An instance of the `Client` class.

See Also Other `IT_create` constructor.

Client.IT_create()

Synopsis `public static Client IT_create(org.omg.CORBA.ORB);`

Description Creates an instance of a `Client` pseudo-object and specifies the ORB instance to use. This allows an application to use multiple ORBs.

Returns A new instance of the `Client` class.

See Also Other `IT_create` constructor.

Server Class

Synopsis

```
class Server {
public:
   static Server IT_create();
   static Server IT_create(org.omg.CORBA.ORB);
};
```

Description The class `IE.Iona.OrbixWeb.CosTransactions.Server` is used to instantiate an OrbixOTS Java Server. It supports two methods that return an instance of this class. An application that needs to use the `Server` OrbixOTS Java classes must obtain a `Server` reference using one of the methods described below and initialize it using its inherited `init()` method before attempting transactional operations. The optional ORB parameter allows an application developer to specify the ORB instance to use. If none is specified the default ORB `_CORBA.Orbix` is used.

See Also "`OtsEnv Class`" on page 237

Server.IT_create()

Synopsis `public static Server IT_create()`

Description Create a new `Server` object. This instance is associated with the default ORB.

Returns A new instance of the `Server` class.

See Also Other `IT_create()` method

Server.IT_create()

Synopsis `public static Server IT_create(org.omg.CORBA.ORB)`

Description Create a Server object instance and specify the ORB instance to use. This allows an application to use multiple ORBs.

Returns A new instance of the `Server` class.

See Also Other `IT_create()` method

TransactionPolicy Class

Synopsis

```
public final class TransactionPolicy {
   public static final
   TransactionPolicy TransactionRequired;

   public static final
   TransactionPolicy TransactionAllowed;
};
```

Description The class `IE.Iona.OrbixWeb.CosTransactions.TransactionPolicy` encapsulates different transaction policies. The two policies supported are:

`TransactionRequired`	All invocations on a transactional interface must occur within the scope of a transaction.
`TransactionAllowed`	All invocations on a transactional interface can occur both within a transaction and without a transaction. If the client is associated with a transaction the transaction's context is propogated to the server; otherwise no transaction context is propogated.

Current Class

Synopsis

```
class Current {
public:
   static Current IT_create();
   static Current IT_create(OtsEvn env);
   void begin();
   void commit(boolean);
   void rollback();
   void rollback_only();
   int get_status();
   String get_transaction_name();
   void set_timeout(int);
   Control get_control();
   Control suspend();
   void resume(Control);
```

```
};
```

Description The class `org.omg.CosTransactions.Current` represents a transaction that is associated with the calling thread; the thread defines the transaction context. The transaction context is propagated implicitly when the client issues requests.

This class defines member functions for beginning, committing, and aborting a transaction using the implicit model of transaction control. It also defines member functions for suspending and resuming a transaction and for retrieving information about a transaction.

Class Members

```
Current.begin()
Current.commit()
Current.get_control()
Current.get_status()
Current.get_transaction_name()
Current.IT_create()
Current.resume()
Current.rollback()
Current.rollback_only()
Current.set_timeout()
Current.suspend()
```

See Also "`Control Class`" on page 250

Current.begin()

Synopsis

```
void begin()
   throws SubtransactionsUnavailable, Inactive, SystemException;
```

Description The `begin()` function creates a new transaction and modifies the transaction context of the calling thread to associate the thread with the new transaction. If a transaction is already associated with the current thread, the begin function starts a subtransaction.

If a transaction is associated with the calling thread but is already rolled back, the `SubtransactionsUnavailable` exception is thrown.

See Also "`Current.commit()`" on page 246

"`Current.rollback()`" on page 248

"`Current.rollback_only()`" on page 248

Current.commit()

Synopsis

```
void commit(boolean reportHeuristics)
   throws NoTransaction, HeuristicMixed, HeuristicHazard,
          SystemException;
```

Parameters The `reportHeuristics` parameter specifies whether heuristic decisions should be reported for the transaction associated with the calling thread.

Description The `commit()` member function attempts to commit the transaction associated with the calling thread. If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown. If the `reportHeuristics` parameter is `true`, an exception is thrown when a heuristic decision has possibly been made.

See Also "`Current.begin()`" on page 245

"`Current.rollback()`" on page 248

"`Current.rollback_only()`" on page 248

"`HeuristicHazard`" on page 267

"`NoTransaction`" on page 268

Current.get_control()

Synopsis

```
Control get_control()
   throws SystemException;
```

Description The `get_control()` member function returns a reference to the `Control` object for the transaction associated with the calling thread. If no transaction is associated with the calling thread, a null object reference is returned.

See Also "`Current.resume()`" on page 248

Current.get_status()

Synopsis

```
Status get_status()
   throws SystemException
```

Description The `get_status()` function returns the status of the transaction associated with the calling thread. If no transaction is associated with the calling thread, the `Status.StatusNoTransaction` value is returned.

The status returned indicates the processing phase of the transaction. See the reference page for `Status` for information about the possible status values.

See Also "`Status Enumeration Class Type`" on page 265

Current.get_transaction_name()

Synopsis

```
String get_transaction_name()
   throws SystemException
```

Description The `get_transaction_name()` function returns the name of the transaction associated with the calling thread. If no transaction is associated with the calling thread, a null string is returned.

Current.IT_create()

Synopsis

```
public static Current IT_create();
```

Description This method returns an uninitialized instance of the `TransactionsCurrent` pseudo object. This method call should be followed by a call to `Current.init()`. It is recommended that you use the ORB method `resolve_initial_references("TransactionCurrent")` in preference to this method.

Current.IT_create()

Synopsis

```
public static Current IT_create(OtsEnv env);
```

Description Creates an instance of the `Current` pseudo object. The env parameter specifies the client of server instance to use. `OtsEnv` is the super class of both the `Client` and `Server` classes. This is useful in the case of the use of multiple orbs. `ORB.resolve_initial_references()` should be used in preference to this method. The method is supported for backward compatibility only.

Current.resume()

Synopsis

```
void resume(Control which)
   throw InvalidControl, SystemException;
```

Parameters The `which` parameter specifies a `Control` object that represents the transaction context associated with the calling thread.

Description The `resume()` function resumes the suspended transaction identified by the `which` parameter and associated with the calling thread. If the value of the `which` parameter is a null object reference, the calling thread disassociates from the transaction. If a non-null parameter is invalid, the `InvalidControl` exception is thrown.

See Also "`Current.get_control()`" on page 246

"`Current.suspend()`" on page 249

"`InvalidControl`" on page 268

Current.rollback()

Synopsis

```
void rollback()
   throws NoTransaction, SystemException;
```

Description The `rollback()` function rolls back the transaction associated with the calling thread. If the transaction was started with the `Current.begin()` function, the transaction context for the thread is restored to its state before the transaction was started; otherwise, the transaction context is set to null.

If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown.

See Also "`Current.begin()`" on page 245

"`Current.rollback_only()`" on page 248

"`NoTransaction`" on page 268

Current.rollback_only()

Synopsis

```
void rollback_only()
   throw NoTransaction, Inactive, SystemException;
```

Description The `rollback_only()` function marks the transaction associated with the calling thread for rollback. The transaction is modified so that outcome must be that the transaction is rolled back. Any participant in the transaction can mark the transaction for rollback. The transaction is not rolled back until the participant that created the transaction either commits or aborts the transaction.

If no transaction is associated with the calling thread, the `NoTransaction` exception is thrown. If the transaction is already prepared, the `Inactive` exception is thrown.

The `Current.rollback()` function can be called instead of `rollback_only()`. Calling `Current.rollback()` rolls back the transaction immediately, preventing unnecessary work from being done between the time the transaction is marked for rollback and the time the transaction is actually rolled back.

See Also “Current.rollback()” on page 248

“Inactive” on page 268

“NoTransaction” on page 268Current.rollback()

Current.set_timeout()

Synopsis

```
static void set_timeout(int seconds)
   throws SystemException;
```

Parameters The `seconds` parameter specifies the number of seconds that the transaction waits for completion before rolling back.

Description The `set_timeout()` member function sets a timeout period for subsequent transactions begun with the `Current.begin()` function. (Transactions already underway are not affected.) The `seconds` parameter sets the number of seconds from the time the transaction is begun that it waits for completion before being rolled back; if the `seconds` parameter is set to zero, no timeout is set for the transaction.

Current.suspend()

Synopsis

```
Control suspend()
   throws SystemException;
```

Description The `suspend()` member function suspends the transaction associated with the calling thread. It returns the control object for the current transaction. This control object can be passed to the `Current.resume()` function to resume the suspended transaction. If there is no current transaction, this function returns a null object reference.

See Also "`Current.resume()`" on page 248

Control Class

Synopsis

```
public interface Control
    extends org.omg.CORBA.Object
{
    public org.omg.CosTransactions.Terminator get_terminator()
        throws org.omg.CosTransactions.Unavailable;
    public org.omg.CosTransactions.Coordinator get_coordinator()
        throws org.omg.CosTransactions.Unavailable;
    public org.omg.CosTransactions.Control get_parent()
        throws org.omg.CosTransactions.NotSubtransaction;
    public org.omg.CosTransactions.Control get_top_level()
        throws org.omg.CosTransactions.NotSubtransaction;
    public int id();
    public void id(int value);
}
```

Description The `Control` class enables explicit control of a factory-created transaction; the factory creates a transaction and returns a `Control` instance associated with the transaction. The `Control` object provides access to the `Coordinator` and `Terminator` objects used to manage and complete the transaction. A `Control` object can be used to propagate a transaction context explicitly.

Class Members

`Control.get_coordinator()`
`Control.get_parent()`
`Control.get_terminator()`
`Control.get_top_level()`
`Control.id()`
`Control.id(int value)`

See Also

"Control Class" on page 250
"Coordinator Class" on page 253
"Terminator Class" on page 262

Control.get_coordinator()

Synopsis

```
_CoordinatorRef get_coordinator()
   throws org.omg.CosTransactions.Unavailable;
```

Description

The `get_coordinator()` function returns the `Coordinator` object for the transaction with which the `Control` object is associated. The returned `Coordinator` object can be used to determine the status of the transaction, determine the relationship between the associated transaction and other transactions, create subtransactions, and so on.

The `get_coordinator()` function throws the `Unavailable` exception if the Coordinator associated with the `Control` object is not available.

See Also

“`Control Class`” on page 250

“`Coordinator Class`” on page 253

“`Unavailable`” on page 269

Control.get_parent()

Synopsis

```
_ControlRef get_parent()
   throws org.omg.CosTransactions.NotSubtransaction;
```

Description

Returns the `Control` object for the parent of the transaction with which the `Control` object is associated. If the associated transaction is not a subtransaction, the `NotSubtransaction` exception is thrown.

See Also

“`Control Class`” on page 250

“`NotSubtransaction`” on page 268

Control.get_terminator()

Synopsis

```
_TerminatorRef get_terminator()
   throws org.omg.CosTransactions.Unavailable;
```

Description

The `get_terminator()` function returns the `Terminator` object for the transaction with which the `Control` object is associated. The returned `Terminator` object can be used to either commit or roll back the transaction.

The `get_terminator()` function throws the `Unavailable` exception if the Terminator associated with the `Control` object is not available.

See Also

"`Control Class`" on page 250

"`Terminator Class`" on page 262

"`Unavailable`" on page 269

Control.get_top_level()

Synopsis

```
_ControlRef get_top_level()
   throws org.omg.CosTransactions.NotSubtransaction
```

Description

The `get_top_level()` function returns the `Control` object for the top-level ancestor of the transaction with which the `Control` object is associated. If the associated transaction is not a subtransaction, the `NotSubtransaction` exception is thrown.

See Also

"`Control Class`" on page 250

"`NotSubtransaction`" on page 268

Control::id()

Synopsis

```
public int id();
```

Description

The `id()` member function returns the transaction identifier for the transaction with which the `Control` object is associated.

Notes

This function is specific to OrbixOTS and is not a standard CORBA function.

The `id()` function is an OrbixOTS extension to the OMG OTS interface. The return value can be used to display the identity of the transaction associated with the `Control` object.

See Also

Other `id` constructor

Control::id()

Synopsis `public int id(int value);`

Parameters The `value` parameter is the transaction identifier for the transaction with which the `Control` object is associated.

Description The `id()` member function sets the transaction identifier for the transaction with which the `Control` object is associated.

Notes This function is specific to OrbixOTS and is not a standard CORBA function.

The `id()` function is an OrbixOTS extension to the OMG OTS interface.

See Also Other `id` constructor

Coordinator Class

Synopsis

```
public class Coordinator {
   public int get_status();
   public int get_parent_status();
   public int get_top_level_status();
   public boolean is_same_transaction(Coordinator tc);
   public boolean is_related_transaction(Coordinator tc);
   public boolean is_ancestor_transaction(Coordinator tc);
   public boolean is_descendant_transaction(Coordinator tc);
   public boolean is_top_level_transaction();
   public int hash_transaction();
   public int hash_top_level_tran();
   public String get_transaction_name();
   public _RecoveryCoordinatorRef
      register_resource(_ResourceRef r);
   public void register_synchronization(_SynchronizationRef sync);
   public void
      register_subtran_aware(_SubtransactionAwareResourceRef r);
   public _ControlRef create_subtransaction();
   public void rollback_only();
   public _PropagationContextRef get_txcontext();
};
```

Description The Coordinator class enables explicit control of a factory-created transaction. The factory creates a transaction and returns a Control instance associated with the transaction. The Control.get_coordinator() function returns the Coordinator object used to manage the transaction.

The operations defined by the Coordinator class can be used by the participants in a transaction to determine the status of the transaction, determine the relationship of the transaction to other transactions, mark the transaction for rollback, and create subtransactions. The Coordinator class also defines operations for registering resources as participants in a transaction and registering subtransaction-aware resources with a subtransaction.

Class Members

```
Coordinator.create_subtransaction()
Coordinator.get_parent_status()
Coordinator.get_status()
Coordinator.get_top_level_status()
Coordinator.get_transaction_name()
Coordinator.get_txcontext();
Coordinator.hash_top_level_tran()
Coordinator.hash_transaction()
Coordinator.is_ancestor_transaction()
Coordinator.is_descendant_transaction()
Coordinator.is_related_transaction()
Coordinator.is_same_transaction()
Coordinator.is_top_level_transaction()
Coordinator.register_resource()
Coordinator.register_subtran_aware()
Coordinator.register_synchronization();
Coordinator.rollback_only()
```

See Also "Control Class" on page 250

"Terminator Class" on page 262

"Control.get_coordinator()" on page 251

Coordinator.create_subtransaction()

Synopsis

```
org.omg.CosTransactions.Control create_subtransaction()
throws org.omg.CosTransactions.SubtransactionsUnavailable,
       org.omg.CosTransactions.Inactive;
```

Description The `create_subtransaction()` member function creates a new subtransaction for the transaction associated with the `Coordinator` object. A subtransaction is one that is embedded within another transaction; the transaction within which the subtransaction is embedded is referred to as its parent. A transaction that has no parent is a top-level transaction.

A subtransaction executes within the scope of its parent transaction and can be used to isolate failures; if a subtransaction fails, only the subtransaction is rolled back. If a subtransaction commits, the effects of the commit are not permanent until the parent transaction commits. If the parent transaction rolls back, the subtransaction is also rolled back.

If the parent transaction is already rolled back when `create_subtransaction()` is called, the `SubtransactionsUnavailable` exception is thrown. The `create_subtransaction()` function throws the `Inactive` exception if the transaction is already prepared.

Return Values The `create_subtransaction()` function returns the `Control` object associated with the new subtransaction.

See Also "`Control Class`" on page 250

"`Inactive`" on page 268

"`SubtransactionsUnavailable`" on page 269

Coordinator.get_parent_status()

Synopsis `public org.omg.CosTransactions.Status get_parent_status();`

Description The `get_parent_status()` function returns the status of the parent of the transaction associated with the `Coordinator` object. See the `Coordinator.create_subtransaction()` function reference page for more information.

Return Values The status returned indicates which phase of processing the transaction is in. If the transaction associated with the `Coordinator` object is a subtransaction, the status of its parent transaction is returned. If there is no parent transaction, the status of the transaction associated with the `Coordinator` object itself is returned.

See Also "`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.get_status()`" on page 256

"Coordinator.get_top_level_status()" on page 256

"Status Enumeration Class Type" on page 265

Coordinator.get_status()

Synopsis `public org.omg.CosTransactions.Status get_status();`

Description The `get_status()` function returns the status of the transaction associated with the `Coordinator` object. The status returned indicates which phase of processing the transaction is in. See the reference page for the `Status` type for information about the possible status values.

See Also "Coordinator.get_parent_status()" on page 255

"Coordinator.get_top_level_status()" on page 256

"Status Enumeration Class Type" on page 265

Coordinator.get_top_level_status()

Synopsis `public org.omg.CosTransactions.Status get_top_level_status();`

Description The `get_top_level_status()` function returns the status of the top-level ancestor of the transaction associated with the `Coordinator` object. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

The status returned indicates which phase of processing the transaction is in. See the reference page for the `Status` type for information about the possible status values. If the transaction associated with the `Coordinator` object is the top-level transaction, its status is returned.

See Also "Coordinator.create_subtransaction()" on page 254

"Coordinator.get_status()" on page 256

"Coordinator.get_parent_status()" on page 255

"Status Enumeration Class Type" on page 265

Coordinator.get_transaction_name()

Synopsis

```
public String get_transaction_name();
```

Description

The `get_transaction_name()` function returns the name of the transaction associated with the `Coordinator` object.

Coordinator::get_txcontext()

Synopsis

```
public org.omg.CosTransactions.PropagationContext get_txcontext()
        throws org.omg.CosTransactions.Unavailable;
```

Description

Returns the propagation context object which is used to export the current transaction to a new transaction service domain. The exception `Unavailable` is raised if the propagation context is unavailable.

See Also

"`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.get_status()`" on page 256

"`Coordinator.get_top_level_status()`" on page 256

"`Status Enumeration Class Type`" on page 265

`Unavailable` exception

Coordinator.hash_top_level_tran()

Synopsis

```
public int hash_top_level_tran();
```

Description

The `hash_top_level_tran()` function returns a hash code for the top-level ancestor of the transaction associated with the `Coordinator` object. If the transaction associated with the `Coordinator` object is the top-level transaction, its hash code is returned. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

The returned hash code is typically used as an index into a table of `Coordinator` objects. The low-order bits of the hash code can be used to hash into a table with a size that is a power of two.

See Also

"`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.hash_transaction()`" on page 258

Coordinator.hash_transaction()

Synopsis

```
public int hash_transaction();
```

Description

The `hash_transaction()` function returns a hash code for the transaction associated with the `Coordinator` object.

The returned hash code is typically used as an index into a table of `Coordinator` objects. The low-order bits of the hash code can be used to hash into a table with a size that is a power of two.

See Also

"`Coordinator.hash_top_level_tran()`" on page 257

Coordinator.is_ancestor_transaction()

Synopsis

```
public boolean
  is_ancestor_transaction(org.omg.CosTransactions.Coordinator tc);
```

Parameters

The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description

The `is_ancestor_transaction()` function determines whether the transaction associated with the `Coordinator` object is an ancestor of the transaction associated with the coordinator specified in the tc parameter. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

Return Values

The `is_ancestor_transaction()` function returns `true` if the transaction is an ancestor or if the two transactions are the same; otherwise, the function returns `false`.

See Also

"`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.is_descendant_transaction()`" on page 259

"`Coordinator.is_related_transaction()`" on page 259

"`Coordinator.is_same_transaction()`" on page 260

"`Coordinator.is_top_level_transaction()`" on page 260

"`Status Enumeration Class Type`" on page 265

Coordinator.is_descendant_transaction()

Synopsis

```
boolean
is_descendant_transaction(org.omg.CosTransactions.Coordinator tc);
```

Parameters The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description The `is_descendant_transaction()` function determines whether the transaction associated with the `Coordinator` object is a descendant of the transaction associated with the coordinator specified in the `tc` parameter. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

Return Values The `is_descendant_transaction()` function returns `true` if the transaction is a descendant or if the two transactions are the same; otherwise, the function returns `false`.

See Also

"`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.is_ancestor_transaction()`" on page 258

"`Coordinator.is_related_transaction()`" on page 259

"`Coordinator.is_same_transaction()`" on page 260

"`Coordinator.is_top_level_transaction()`" on page 260

"`Status Enumeration Class Type`" on page 265

Coordinator.is_related_transaction()

Synopsis

```
boolean
   is_related_transaction(org.omg.CosTransactions.Coordinator tc);
```

Parameters The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description The `is_related_transaction()` function determines whether the transaction associated with the `Coordinator` object and the transaction associated with the coordinator specified in the `tc` parameter have a common ancestor. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

Return Values The `is_related_transaction()` function returns `true` if both transactions are descendants of the same transaction; otherwise, the function returns `false`.

See Also
"`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.is_ancestor_transaction()`" on page 258

"`Coordinator.is_descendant_transaction()`" on page 259

"`Coordinator.is_same_transaction()`" on page 260

"`Coordinator.is_top_level_transaction()`" on page 260

Coordinator.is_same_transaction()

Synopsis

```
boolean
   is_same_transaction(org.omg.CosTransactions.Coordinator tc);
```

Parameters The `tc` parameter specifies the coordinator of another transaction to compare with the `Coordinator` object.

Description The `is_same_transaction()` function determines whether the transaction associated with the `Coordinator` object and the transaction associated with the coordinator specified in the `tc` parameter are the same transaction.

Return Values The `is_same_transaction()` function returns `true` if the transactions associated with the two `Coordinator` objects are the same transaction; otherwise, the function returns `false`.

See Also
"`Coordinator.is_ancestor_transaction()`" on page 258

"`Coordinator.is_related_transaction()`" on page 259

"`Coordinator.is_descendant_transaction()`" on page 259

"`Coordinator.is_top_level_transaction()`" on page 260

Coordinator.is_top_level_transaction()

Synopsis

```
boolean is_top_level_transaction();
```

Description The `is_top_level_transaction()` function determines whether the transaction associated with a `Coordinator` object is a top-level transaction. See the reference page for the `Coordinator.create_subtransaction()` function for more information.

Return Values The `is_top_level_transaction()` function returns `true` if the transaction is a top-level transaction; otherwise, the function returns `false`.

See Also "`Coordinator.create_subtransaction()`" on page 254

"`Coordinator.is_ancestor_transaction()`" on page 258

"`Coordinator.is_descendant_transaction()`" on page 259

"`Coordinator.is_same_transaction()`" on page 260

Coordinator::register_synchronization()

Synopsis

```
public void
  register_synchronization(org.omg.CosTransactions.Synchronization
                                sync)
  throws org.omg.CosTransactions.Inactive;
```

Parameters The `sync` parameter specifies the synchronization object to register.

Description The `register_synchronization()` member function registers a specified synchronization object for the transaction associated with a Coordinator object. See the reference page for the `Synchronization` class for more information.

The `register_synchronization()` function throws the `Inactive` exception if the transaction is already prepared. It throws the `CORBA::TRANSACTION_ROLLEDBACK` exception if the transaction is marked for rollback only.

See Also "`Inactive`" on page 268

Coordinator.rollback_only()

Synopsis

```
public void rollback_only()
        throws org.omg.CosTransactions.Inactive;;
```

Description The `rollback_only()` function marks the transaction associated with the `Coordinator` object so that the only possible outcome for the transaction is to roll back. The transaction is not rolled back until the participant that created the transaction either commits or aborts the transaction.

The `rollback_only()` function throws the `Inactive` exception if the transaction is already prepared.

The `Terminator.rollback()` function can be called instead of `rollback_only()`. Calling `Terminator.rollback()` rolls back the transaction immediately, preventing unnecessary work from being done between the time the transaction is marked for rollback and the time the transaction is actually rolled back.

See Also "`Inactive`" on page 268

"`Terminator.rollback()`" on page 263

Terminator Class

Synopsis

```
public class Terminator
implements COM.Transarc.CosTransactions._TerminatorRef {
public:
   void commit(boolean);
   void rollback();
};
```

Description The `Terminator` class enables explicit termination of a factory-created transaction. The transaction with which the `Terminator` object is associated can be either committed or rolled back. The `Control.get_terminator()` function can be used to return the `Terminator` object associated with a transaction.

Class Members

`Terminator.commit()`
`Terminator.rollback()`

See Also "`Control Class`" on page 250

"`Control.get_terminator()`" on page 251

"`Coordinator Class`" on page 253

Terminator.commit()

Synopsis

```
void commit(boolean report_heuristics)
   throws org.omg.CosTransactions.HeuristicMixed,
          org.omg.CosTransactions.HeuristicHazard;
```

Parameters The `report_heuristics` parameter specifies whether heuristic decisions are to be reported for the commit.

Description The `commit()` member function attempts to commit the transaction associated with the `Terminator` object. If the `report_heuristics` parameter is `true`, the `HeuristicHazard` or `HeuristicMixed` exception is thrown when the participants report that a heuristic decision has possibly been made.

See Also

"`Terminator Class`" on page 262

"`HeuristicHazard`" on page 267

"`HeuristicMixed`" on page 267

"`Coordinator Class`" on page 253

"`Terminator.rollback()`" on page 263

Terminator.rollback()

Synopsis

```
void rollback();
```

Description The `rollback()` member function rolls back the transaction associated with the `Terminator` object.

See Also

"`Terminator Class`" on page 262

"`Coordinator Class`" on page 253

"`Terminator.commit()`" on page 263

TransactionalObject Base Class

Synopsis

```
class TransactionalObject {};
```

Description The `TransactionalObject` class is the base class for all transactional objects. If an object's interface is derived from this class, the object behaves transactionally. Requests to a transactional object propagate the transaction context of the current thread to the object; that is, the requested operation is executed within the scope of the transaction. If a request is sent to a `TransactionalObject` and there is no current transaction, the `TRANSACTION_REQUIRED` exception is thrown.

See Also "`Control Class`" on page 250

"`Terminator Class`" on page 262

TransactionFactory Class

Synopsis

```
public interface TransactionFactory
    extends org.omg.CORBA.Object {
   public org.omg.CosTransactions.Control create(int);
};
```

Description The `TransactionFactory` class represents a transaction factory that allows the originator of transactions to begin a new transaction for use with the explicit model of transaction demarcation. OrbixOTS C++ Servers provide a default instance of this class. Clients can bind to the default instance by using the standard binding mechanism for the object request broker.

Class Members

`TransactionFactory.create()`

See Also "`Control Class`" on page 250

"`TransactionFactory Class`" on page 264

TransactionFactory.create()

Synopsis

```
_ControlRef create(int time_out)
   throws SystemException;
```

Parameters The `time_out` parameter specifies the number of seconds that the transaction waits to complete before rolling back.

Description The `create()` function creates a new top-level transaction for use with the explicit model of transaction demarcation. A `Control` object is returned for the transaction. The `Control` object can be used to propagate the transaction context. See the reference page for the `Control` class for more information. The `time_out` parameter sets the number of seconds that the transaction waits for completion before being rolled back; if the `time_out` parameter is zero, no timeout is set for the transaction.

See Also “`Control Class`” on page 250

“`TransactionFactory Class`” on page 264

Status Enumeration Class Type

Synopsis

```
public class Status{
   public static final status StatusActive
   public static final status StatusMarkedRollback
   public static final status StatusPrepared
   public static final status StatusCommitted
   public static final status StatusRolledBack
   public static final status StatusUnknown
   public static final status StatusNoTransaction
   public static final status StatusPreparing
   public static final status StatusCommitting
   public static final status StatusRollingBack
};
```

Constants

`StatusActive`	Indicates that processing of a transaction is still in progress.
`StatusMarkedRollback`	Indicates that a transaction is marked to be rolled back.
`StatusPrepared`	Indicates that a transaction has been prepared but not completed.
`StatusCommitted`	Indicates that a transaction has been committed and the effects of the transaction have been made permanent.

`StatusRolledBack`	Indicates that a transaction has been rolled back.
`StatusUnknown`	Indicates that the status of a transaction is unknown.
`StatusNoTransaction`	Indicates that a transaction does not exist in the current transaction context.
`StatusPreparing`	Indicates that a transaction is preparing to commit.
`StatusCommitting`	Indicates that a transaction is in the process of committing.
`StatusRollingBack`	Indicates that a transaction is in the process of rolling back.

Description The `Status` class defines values that are used to indicate the status of a transaction. Status values are used in both the implicit and explicit models of transaction demarcation defined by the Object Transaction Service (OTS). The `Current.get_status()` function can be called to return the transaction status if the implicit model is used. The `Coordinator.get_status()` function can be called to return the transaction status if the explicit model is used.

See Also "`Coordinator.get_status()`" on page 256

"`Current.get_status()`" on page 246

Common Exceptions

Exceptions are defined as classes and have the following form:

```
package org.omg.CosTransactions;
class ExceptionName {};
```

The exceptions are shown here in two tables: one for the OrbixOTS exceptions and another for the system exceptions:

Exception	Description
`HeuristicCommit`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that all updates have been committed.

`HeuristicHazard`	This exception is thrown to report that a heuristic decision has possibly been made by one or more participants in a transaction and the outcome of all participants in the transaction is unknown. See Also:
	`Current.commit()`
	`Terminator.commit()`

`HeuristicMixed`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that some updates have been committed and others rolled back. See Also:
	`Current.commit()`
	`Terminator.commit()`

`HeuristicRollback`	This exception is thrown to report that a heuristic decision was made by one or more participants in a transaction and that all updates have been rolled back. See Also:
	`Current.commit()`
	`Terminator.commit()`

Table 13.1: *OrbixOTS Exceptions for Java*

Exception	Description

`Inactive`	This exception is thrown when a transactional operation is requested for a transaction, but that transaction is already prepared. See Also:
	`Coordinator.create_subtransaction()`
	`Coordinator.register_resource()`
	`Coordinator.register_subtran_aware()`
	`Coordinator.rollback_only()`

`InvalidControl`	This exception is thrown when an invalid `Control` object is used in an attempt to resume a suspended transaction. See Also:
	`Control class`
	`Current.resume()`

`NotPrepared`	This exception is thrown when an operation (such as a commit) is requested for a resource, but that resource is not prepared. See Also:
	`Current.commit()`
	`Terminator.commit()`

`NoTransaction`	This exception is thrown when an operation is requested for the current transaction, but no transaction is associated with the client thread. See Also:
	`Current.commit()`
	`Current.rollback()`
	`Current.rollback_only()`

`NotSubtransaction`	This exception is thrown when an operation that requires a subtransaction is requested for a transaction that is not a subtransaction. See Also: `Control.get_parent()`

Table 13.1: *OrbixOTS Exceptions for Java*

Exception	Description

`SubtransactionsUnavailable`	This exception is thrown when an operation that is intended for subtransactions only is requested, but the transaction service does not support nested transactions. This exception is also thrown if an application attemps to create subtransaction after the parent transaction is already prepared. See Also:
	`Coordinator.create_subtransaction()`
	`Current.begin()`

`Unavailable`	This exception is thrown when a `Terminator` or `Coordinator` object cannot be provided by a `Control` object due to environment restrictions. See Also:
	`Control.get_coordinator()`
	`Control.get_terminator()`

Table 13.1: *OrbixOTS Exceptions for Java*

The following table shows the system exceptions that may be thrown:

Exception	Description
`INVALID_TRANSACTION`	This exception is thrown when the transaction context is invalid for a request.
`TRANSACTION_REQUIRED`	This exception is thrown when a null transaction context is associated with the client thread, and a transactional operation is requested.

`TRANSACTION_ROLLEDBACK`	This exception is thrown when a transactional operation (such as a `commit()`) is requested for a transaction that has been rolled back or marked for rollback. See Also:
	`Current::commit()`
	`Terminator::commit()`

Table 13.2: *System Exceptions*

14

Threading Transactions

The TranPthread class allows you to create multiple threading in transactions.

You can build concurrent transaction models using threads in OrbixOTS with the `TranPthread` class. This class allows you to start threads that can either join an existing transaction or run in an new top-level or nested transaction. For more information, see “Threads and Transactions” on page 72.

TranPthread Class

Synopsis

```
class TranPthread
{
   public:
      void
      Create(
         void* (*start_func)(void *),
         void* arg,
         int start_new_tran = 0
      );
      void
      Background(
         void* (*start_func)(void *),
         void* arg,
         int start_new_tran = 0
      );
      void*
      Join();
```

```
};
```

Description The `TranPthread` class provides a means for programmer to create threads which either participate in an existing transaction of run in a new top-level or nested transaction.

Class Members

```
TranPthread::Create()
TranPthread::Background()
TranPthread::Join()
```

TranPthread::Create()

Synopsis

```
void Create(void* (*start_func)(void*), void* arg, int
        start_new_tran =0);
```

Parameters

`start_func`	A pointer to a function where the thread starts.This function takes a single parameter of type `void*` and also returns a value of type `void*`.
`arg`	This is the value that is passed to the thread's start function pointed to by `start_func`.
`start_new_tran`	This indicates whether a new transaction is to be created or not.

Description `TranPthread::Create()` creates a new thread that starts execution at the function pointed to by the `start_func` parameter. The argument to the function is passed the value of the second paramter `arg`. The `start_new_tran` parameter indicates whether a new transaction is to be created for the new thread. If this parameter is zero, the thread joins the current transaction if any. If it is non-zero then a new transaction is created for the thread. If the thread terminates normally, the new transaction is committed. If the caller is already in a transaction then a nested transaction is created for the new thread; otherwise a top-level transaction is created. The `Join()` operation can be used to wait for the thread to complete and to retrieve the return value of the thread's start function.

See Also `TranPthread::Background()`, `TranPthread::Join()`

TranPthread::Background()

Synopsis

```
void Background(void* (*start_func)(void*), void* arg,
        int start_new_tran = 0);
```

Parameters

`start_func`	A pointer to a function where the thread starts. This function takes a single parameter of type `void*` and also returns a value of type `void*`.
`arg`	This is the value that is passed to the thread's start function pointed to by `start_func`.
`start_new_tran`	This indicates whether a new transaction is to be created or not.

Description `TranPthread::Background()` creates a new detached thread that starts execution at the function pointed to by the `start_func` parameter. The argument to the function is passed the value of the second paramter `arg`. The `start_new_tran` parameter indicates whether a new transaction is to be created for the new thread. If this parameter is zero, the thread joins the current transaction if any. If it is non-zero, a new transaction is created for the thread.

When the thread terminates normally, the new transaction is committed. If the caller is already in a transaction, a nested transaction is created for the new thread; otherwise a top-level transaction is created.

Threads created using this operation are detached and the `Join()` operation cannot be used.

See Also `TranPthread::Create()`

TranPthread::Join()

Synopsis `void* Join();`

Description Waits for the thread to complete and returns the return value of the thread's start function. The `Join()` operation can only be used for threads created using the `Create()` operation.

See Also `TranPthread::Create()`

Appendix A
The DTP Reference Model

The X/Open company has defined a standard for Distributed Transaction Processing (DTP) systems called the DTP Reference Model. See Figure 14.1:

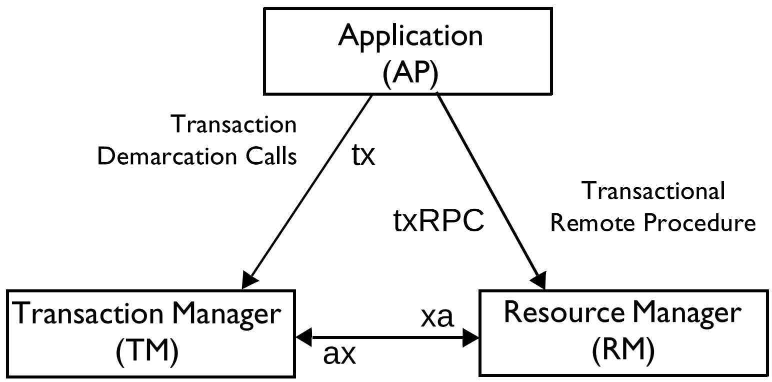


Figure 14.1: *The DTP Reference Model*

This model identifies the three components in a DTP scenario:

- The application (AP)
- The resource manager (RM)
- The transaction manager (TM)

The model also defines procedural interfaces between them: XA between transaction managers and resource managers, and TX between the application and the transaction manager.

The X/Open reference model is well established in industry. It specifies programming language interfaces between three identified entities engaged in a DTP system: the application, the resource manager and the transaction manager.

The application makes TX calls on the transaction manager to begin and complete global transactions, and makes transactional remote procedure calls (RPC) calls (using `txRPC()`) on resource managers in the context of the transaction. The transaction manager and resource managers communicate via the XA interface. In particular, this interface implements a two-phase commit protocol, facilitating atomic committal of global transactions.

Transactions are identified by an XID. This is a data structure that uniquely distinguishes the transaction in the system. Most functions in the reference model take an XID as a parameter, and transactional RPC calls propagate the XID implicitly.

In a typical scenario, some component of the application makes a `tx_begin()` call on the transaction manager and the calling thread is associated with the transaction. Subsequent `txRPC()` calls to resource managers carry knowledge of the transaction with them. Resource managers interested in transaction completion can be registered either statically or dynamically - that is, by the transaction manager calling `xa_start()` on the resource manager, or by the resource manager calling `ax_reg()` on the transaction manager. The transaction manager takes care of generating an appropriate XID for the transaction. The two-phase commit process is triggered by the application calling `tx_commit()` on the transaction manager, which coordinates completion by calling `xa_prepare()` and `xa_commit()` on each X/Open resource manager in turn.

An important point is that, with the exception of transactional RPC, these interfaces are defined at the programming language level only. That is, in the likely case that the entities communicating are distributed, the reference model does not indicate how the invocations should propagate between address spaces. X/Open compliant transaction managers and resource managers generally provide C libraries implementing these interfaces for linking with the relevant components, and use a variety of mechanisms to route, say, an `xa_commit()` call to a DBMS server process.

A number of third party transaction manager vendors support the TX interface, and most commercial database vendors provide an implementation of the XA interface. Prominent examples include Transarc's Encina, Oracle, Informix, and SQL Server.

Appendix B
The OrbixOTS Transaction Factory

This utility lets you develop Java applications without writing any C++ code.

The `otstf` utility is a standalone OrbixOTS server providing transaction and transactional lock set factory implementations. The server supports concurrent requests from multiple clients using the OrbixOTS configurable thread pools.

It is primarily intended for use with OrbixOTS for Java but it can be used any OTS client or server that must create transactions. A primary benefit of the utility is that it allows Java developers to write transactional Java applications without writing any C++ code.

The utility is intended to be self-managing by default, but it allows flexibility through the use of optional command line parameters.

Launching otstf

When launched persistently without parameters the server registers itself with the Orbix daemon as a persistent server using the name "`OrbixOTS_TransactionFactory`". It creates a transaction log, restart file and mirror restart file in the current directory using the names "`OrbixOTS_TF.log`" and "`OrbixOTS_TF.restart`" and "`OrbixOTS_TF.restartmirror`" respectively.

The server registers `TransactionFactory` and `TransactionalLockSetFactory` references in the NameService in the root context using the qualified names `OrbixOTS.TransactionFactory` and `OrbixOTS.LockFactory` respectively. As a result once you start the server persistently your transactional Java applications are able to use it.

Command-line Options

The following options allow you to override the default `otstf` behavior:

Command-line Option	Effect
`-f <LockSetFactory name>`	This option allows you to specify the name for the transaction lock set factory in the NameService. The name provides can be fully qualified in terms of naming contexts where contexts are delimited by a period (.). If the name includes naming contexts then the naming contexts must already exist.
`-h or ?`	Display help text.
`-i <id>`	This option allows you to specify the group identifier for this instance of the Transaction and Lock factories in an OrbixNames load balancing group. If this option is supplied then the `-t` and `-f` options must identify valid OrbixNames load balancing object groups. Using this option it is possible to distribute the load of transaction creation across a set of transaction factories.

`-l <device|file>`
`-r <file>`
`-m <file>`
	These options allow you to specify an alternative names for transaction log, restart and restart mirror files respectively. The supplied names can reference files created using the OrbixOTS `otsmklog` utility.

Table 14.2: *OrbixOTS Transaction Factory Command-line Options*

Command-line Option	Effect
`-s <name>`	This option allows you to specify an alternative name for the `otstf` server. This name should be registered with the Orbix daemon using the Orbix `putit` command line tool or the Orbix ServerManager GUI utility. This name is also used to scope SSL configuration information in the OrbixSSL configuration file. Using OrbixSSL with `otstf` is described on page 280.

`-T <transaction factory IOR file>`
 	
`-L <transactional lock set factory IOR file>`
	These options allow you to obtain the stringified object reference of the transaction and transactional lock set factories. In both cases you must supply the name of a file to which the IORs are written. Also the relevant factory will not be registered with the NameService.

`-t <TransactionFactory name>`	This option allows you to specify the name for the transaction factory in the Name Service. The name provided can be fully qualified in terms of naming contexts where contexts are delimited by a period(.). If the name includes naming contexts then the naming contexts must already exist. When OrbixOTS for Java needs to locate the default transaction factory it uses the OrbixOTS configuration value `OrbixOTS.OTS_DEFAULT_TRANSACTION_FACTORY_NS_NAME`. The default value for this variable matches the default values used by `otstf` when registering itself in the NameService. If you change this value using the `-t` option then you must change the `OrbixOTS.OTS_DEFAULT_TRANSACTION_FACTORY_NS_NAME` configuration variable to allow applications to continue finding the default transaction factory.

Table 14.2: *OrbixOTS Transaction Factory Command-line Options*

Command-line Option	Effect
-v	Display version information.

Table 14.2: *OrbixOTS Transaction Factory Command-line Options*

Using otstf with SSL

The otstf server supports SSL to allow it to operate in a secure environment. The otstf server is built as an OrbixSSL server - please refer to the O*rbixSSL Programmer's and Administrator's Guide* for general information on how to administer SSL applications.

A configuration variable must be set to allow OrbixSSL to locate the certificate file for otstf. The variable is called IT_CERTIFICATE_FILE and it must be set in the OrbixSSL configuration file in the appropriate scope for the server. The fully qualified scope is generated by appending the server's name to the string OrbixOTS.

Consider, for example, that the certificate for the otstf server is located in a file called c:\iona\config\repositories\certificates\services\orbixots\otstf.

In this case there should be a section in the OrbixSSL configuration file like this:

```
# OrbixOTS specific configuration information
OrbixOTS {
   OrbixOTS_TransacitonFactory {
   IT_CERTIFICATE_FILE="c:\iona\
config\repositories\certificates\services\orbixots\otstf";
}
}
```

In the above example the default server name OrbixOTS_TransactionFactory is used so the configuration scope is OrbixOTS.OrbixOTS_TransactionFactory. If you use the '-s' option to specify an alternative server name for a secure otstf server, there must be a corresponding configuration scope for that server name in the OrbixSSL configuration file.

OrbixSSL certificate files are usually password protected. In this case the server must obtain the password from some source in order to use its certificate. Generally, when SSL-enabled servers are launched automatically, OrbixSSL contacts the key distribution manager (KDM) server to obtain the password for the server's certificate.

In order to lauch the `otstf` server automatically, you must first use the OrbixSSL `putkdm` utility to record the `otstf` server certificate password in the KDM database. For example:

```
putkdm OrbixOTS_TransactionFactory demopassword
```

If the server is launched persistently and SSL is enabled, the server prompts you for a certificate password. When the password is correct the server continues, or otherwise it exits with an appropriate error message.

Use of otstf by OrbixOTS for Java

The OrbixOTS Java classes do not create or coordinate transactions. They therefore need to delegate these operations to a capable server. All recoverable OrbixOTS C++ servers are capable of creating and coordinating transactions because they each support a transaction factory object and maintain transaction logs to track the state of distributed transactions. The `otstf` server is just a specialization of such an OrbixOTS C++ recoverable server that has been extended to export its factory object references to OrbixNames and for ease of use.

The OrbixOTS Java classes use the `otstf` server to create and coordinate transactions by default. They resolve the default transaction factory name from the NameService. An application developer can specify an alternative default transaction factory using the `setDefaultFactory()` operation or through the OrbixOTS configuration value `OrbixOTS.OTS_DEFAULT_TRANSACTION_FACTORY_NS_NAME`.

Using a remote transaction factory in this manner incurs the overhead of remote invocations for transaction management so you must take care to minimize remote operations. This is particularly important when many resources are registered with a distribute transaction.

As an example, consider the scenario where a client invokes a transactional operation on a C++ server that registers a `CosTransactions::Resource`:

1. The Java client first creates a transaction using the standalone transaction factory (by default `otstf`) and propagates the transaction to the C++ server.
2. The server registers its resource with the transaction.
3. The client commits the transaction and the transaction coordinator (by default the first recoverable server – in this case is the otstf server) initiates two-phase commit (2PC) protocol.
4. Because the coordinator and resource are in different servers, remote invocations will be necessary that may accentuate the 2PC overhead for transaction completion.

To avoid this additional overhead the OrbixOTS classes provide a configurable optimization that delays transaction creation until the transaction is first propagated to a remote server. Before the remote operation an attempt will be made to use a transaction factory on the remote server to create the transaction. If this succeeds the performance benefit can be considerable. Consider the above example again. In this case the C++ recoverable server is responsible for coordination and it is co-located with the registered resources. Thus there will be no remote invocation overhead during the 2PC protocol.

If the attempt to create the transaction on the remote server fails because, for example, the server does not support the transaction factory interface, the default transaction factory creates the transaction and the operation proceeds as before. This optimization is off by default, but it can be enabled using the OrbixOTS configuration value:

```
OrbixOTS.OTS_USE_DEFAULT_FACTORY="FALSE"
```

This optimization is only useful when the remote servers are recoverable OrbixOTS C++ servers. The benefit is gained by minimizing the amount of remote operations required to implement 2PC.

The example describes above is a very simple scenario. If, in your case, the server that implements the most resources is not normally the first point of contact for a transaction you can use the `Client.setDefaultFactory()` operation to specify that server be used as the transaction factory. In this case you should leave the `OrbixOTS.OTS_USE_DEFAULT_FACTORY` configuration variable at it's default `TRUE` value to ensure that your specified default factory is always used.

Appendix C
OrbixOTS Configuration Variables

All OrbixOTS configuration variables are contained in the "OrbixOTS" configuration scope. This appendix describes these values.

Variable	Use	Value	Default
OTS_ABORT_TIMEOUT	The default timeout in seconds for transactions created without an explicit timeout.	int	180
OTS_ADMIN_TPOOL_HWM	Sets the high-water mark for the number of threads in the thread pool servicing administration requests sent by the otsadmin tool. The number of threads can never rise above this value.	int	10 x OTS_ADMIN_TPOOL_LWM
OTS_ADMIN_TPOOL_LWM	Sets the low-water mark for the number of threads in the thread pool servicing administration requests sent by the otsadmin tool.	int	5
OTS_ALWAYS_RETURN_CONTEXT	Sets whether a propagation context is always sent in the reply message of a transactional invocation.	bool	FALSE

Table 14.3: *OrbixOTS Configuration Variables*

Variable	Use	Value	Default
OTS_DEFAULT_TRANSACTION_ FACTORY_NS_NAME	The name used for the default transaction factory in the name service (used by Java OTS clients and servers).	string	"OrbixOTS.TransactionFactory"
OTS_GC_PERIOD	Sets the time in seconds between cleaning up caches used to store information required by the OTS.	int	300
OTS_LISTEN_TIMEOUT	Sets the timeout in milliseconds for servers calling the operation OrbixOTS::Server::impl_is_ready() with no timeout parameter. If no value is set the default Orbix	int	n/a
OTS_LOG_TPOOL_HWM	Sets the high-water mark for the number of threads in the thread pool servicing remote log requests (used by the OrbixOTS::Server::logServer() operation).	int	10 x OTS_LOG_TPOOL_LWM
OTS_LOG_TPOOL_LWM	Sets the low-water mark for the number of threads in the thread pool servicing remote log requests (used by the OrbixOTS::Server::logServer() operation). The number of threads can never rise above this value.	int	5

Table 14.3: *OrbixOTS Configuration Variables*

Variable	Use	Value	Default
OTS_NO_ABORT_ON_USER_ EXCEPTION	Whether raising a user exception does not cause the transaction to be rolled back.	bool	FALSE
OTS_NO_NICE_MESSAGES	Determines whether the user friendly interpretaion of Encina Toolkit errors is disabled.	bool	FALSE
OTS_NO_OPTIMIZE_PROPAGATION	Sets whether certain optimizations dealing with transaction context propagation are disabled.	bool	FALSE
OTS_NO_PING_DURING_BIND	Sets whether to enable or disable the Orbix ping-during-bind feature (see the Orbix operation pingDuringBind() for more details).	bool	FALSE
OTS_OOB_SYNCHRONOUS	Whether to disable the use of the thread pool for transaction protocol requests (also known as *out-of-band* or OOB requests).	bool	FALSE

Table 14.3: *OrbixOTS Configuration Variables*

Variable	Use	Value	Default
OTS_OOP_TPOOL_HWM	Sets the low-water mark for the number of threads in the thread pool servicing transaction protocol requests (also known as *out-of-band* or OOB requests). The number of threads can never rise above this value.	int	10 x OTS_OOP_TPOOL_LWM
OTS_OOP_TPOOL_LWM	Sets the low-water mark for the number of threads in the thread pool servicing transaction protocol requests (also known as *out-of-band* or OOB requests).	int	5
OTS_ORBIX_DIAGNOSTICS	Sets the Orbix diagnostics level using the setDiagnostics () operation.	int	0
OTS_ORBIX_DISPATCH_YIELD	Whether a thread yields before servicing a request from the user request thread pool.	bool	FALSE
OTS_RECOVERY_RETRY_TIMEOUT	Sets the time in seconds between attempts to complete Resource transaction protocol after failure.	int	180
OTS_SERVER_NAME	Sets the default name of the server to which otsadmin commands are directed.	string	n/a

Table 14.3: *OrbixOTS Configuration Variables*

Variable	Use	Value	Default
OTS_TPOOL_HWM	Sets the high-water mark for the number of threads in the thread pool servicing user requests. The number of threads can never rise above this value.	int	10 x OTS_TPOOL_LWM
OTS_TPOOL_LWM	Sets the low-water mark for the number of threads in the thread pool servicing user requests.	int	5
OTS_USE_DEFAULT_FACTORY	Controls whether the default transaction factory obtained from the name service is used when creating new transactions for Java clients and servers.	bool	TRUE

Table 14.3: *OrbixOTS Configuration Variables*

Index

H

I

S

T

U

V

W

X