ORBacus OTS

Version 1.0 Beta 2

CHAPTER 1

Introduction

1.1 What is ORBacus OTS?

ORBACUS OTS is an implementation of the Object Management Group (OMG) Transaction Service Version 1.2 draft specification[1] [5]. ORBACUS OTS provides a robust and full-featured platform on which to build large scale distributed applications that require absolute transactional integrity.

Some highlights of ORBACUS OTS are:

- Explicit and implicit transaction programming models
- Subtransactions
- X/Open DTP checked transaction support
- Interposition
- Synchronization support
- Seamless X/Open DTP XA integration with databases such as Oracle
- Advanced administrative support

1. The OMG OTS 1.1 specification [4] is being revised by the OMG OTS Revision Task Force. ORBACUS OTS is compliant with the current 1.2 draft [5]. The OMG Transaction Service Revision Task Force may continue to make changes to improve the specification. OOC, which is an active participant in the Revision Task Force, will continue to insure ORBACUS OTS is compliant to the latest standard.

ORBACUS OTS features the ORBACUS OTS Console, a graphical client for administering multiple OTS servers. It is written in Java for maximum portability. The ORBACUS OTS Console supports:

- Transaction service shutdown
- Querying complete details of active transactions
- Forcing transaction rollback
- Removing a resource from a transaction

1.2 About this Document

The ORBACUS OTS manual provides a brief overview of OMG Transaction Service concepts. It includes a tutorial transactional programming example with implementations in C++ and Java. Chapters on the configuration of ORBACUS OTS and the use of the ORBACUS OTS Console are provided.

Note: The tutorial transactional example will be included in a future release of the ORBACUS OTS *manual.*

This document does not serve as a detailed guide to transaction processing or as a substitute for the OMG Transaction Service specification. Please consult [5] for a detailed description of the Transaction Service. Other publications describing transaction processing concepts [1], [2], [8] are listed in Chapter 7.

1.3 Getting Help

Licensed, commercial users of ORBACUS OTS can obtain prompt, expert assistance by contacting OOC via email at `support@ooc.com`.

Since ORBACUS OTS is a member of the ORBACUS product family, you might also consider subscribing to our ORBACUS mailing list. Subscribers to this list receive announcements about bug fixes and updates. For more information on this mailing list, please see `http://www.ooc.com/ob/mailing-list.html`.

CHAPTER 2

Configuration and Startup

2.1 ORBacus OTS

2.1.1 Configuring a Transaction Domain

Before deploying applications using ORBACUS OTS, the OTS domain must be configured. The OTS domain is the set of OTS servers where transaction identifiers are guaranteed to be unique. In simple cases the domain will consist of only one OTS server.

Each server in a domain is given a unique name which is used in constructing the transaction ID[1]. For example, an organization might run several OTS servers based in different cities as shown below:

1. There are other ways of generating globally unique transaction IDs, however a clear text server name is of tremendous benefit when debugging or viewing transactions using administration tools.

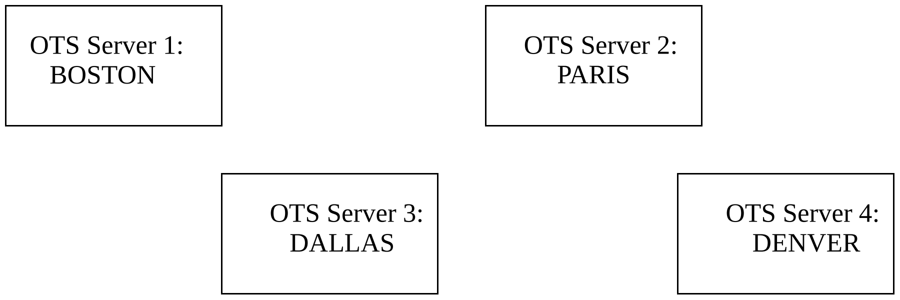


Figure 2.1: OTS Server Domain with 4 Servers

Each OTS server is named by the city it is running in. Applications can use any of the OTS servers by configuring their initial TransactionFactory reference. On the command line this is:

```
-ORBInitRef TransactionFactory=corbaloc::<OTSServer>:<OTSPort>/
TransactionFactory
```

where `<OTSServer>` and `<OTSPort>` are set to point at the appropriate server. Typically applications would be configured to use the OTS server closest[1] to them.

Interposition

If a transactional clients start transactions with one OTS server and then involves transactional objects that are closer to another OTS server in the domain, the OTS servers can be configured to use interposition. Interposition allows a local OTS server to create a proxy for a transaction that has originated at a remote OTS server. This will reduce network traffic to the remote originating OTS server.

For example, if a transaction is started using the `BOSTON` OTS server and involves several hundred transactional objects from an application server running in Denver there will be long distance network traffic for each transactional object. Interposition allows all of the Denver based transactional objects to interact with a local proxy from the `DENVER` OTS server instead of the transaction in Boston.

1. Closest in terms of network performance, not geographical distance.

Use of interposition is configured by setting the `ooc.ots.implicit.interposition` property, (discussed in section.2.1.4 on page 13.)

Dedicated Interposition Servers

In the above example, the interposition server for Denver applications is set to the `DENVER` OTS Server itself. The interposition server can also be set to any OTS server in a domain. It is then possible to configure OTS servers that are dedicated interposition servers and are not used for originating transactions themselves.

2.1.2 Synopsis

The OTS server creates and coordinates transactions. This server is also responsible for transaction recovery.

Usage

```
ots [-v,--version] [-h,--help] [-OTSserver_name ][-i,--ior] [-l,--
log-console]
```

Options:

`-v` `--version`	Reports the ORBACUS OTS version number.
`-h` `--help`	Displays `ots` command information.
`-OTSserver_name`	Set the OTS service name. This name must be unique within an OTS domain.
`-i` `--ior`	Prints IOR on standard output.
`-l` `--log-console`	Redirects logging output to the console instead of `<server_name>.log`.

2.1.3 Running OTS as an NT Native Service

Note: ORBACUS OTS *will be runnable as an NT Native Service in a future release. The documentation in this section is preliminary and subject to change.*

ORBACUS OTS is also available as a native Windows NT service.

```
nt-ots-service[-h,--help] [-i,--install] [-u,--uninstall]
              [-d,--debug]
```

Options:

`-h` `--help`	Displays command line options supported by the server.
`-i` `--install`	Install the service. The service must be started manually.
`-s` `--start-install`	Install the service. The service will be started automatically.
`-u` `--uninstall`	Uninstall the service.
`-d` `--debug`	Run the service in debug mode.

In order to use ORBACUS OTS as a native Windows NT service, it is first necessary to add the `OTSService` initial reference to the HKEY_LOCAL_MACHINE NT registry key (see "Using the Windows NT Registry" in the ORBACUS manual for more details).

Next the service is installed with:

```
nt-ots-service -i
```

This adds the ORBacus OTS entry to the Services dialog in the Control Panel. To start ORBACUS OTS, select the ORBacus OTS entry and press Start. If the service is to be started automatically when the machine is booted, select the ORBacus OTS entry, then click Startup. Next select Startup Type - Automatic, and press OK.

If you want to remove the service, run:

```
nt-ots-service -u
```

Note: If the executable for ORBACUS OTS *is moved, it must be uninstalled and re-installed.*

Any trace information provided by the service is placed in the Windows NT Event Viewer with the title OTSService.

2.1.4 Configuration Properties

OTS Server Properties

The following properties are used to configure the OTS server.

ooc.ots.port

Value: *0 < port <= 65535*

Specifies the port on which the OTS server `CosTransactions::TransactionFactory` listens for requests. If no value is selected a random port will be assigned by the operating system.

ooc.ots.server_name

Value: *`string`*

Specifies the name of the OTS server. Note that this can be overridden with the `-OTSserver_name` command line option. An OTS server cannot be started without specifying a server name. Every OTS server in a domain must have a unique name.

ooc.ots.log_to_console

Value: true, false

If true, the debug log will be displayed on the console and not in a file. The default value is false.

ooc.ots.working_dir

Value: *absolute directory path*

The directory in which the OTS database file and debug log are placed. The database file will be named ‘`<OTSNAME>`’. The debug log will be named ‘`<OTSNAME>`.log’. The default value is the current working directory. `<OSTNAME>` is the value of the **ooc.ots.server_name** property.

ooc.ots.trace.heuristics

Value: *`level >=0`*

Trace any transaction that results in a heuristic exception. The default value is 0, providing no trace.

ooc.ots.trace.recovery

Value: *level* >= 0

Log trace information on transaction recovery. The default value is 0, providing no trace.

ooc.ots.trace.sync

Value: *level* >= 0

Trace method invocations on `CosTransactions::Synchronization` objects. The default value is 0, providing no trace.

ooc.ots.trace.transactions

Value: *level* >= 0

Log trace information on transaction outcomes. The default value is 0, providing no trace.

ooc.ots.trace.resources

Value: *level* >= 0

Trace method invocations on `CosTransactions::Resource` objects. The default value is 0, providing no trace.

ooc.ots.recovery_timeout

Value: *seconds* >= 0

The number of seconds between calls to a superior coordinator for recovery and status verification. The default value is 60 seconds.

Implicit Propagation Properties

The following properties are used to configure the OTS client behavior in regard to implicit transactions.

ooc.ots.implicit.default_timeout

Value: *seconds* >= 0

The time-out value given to top-level transactions created by calling `CosTransactions::Current::begin`. The default value is 0, which means there is no time-out.

ooc.ots.implicit.checked_behavior

Value: true, false

If true, implicit transaction use will be subject to X/Open checked transaction behavior. In this case a transaction cannot be committed if there are still outstanding transactional requests associated with it. If the value is false, a transaction can be committed with associated transactional requests still in progress. The default value is true, providing checked behavior.

ooc.ots.implicit.subtransactions

Value: true, false

If true, the OTS client will allow the creation of nested transactions by calling `CosTransactions::Current::begin`. If false, attempting to create a subtransaction by calling `begin` will raise a `CORBA::NO_IMPLEMENT` exception. The default value is true, allowing subtransactions.

ooc.ots.implicit.interposition

Value: `CosTransactions::TransactionFactory` *URL*

If this property is set, the OTS client will use interposition to 'import' a transaction into the execution environment governed by the provided `CosTransactions::TransactionFactory` reference. It is legal to provide the same reference as for the "`TransactionFactory`" initial reference. The default is no URL, interposition is not used.

ooc.ots.implicit.trace.transactions

Value: *level* >= 0

This property controls the level of tracing information for implicit transactions. The default value is 0. The following values are defined. (A trace value includes the lower-level traces as well):

0 - No implicit transaction trace.

1 - Trace basic implicit transaction operations, showing transaction operations such as begin, commit, and rollback.

2 - Trace transactions associated with requests in the ots client and server side interceptors.

3 - Supplement the transaction trace with the outstanding checked request count.

4 - Provide all available transaction tracing.

X/Open XA Properties

Note: The XA Properties listed are subject to change

The following properties are used to configure the OTS client behavior in regard to XA X/Open transaction processing.

ooc.ots.xa.recovery_timeout

Value: *seconds* >= 0

The number of seconds between calls to the coordinator for recovery and status verification. The default value is 60 seconds.

ooc.ots.xa.working_dir

Value: *directory path*

The directory in which XA log files are placed. The log will be named '`xa-log-<RESOURCEMANAGERNAME>`'. The default value is the current working directory.

2.1.5 Connecting to the Service

The object key of ORBACUS OTS service is `TransactionFactory`, which identifies an object of type `CosTransactions::TransactionFactory`. The object key can be used when composing URL-style object references. For example, the following URL identifies the OTS service running on host `otshost` at port `10000`:

```
corbaloc::otshost:10000/TransactionFactory
```

2.2 ORBacus OTS Console

ORBACUS OTS Console is a graphical client to administer the OTS service.

2.2.1 Synopsis

```
java com.ooc.CosTransactionsConsole.Main [--service-file file-name]
```

2.3 Startup Example

The following is an example for how to start ORBACUS OTS and the ORBACUS OTS Console, using an ORBACUS configuration file. For more information on ORBACUS configuration files, please refer to the ORBACUS User's Guide [6]. Note that it is also possible to use command line parameters instead of configuration files.

Create a file with the following contents, and save it as `/tmp/ob.conf` (Unix) or `C:\temp\ob.conf` (Windows):

```
1 ooc.ots.server_name=<OTSServer>
2 ooc.ots.port=<OTSPort>
3 ooc.ots.working_dir=<database directory>
4 ooc.orb.service.TransactionFactory=corbaloc::<host>:<port>/
  TransactionFactory[1]
```

1 Specify the OTS server name.

2 Specify the OTS port

3 Specifies the path to the service's database directory. Replace <database directory> with the directory where the service should create its databases.

4 Provides a reference to the transaction factory. Replace `<host>` with your system's host name and `<port>` with an arbitrary, free TCP port (e.g. `10001`).

2.3.1 Starting ORBacus OTS

After ORBACUS OTS has been properly built and installed, there will be an ots executable in the installation target directory. For example, on UNIX, assuming the installation path was set to `/usr/local`, the executable is:

```
/usr/local/bin/ots
```

And on Windows, with the installation path set to `C:\OOC`:

```
C:\OOC\bin\ots.exe
```

You can start ORBACUS OTS as follows:

- Specify the configuration file on the command line:

  Unix

  ```
  /usr/local/bin/ots -ORBconfig /tmp/ob.conf
  ```

  Windows

  ```
  C:\OOC\bin\ots.exe -ORBconfig C:\temp\ob.conf
  ```

- Specify the configuration file with an environment variable:

1. Note that for typesetting reasons this configuration option spans two lines, but in your configuration file, this must be a single line.

Unix

```
ORBACUS_CONFIG=/tmp/ob.conf
export ORBACUS_CONFIG
/usr/local/bin/ots
```

Windows

```
set ORBACUS_CONFIG=C:\temp\ob.conf
C:\OOC\bin\ots.exe
```

2.3.2 Starting the ORBacus OTS Console

The Java archive `OBOTS.jar` contains the ORBACUS OTS Console. For example, on Unix, assuming the installation path was set to `/usr/local`, the archive can be found at:

```
/usr/local/lib/OTS.jar
```

And on Windows, assuming the installation path was set to `C:\OOC`:

```
C:\OOC\lib\OTS.jar
```

Note that the console application also requires `OB.jar`, `OBTime.jar`, and `OBUtil.jar` from the ORBACUS for Java distribution. Assuming these files are in the same directory as `OTS.jar`, the console can be started as follows:

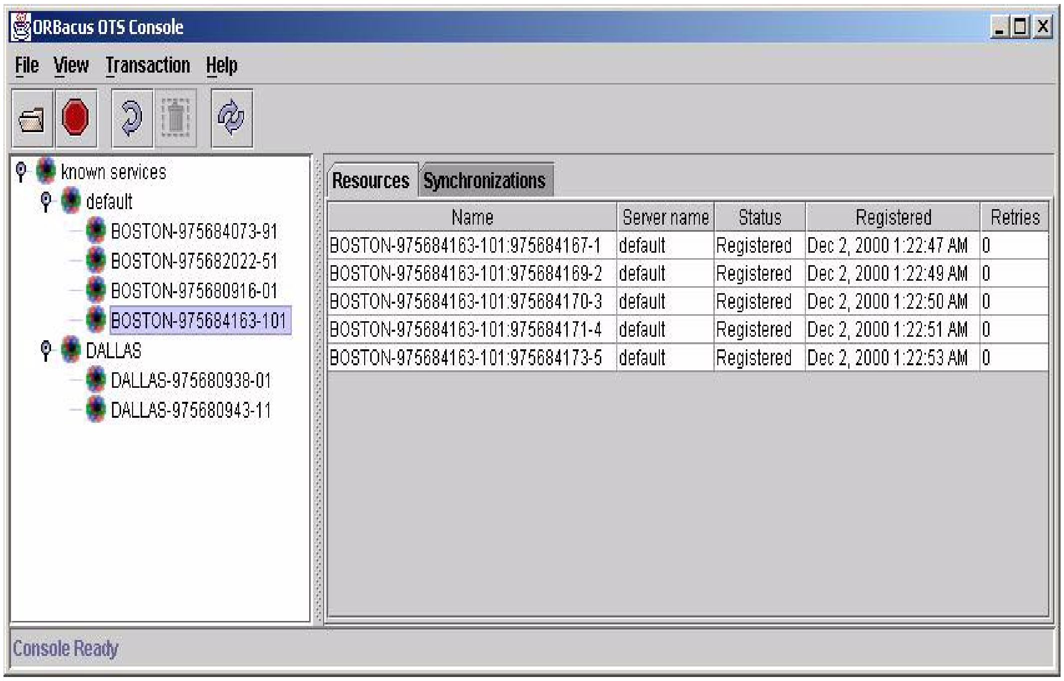
Unix

```
CLASSPATH=/usr/local/lib/OB.jar:/usr/local/lib/OBTime.jar:/usr/
local/lib/OBUtil.jar:/usr/local/lib/OTS.jar:$CLASSPATH

export CLASSPATH

java com.ooc.CosTransactionsConsole.Main -ORBconfig /tmp/ob.conf
```

Windows

```
set CLASSPATH=C:\OOC\lib\OB.jar;C:\OOC\lib\OBTime.jar;C:\OOC\lib\O
BUtil.jar;C:\OOC\lib\OTS.jar;%CLASSPATH%

java com.ooc.CosTransactionsConsole.Main -ORBconfig C:\temp\ob.conf
```

Figure 2.2: shows a screenshot of the OTS Console.



Name	Server name	Status	Registered	Retries
BOSTON-975684163-101:975684167-1	default	Registered	Dec 2, 2000 1:22:47 AM	0
BOSTON-975684163-101:975684169-2	default	Registered	Dec 2, 2000 1:22:49 AM	0
BOSTON-975684163-101:975684170-3	default	Registered	Dec 2, 2000 1:22:50 AM	0
BOSTON-975684163-101:975684171-4	default	Registered	Dec 2, 2000 1:22:51 AM	0
BOSTON-975684163-101:975684173-5	default	Registered	Dec 2, 2000 1:22:53 AM	0

Figure 2.2: ORBacus OTS Console

Specifying the OTS Domain

By default, the ORBACUS OTS Console displays transactions in the default OTS Server. This is the one configured to be the `TransactionFactory` initial reference.

To allow the display of other OTS servers in the OTS domain a service file must be provided that lists the OTS servers to display. This file consists of lines with:

```
<OTSServerName>=<OTSServer URL>
```

This file is then specified on the command line by:

--service-file <OTS Server File Name>

For example, if the initial TransactionFactory reference is the `BOSTON` OTS server and the ORBACUS OTS Console should display the `BOSTON` and `DENVER` OTS servers the service file would be:

```
#
# Line’s starting in ‘#’ are comments
#
DENVER=corbaloc::<DenverOTSHost>:<DenverOTSPort>/
TransactionFactory
```

CHAPTER 3 *Transaction Service Concepts*

This chapter provides an overview of transactions and their properties. It provides a functional overview of the OMG Object Transaction Service and ORBACUS OTS features. This chapter is not a substitute for reading the OTS specification [5].

3.1 Fundamentals

3.1.1 Transactions

A transaction is a logical unit of work that affects the state of an application. An example of a transaction is the electronic transfer of funds from a New York bank account to one in Paris as illustrated in Figure 3.1.

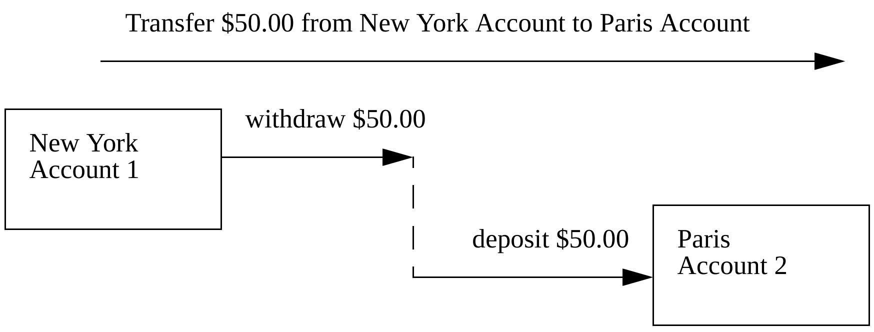


Figure 3.1: Transaction Example

The example transaction logically consists of one ‘transfer of funds’ operation but is composed of two component operations:

- Withdraw $50.00 from the New York account
- Deposit $50.00 to the Paris account

If both of these operations succeed, the ‘transfer of funds’ transaction completes normally. The transaction is now considered committed. A query of the two accounts balances will show that $50.00 has moved from New York to Paris. A permanent state change has been made to the accounts.

It is desirable for this transaction to always complete successfully, however in the real world there are many obstacles to the completion of the funds transfer. The withdrawal from the New York account can fail due to insufficient funds. The deposit to the Paris account can fail if a communications line is down or if the account itself had been closed out. In any of these cases the ‘transfer of funds’ transaction cannot be completed successfully. The transaction must be aborted (rolled back.)

For the example, it is crucial that if the transaction is rolled back the $50.00 transfer amount is accounted for. The money should return to the New York account. In fact, there should be no indication the money ever left New York. A query of the two account balances after an abort will show no funds have moved between the two accounts. The aborted transaction has made no state changes.

The transaction described here is a top-level, or flat transaction. There is another kind of transaction, the subtransaction, that represents a transaction nested within another transaction. This is described in “Nested Transactions” on page 28.

3.1.2 ACID Transaction Properties

The example above illustrates one important property of transactions, they are all or nothing affairs. Either every step in a transaction completes successfully (commit) or every step rolls back leaving no trace of a partially completed procedure. More formally stated, transactions must have the following properties:

- *Atomicity,* A transaction must behave as a logical, *atomic*, unit of work that results in a single state change to an external observer, no matter how many intermediate steps and participants are involved. If the transaction commits, the resulting state change is made visible to outside observers. If the transaction rolls back, an observer will see no state change. In the example, an outside observer will either see that $50.00 has been withdrawn from the New York account *and* deposited to the Paris account or they will see no movement of funds at all.
- *Consistency,* A transaction must not violate application integrity. In the example, the amount withdrawn from one account must match the amount deposited. It is the programmer's responsibility to ensure a transaction's logic does not violate the integrity of the application.
- *Isolation,* A transaction in progress is isolated from the rest of the system. No observer can see the intermediate steps of a transaction. In the example, no observer will see that $50.00 has been withdrawn from the New York account until they can also see that $50.00 has been deposited to the Paris account. The state change only becomes observable when the transaction commits. Isolation makes transactions, even if running in parallel, appear to be executing serially to outside observers.
- *Durability,* Once committed a transaction's results become part of the application's persistent state. If the application crashes after a transaction has committed, the state will be restored on restart.

These properties are known collectively as the *ACID* properties. These properties cannot be violated even if there is some form of system failure (process crash) during the life of a transaction.

3.2 The OMG Object Transaction Service (OTS)

In a CORBA environment, a transaction can involve any number of objects distributed across a network. Regardless of the number of objects involved or the physical locations of the participants, the transaction must still have *ACID* properties.

Managing transactions in a distributed environment is a complex task. Therefore applications rely on a distributed object transaction service to provide transactional capabilities. A transaction service provides an application programming interface (*API)* to begin, commit, rollback, and add participants to transactions. Most importantly a transaction service will guarantee *ACID* transactional integrity in the face of system failure[1]. ORBACUS OTS is an enhanced implementation of the OMG's Object Transaction Service (*OTS)* specification.

3.2.1 Transaction Life Cycle Overview

The diagram below illustrates the life of a 'funds transfer' transaction using the OTS. For the purposes of the illustration, all the components involved in the transaction are portrayed as separate entities although any of the components may be collocated.

.

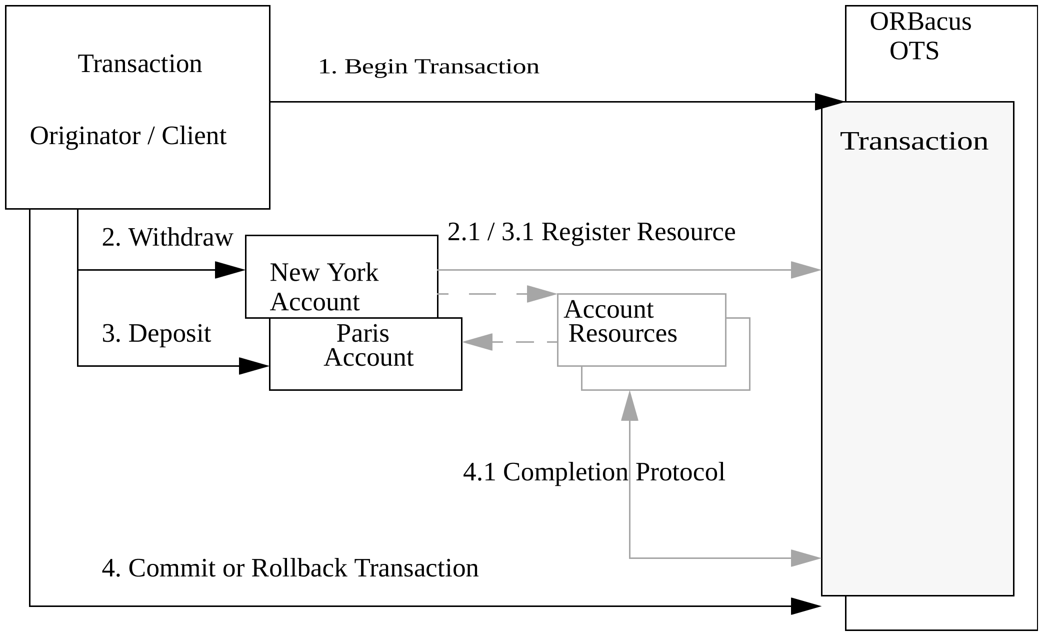


Figure 3.2: Transaction Use in the OTS

1. This guarantee must be qualified to exclude catastrophic failures such as any that result in the total physical destruction of the transaction manager's physical datastore. Some investment is generally made in fault-tolerant hardware and mirroring for mission critical TP systems.

Starting a Transaction

Transactional client applications are any client programs that make use of transactions. A client that sends a request to an OTS to begin a new transaction is known as the transaction originator, (Figure 3.2, Step 1.) Once created, a transaction is in the active state. It remains in the active state until the transaction is ended by the client or the OTS determines it has it timed out[1] and must be aborted.

Adding Resources to a Transaction

In order for a transaction to do any useful work, it must involve objects that can:

- Have a state change.
- Isolate the state change so that it is not visible outside of the transaction
- Either commit or forget the state change at the end of the transaction.

Objects that have these properties are called *transactional* objects.

Each transactional object is represented in the transaction by one or more `Resource`[2] objects, (Figure 3.2, Steps 2.1 and 3.1.) A `Resource` object participates in the transaction completion protocol. Specifically it must be capable of either permanently committing or rolling back the state changes made to an associated transactional object in the context of a transaction. In our banking example, a `Resource` for a bank account is capable of undoing any deposits or withdrawals that occurred in a transaction.[3]

All `Resources` that have been registered with a transaction are contacted by the OTS when a client decides to commit or rollback a transaction. If a client decides to commit, each `Resource` has the opportunity to veto the commit and force an abort. Transaction completion is the heart of the OTS and is described further in “Committing a Transaction” on page 26.

Resources can be written to support specific transactional objects. If a transactional object uses an X/Open DTP XA [9] compliant database such as Oracle or SQL Server for persis-

1. Transactions can be started with or without a timeout value. Clients that begin transactions with no timeout must be capable of ending them under all circumstances.
2. This refers to the `CosTransactions::Resource` interface.
3. At first glance it might appear that a transactional object such as a bank account should be the `Resource` object itself, however this is not the preferred approach. The transactional object and `Resource` object have a *uses*, not an *is-a*, relation. For example, client programs involved in a funds transfer transaction would communicate to a bank account object, which would then in turn register a private `Resource` object with the transaction service.

tent storage ORBACUS OTS can handle much of the `Resource` work for you. (See Chapter 5 for details.)

Performing Transactional Requests

Once a `Resource` is registered for a transactional object, the transactional object is ready for use in the transaction. From the client perspective, there is nothing special about an invocation on transactional objects as shown by the withdraw step 2 and the deposit step 3 in Figure 3.2.

Committing a Transaction

In most applications, a client will complete the transaction normally by asking the OTS server to commit the transaction, (Figure 3.2 Step 4.)

Unfortunately the OTS server cannot just send commit requests to each registered resource in the transaction and be done. As in the Section 3.1.1 example, there are any number of reasons why a bank account `Resource` may not be able to commit its portion of a transfer.

Two-Phase Commit

To allow for individual `Resources` being unable to commit their work, the OTS and the `Resources` take part in a completion protocol called the two-phase commit.

The basic idea of the two-phase commit is that the `Resources` vote to decide whether a transaction will actually commit or abort. If all of the `Resources` vote yes for the commit, then a confirming commit request is sent to each resource. If any `Resource` votes no, then the transaction will be aborted[1]. Even this protocol is not free of complications as a system failure can strike at any moment inside of this two-phase commit. However, recovery in most cases and administrative error reporting is always available when recovery is not possible. In no case will a transaction silently fail.[2]

Once the commit is complete, the transaction is finished and ceases to exist.

Rolling Back a Transaction

If a client decides to abort a transaction, it can ask the OTS server to erase, or roll back the transaction. The OTS server will then contact each of the registered `Resources` and tell

1. This description is an over simplification but conveys the intent.
2. Unless an application chooses to ignore the failure.

them to rollback. In the example from Section 3.1.1, a client could request the OTS server to rollback the funds transfer transaction. At that point the server would contact the bank accounts' `Resource` objects with a rollback request. Once the OTS server completes the rollback there is no trace that the transaction was ever initiated. The transaction is now finished and ceases to exist.

3.2.2 Implicit and Explicit Programming Models

Transactional clients can choose from two methods of interacting with the OTS and handling transactions.

The first method is called the implicit programming model. In this case the OTS maintains an implicit association between threads of execution and the current transaction. No references to the transaction are required in any IDL, so a transactional implementation's IDL can be identical to a non-transactional implementation's. The OTS is responsible for implicitly propagating the transaction context with method invocations. The OTS can also provide what is called *checked* behavior, doing some additional runtime checks to further ensure it is ok to commit a transaction. To provide transactional implementations of pre-existing IDL interfaces, implicit propagation must be used.

The other available method is the explicit programming model. Using this method, the client program must pass the transaction as a parameter to any objects that need it. This requires transactions to be passed as parameters in an object's interface IDL. There is nothing special about the explicit programming model, it is the normal method of using `CORBA`. It is only given a name to differentiate it from the implicit model.

One programming model should be used consistently throughout an application. It is possible to switch back and forth from implicit to explicit (with some limitations.)

It should be noted that once an ORB instance has been configured to provide implicit programming support, it is provided for every invocation. This includes invocations that use the explicit programming model, since these are indistinguishable from any other `CORBA` invocation.

3.2.3 Advanced ORBacus OTS Features

The next sections describe advanced features of the OTS. Although considered optional in the OMG specification, ORBACUS OTS fully supports the following features.

Nested Transactions

The discussion of transactions so far has been limited to single, top-level transactions. Using ORBACUS OTS it is possible to start a transaction within a transaction. Such a transaction is called a subtransaction. A subtransaction may be rolled back without affecting the outcome of its parent transaction. If a subtransaction is committed, the sub-commit is not made permanent until the top-level parent transaction commits successfully.

Subtransactions may be nested to any level. A transaction and all of its child transactions compose a transaction family. The OTS intefaces provide methods to determine the relationship, if any, between individual transactions.

By default ORBACUS OTS fully supports nested transactions. However in some environments, use of subtransactions may be incompatible with specific databases or programming interfaces. For these cases, ORBACUS OTS can be configured to forbid the use of subtransactions.

Checked Transactions

The X/Open DTP model places certain restrictions on the use of transactions in order to guarantee transactional integrity. A transaction service that enforces these restrictions is a checked transaction service.

A checked transaction service will prevent a transaction from being committed prematurely while there is still outstanding work being performed in the transaction. This helps to insure data integrity. An unchecked transaction service will allow a premature commit.

By default ORBACUS OTS is a checked transaction service but can be configured to provide an unchecked service.

Synchronization Support

Entities interested in being notified of the beginning and end of a top-level transaction commit and rollback register an object supporting the `CosTransactions::Synchronization` interface. This interface is typically used to manage system resources, such as database handles and data caches, associated with a transaction's `Resources`. Support for this interface is always available with ORBACUS OTS.

Interposition

As an optimization, an OTS implementation can make a local representation of a transaction so that local entities, such as `Resources`, interact with a local transaction representation instead of the original. This can be of considerable benefit in optimizing network usage. Local can mean either in-process, the same-machine, or on the local network. Interposition is transparent to client programs when using the implicit programming model. It can be used with the explicit model but the client code must be modified.

The interposition technique works by having the local representation register itself as a `Resource` with the original transaction. This allows the local representation to take part in the completion protocol of the original transaction.

ORBACUS OTS can be configured to use interposition at the in-process, machine, or network scope. In-process is provided by a simple helper class. The others can be configured on the command line or by an ORBACUS OTS property.

3.2.4 Transaction Policies

As described in the OMG OTS specification [5], the transaction service uses POA policies to define an object's transactional characteristics. The three available `CosTransactions::OTSPolicy` values are:

- `REQUIRES`, The object must be called within a transaction.
- `FORBIDS`, The object can not be called within a transaction.
- `ADAPTS`, The object will behave the right way whether called within or without a transaction.

If an object has no transactional policy value associated with it, the default policy is `FORBIDS`. These policies are checked by the ORBACUS OTS runtime when the implicit programming model is used.

There is one client side policy, the `CosTransactions::NonTxTargetPolicy`, available as a convenience when invoking on objects that have a transactional policy of `FORBIDS`. The values are:

- `PREVENT`, if a target object has a FORBIDS policy and a request is made within the context of a transaction, the transaction will not be allowed to proceed.
- `PERMIT`, if a target object has a FORBIDS policy and a request is made within the context of a transaction, the transaction is allowed to proceed. The transaction will not be propagated to the target object.

Transactional policies are not checked when the explicit programming model is used.

3.3 *Functional Overview of the OTS Interfaces*

The OTS provides several interfaces and structures to manage the use of transactions in applications. All of the OTS IDL is defined in the `CosTransactions` module. These sections provide a functional overview. Please consult the OTS specification [5] for details. A complete listing of the `CosTransactions` module IDL is presented in Appendix A.

3.3.1 Transaction

In the OTS, there is no ‘transaction’ interface. Instead two interfaces, `Coordinator` and `Terminator`, model a transaction. A third interface, `Control`, is provided as an access container for the first two.

The logical transaction interface is partitioned into two interfaces to provide control over what programming entities may do with the transaction. For example, it may not be appropriate to allow every process involved with a transaction the ability to commit the transaction. This restriction can be enforced by splitting the transaction interface such that operations to commit and rollback the transaction are on a separate interface (`Terminator`) and restricting access to this object.

In addition to these interfaces, the OTS specification defines three structures which define the identity of a transaction and describe a transaction hierarchy or branch.

The interfaces and structures are described below.

Coordinator

The `Coordinator` interface provides the bulk of the available transaction operations. It has no operations to commit or rollback a transaction. It can, however, mark a transaction for rollback only. Once marked for rollback, the transaction cannot be committed successfully.

The `Coordinator` interface provides operations to:

- register `Resources` with a transaction
- register `Synchonizations` with a transaction
- create subtransactions
- register subtransaction aware `Resources`
- mark a transaction for rollback only
- query transaction status
- query a transaction’s relationship to another transaction

- query a transaction’s diagnostic name

An OTS implementation can restrict the threads and processes in which a `Coordinator` is valid. ORBACUS OTS places no limitations on the valid scope of the coordinator.

Terminator

The `Terminator` interface provides the two necessary operations to complete a transaction. The `commit` operation initiates the two-phase commit and `rollback` rolls a transaction back.

An OTS implementation can restrict the threads and processes in which a `Terminator` is valid. ORBACUS OTS places no limitations on the valid scope of the `Terminator`.

Control

The `Control` interface can be considered a simple wrapper for the `Coordinator` and `Terminator` interfaces described above. A `Control` reference is returned from any OTS methods that start a new transaction or subtransaction. `Control` provides two accessor methods:

- `get_terminator`
- `get_coordinator`

Either of these operations may raise an exception indicating the desired interface is not available in the scope of the caller.

An OTS implementation can restrict the threads and processes in which a `Control` is valid. ORBACUS OTS places no limitations on the valid scope of a `Control.`

ORBACUS OTS does not restrict access to the `Coordinator` or `Terminator` through `Control`. All participants in the transaction may obtain a reference to the `Coordinator` and `Terminator` objects.

Identity

Transaction identity and context is defined by three structures in the OTS. The first struct is `otid_t`. It represents the X/Open transaction identifier `(XID)`[9]. Both the `otid_t` and `XID` logically consist of two octet sequences. The first sequence uniquely identifies a transaction within a domain. The second defines the transaction branch.

The next struct, `TransIdentity`, consists of an `otid_t`, a `Coordinator` and a `Terminator`. The members of this structure provide full access to the transaction.

The final struct, `PropagationContext,` has a `TransIdentity` member with an additional `TransIdentity` sequence representing the parent transactions. This sequence is empty for a top-level transaction.

The `PropagationContext` represents a single branch of a transaction family. When the implicit programming model is used, the OTS is maintaining an implicit association between a `PropagationContext` and a thread of execution`.`

3.3.2 Transaction Creation

The `TransactionFactory` interface is responsible for creating top-level transactions. The `create` operation is used to instantiate a new top-level transaction. For clients using the explicit programming model, a reference to this interface is available to client applications by calling `resolve_initial_references` with the token "`TransactionFactory`". The implicit OTS runtime uses this reference internally.

In addition to creating a new transaction, the `TransactionFactory` allows an existing PropagationContext to be imported into the OTS. Given a `PropagationContext`, the `recreate` operation will return a `Control` that represents the `PropagationContext` within this particular OTS. The `recreate` operation is useful for importing a transaction into the OTS that originated elsewhere. This is the basis for interposition support.

Subtransactions are created using the `create_subtransaction` method of the `Coordinator` interface. Since a subtransaction is dependent on the context of its parent, this operation must be on a transaction specific interface, not on the `TransactionFactory`.

3.3.3 Transaction Completion

The `CosTransactions` module defines the `Resource` and `SubtransactionAwareResource` interfaces for registering objects that will take part in transaction completion. A `Terminator` object is used to initiate the commit or rollback of a transaction. The commit or rollback process will involve the `Resource` objects.

Resource

The `Resource` interface defines the operations required to carry out the two-phase commit. `Resource` objects are registered with a transaction by calling `Coordinator::register_resource`.

As part of the two-phase commit process, the OTS will first call `prepare` on each of the registered `Resources` in order to get a vote. A `Resource` may vote `VoteCommit`, `VoteRollback`, or `VoteReadOnly`.

If the `Resources` all vote to commit, the OTS finalizes the decision by calling the `commit` method on each `Resource`.

If any `Resource` votes to roll back, the OTS must then abort the transaction by calling `Resource::rollback` on any `Resources` that have not yet voted or have previously voted `VoteCommit`.

Heuristic Decisions

Even with voting, it is not guaranteed that a `Resource` will be able to comply with the intent of the two-phase commit. It may be for instance, that by the time a `Resource` receives the `commit`, it has timed out and has already rolled back. Such an independent decision is a heuristic decision. A `Resource` will raise an exception if it made a heuristic decision in conflict with the requested transaction completion.

Subtransaction Resource

Subtransactions do not use the two-phase commit protocol like top-level transactions. There is no voting, only notification of commit or rollback. The `SubTransactionAwareResource` interface, which derives from `Resource`, is used to receive notification.

A `SubtransactionAwareResource` object must call the transaction `Coordinator’s` `register_subtran_aware` to receive notification. Then when a subtransaction commits or rolls back, the OTS will call the `SubtransactionAwareResource` object’s `commit_subtransaction` or `rollback_subtransaction` method.

Synchronization

The Synchronization interface provides callback methods to allow for system resource management associated with top-level transactions. The interface provides two methods:

- `before_completion`, called before the initiation of the two-phase commit protocol
- `after_completion`, called after commit or rollback is complete

The `before_completion` method would be the place where a transactional object might flush a local cache to its backing `Resource` object. The `after_completion` method could be used release a database handle or lock associated with a transaction.

3.3.4 Implicit Programming Support

So far all the interfaces presented allow for direct manipulation of a transaction using the explicit programming model. Programs using the implicit model, primarily interact with a transaction through the `Current` interface. The `Current` interface allows basic operations on the transaction implicitly associated with the current thread. It also allows for changing the association between threads and transactions. The interface provides operations to:

- `begin` a transaction
- `commit` a transaction
- `rollback` a transaction
- `suspend` the association of a thread with a transaction
- `resume` the association of a thread with a transaction
- get the `Control` for the implicit transaction.

The Current object is available to client applications through `resolve_initial_references("TransactionCurrent")`. It is only available if an ORB instance has been initialized for use with the implicit programming model.

3.4 Programming Examples

The following sections give a small glimpse of what coding with an OTS is like. These examples are incomplete fragments and have omitted absolutely essential things like error checking and exception handling. They are provided to give some initial context for the concepts introduced in this chapter. To fully understand them it is essential to read the chapters that follow as well as the OTS specification [5].

3.4.1 Implicit Client

The following code fragment illustrates a funds transfer transaction using the implicit programming model.

```
// C++
CosTransactions::Current_var current =
   CosTransactions::Current::_narrow(CORBA::Object_var(
      orb -> resolve_initial_references("TransactionCurrent")));

current -> begin();

newYorkAccount -> withdraw(50.0);
```

```
 9 parisAccount -> deposit(50.0);
10
11 current -> commit();
12
```

2 The Current object is available through `resolve_initial_references`.

6 The transaction is started.

8,9 With the implicit model, the transaction does not have to be an argument to any methods.

11 The transaction is committed.

The `Current` object defines the bounds of the transaction. The ORBACUS OTS implicit runtime insures that the transaction context is associated with all method invocations.

3.4.2 Implicit Server

The following code fragment illustrates a server side implementation of a bank account operation involved for the funds transfer transaction using the implicit programming model. It is somewhat more complex than the client side:

```
 1 // C++
 2 Account_impl::withdraw(float Amount)
 3 {
 4    CosTransactions::Current_var current = ...;
 5    CosTransactions::Control_var control = current -> get_control();
 6    CosTransactions::Coordinator_var coord =
 7                                control -> get_coordinator();
 8
 9    acctRes_ = new AccountResource_impl(this, ...);
10    CosTransactions::Resource_var res = acctRes_ -> this_();
11    CosTransactions::RecoveryCoordinator recovery =
12                          coord -> register_resource(res);
13
14    acctRes_ -> withdraw(Amount);
15 }
16
```

5-7 Obtain the transaction `Coordinator`.

9-10 The `Account_impl` is represented in the transaction by an `AccountResource` object. It is initialized with the Account_impl `this` pointer and has access to the Account_impl’s balance. In this example, the `AccountResource` interface derives from `Resource`.

11 The `AccountResource` is registered with the transaction. The `RecoveryCoordinator` returned from `register_resource` is used in transaction recovery.

14 Note that the `withdraw` is carried out by the `AccountResource`. The `AccountResource` maintains a ‘working’ account balance during the transaction. This is how the resource is able to commit or roll back the changes made during the transaction

The transactional object implementation is responsible for registering with a transaction Coordinator. In addition to overlooking exception handling, this example does not perform other essential tasks like checking whether the account is already involved in a transaction.

3.4.3 Explicit Client

The following code fragment illustrates the same funds transfer transaction using the explicit programming model.

```
 1 // C++
 2 CosTransactions::TransactionFactory_var txnFactory =
 3     CosTransactions::TransactionFactory::_narrow(CORBA::Object_var(
 4         orb -> resolve_initial_references("TransactionFactory")));
 5
 6 CosTransactions::Control_var ctrl = txnFactory -> create(60);
 7 CosTransactions::Coordinator_var coord = ctrl -> get_coordinator();
 8
 9 newYorkAccount -> txn_withdraw(coord, 50.0);
10 parisAccount -> txn_deposit(coord, 50.0);
11
12 CosTransactions::Terminator_var term = ctrl -> get_terminator();
13 term -> commit();
14
```

2 The `TransactionFactory` object is available through `resolve_initial_references` in ORBACUS OTS.

6 This time the client starts the transaction directly using the `TransactionFactory`. This transaction has a timeout of sixty seconds. Passing a zero to `create` will start a transaction with no timeout. This is not recommended.

7-10 The `Coordinator` is explicitly passed as an argument to the `txn_withdraw`, and `txn_deposit` operations.

12,13 To finish the transaction, the client must obtain a reference to the transaction’s `Terminator`.

3.4.4 Explicit Server

The following code fragment revisits the server side implementation of the withdraw operation using the explicit programming model.

```
 1 // C++
 2 Account_impl::txn_withdraw(CosTransactions::Coordinator coord,
 3                            float Amount)
 4 {
 5    acctRes_ = new AccountResource_impl(this, ...);
 6    CosTransactions::Resource_var res = acctRes_ -> this_();
 7    CosTransactions::RecoveryCoordinator_var recovery =
 8                            coord -> register_resource(res);
 9
10    acctRes_ -> withdraw(Amount);
11 }
12
```

5 The `AccountResource` object is still used to represent the `Account` when using the explicit programming model. `Resources` are used in transactions, but are not themselves transactional objects, so they do not rely on the implicit programming model.

7 The `AccountResource` is registered with the transaction. The `RecoveryCoordinator` returned from `register_resource` is used for transaction recovery.

10 The `withdraw` is carried out by the `AccountResource`. As with the implicit example, this only affects the transaction’s ‘working’ balance.

CHAPTER 4

Working with ORBacus OTS

4.1 Initializing ORBacus OTS

4.1.1 Configuring the TransactionFactory

The ORBACUS OTS server[1] supports the `CosTransactions::TransactionFactory` interface. The `TransactionFactory` is usually made available through `CORBA::ORB::resolve_initial_references` by specifying the initial reference on the command line:

```
-ORBInitRef TransactionFactory=corbaloc::<otshost>:<otsport>/
TransactionFactory
```

or in an ORBACUS configuration file:

```
ooc.orb.service.TransactionFactory=corbaloc::<otshost>:<otsport
>/TransactionFactory
```

where `<otshost>` and `<otsport>` are set to the appropriate host name or IP address and port.

The `TransactionFactory` must be configured as an initial reference to use the implicit programming model with ORBACUS OTS. If a program only uses the explicit model, the

1. See Section 2.1 on page 9 for details on configuring and starting an OTS server process.

TransactionFactory reference can be obtained by using resolve_initial_references or in any other manner appropriate, such as through a naming service.

4.1.2 Implicit Programming Support

To support the implicit programming model, the ORBACUS OTS runtime must implicitly propagate transaction contexts with every CORBA request. Implicit support is enabled by calling the ORBACUS OTS OTSInit method before instantiating any ORBs[1] that require implicit transaction support. In C++:

```
// C++
...
CORBA::ORB_var noTxnOrb = CORBA::ORB_init(argc, argv);
OB::OTSInit(argc, argv);
CORBA::ORB_var txnOrb = CORBA::ORB_init(argc, argv);
...
```

and in Java:

```
// Java
...
org.omg.CORBA.ORB noTxnOrb = org.omg.CORBA.ORB.init(args, props);
com.ooc.CosTransactions.OTSInit.OTSInit(props);
org.omg.CORBA.ORB txnOrb = org.omg.CORBA.ORB.init(args, props);
...
```

In the examples above, only the second orb, txnOrb, will support implicit programming and the use of CosTransactions::Current. The first orb, noTxnOrb, will not support the implicit programming model.

TransactionCurrent

After OTSInit is called, a CosTransactions::TransactionCurrent object is available by calling CORBA::ORB::resolve_initial_references with the token "TransactionCurrent".

1. Any additional orbs instantiated will also be associated with the ORBACUS OTS server. Using the implicit programming model with multiple orbs does require extra work on the part of the application program. See “Multiple ORBs” on page 46 for details.

Interposition

Interposition can be used to decrease network traffic by OTS operations by recreating a local representation of a transaction that originated on a remote network. Interposition should not be used without giving adequate consideration to the particular processes and network topology involved.

ORBACUS OTS interposition can be configured in two ways. The first method, in-process, allows an OTS server to be collocated with an application client or server. This method is only available when using C++. In-process interposition is enabled by creating an interposed OTS instance after instantiating an orb:

```
// C++
...
OB::OTSInit(argc, argv);
CORBA::ORB_var txnOrb = CORBA::ORB_init(argc, argv);
OB::OTSInterposed(txnOrb, argc, argv);
...
```

This technique should be used with caution as generally speaking a process containing an OTS server must have high availability in order to guarantee transaction completion in the face of system failure.

The second way to configure interposition is by setting up an out of process interposition `TransactionFactory`. This option is available for both C++ and Java by setting an ORBACUS configuration property:

```
ooc.ots.implicit.interposition=<txnfactory-url>
```

where `<txnfactory-url>` is set to an appropriate `TransactionFactory` URL. This URL can be the same one that was used for `ooc.orb.service.TransactionFactory`.

If ORBACUS OTS is not explicitly configured to use interposition all transaction communication will be with the original transaction's `Control`, `Coordinator` and `Terminator`.

4.2 *Application Programming Models*

4.2.1 Implicit

The implicit programming model relies on the OTS runtime to maintain an implicit association between the current thread of execution and a transaction. Programs interact with the OTS service through the `CosTransactions::Current` interface. See Interface Cos-Transactions::Current for a detailed listing of this interface.

Transaction Policy Enforcement

All CORBA objects have one of three OTS transaction policy values: `FORBIDS, ADAPTS`, or `REQUIRES`. The implicit OTS runtime is responsible for enforcing these polices. Enforcement is performed at both the client and target ends of a request so that a non-compliant client cannot corrupt a server and vice-versa.

FORBIDS

An object with a `FORBIDS` policy cannot be invoked within a transaction. If an object has no explicit OTS policy then `FORBIDS` is the default.

The OTS implicit runtime will raise a `CORBA::INVALID_TRANSACTION` if a request is made to such an object from within a transaction.

This behavior can be relaxed to accommodate situations where use of pre-existing CORBA objects with no OTS policy is problematic. The orb policy `CosTransctions::NonTxTargetPolicy` can be set to either `PREVENT` or `PERMIT`. If set to `PREVENT`, `FORBIDS` operates as described above. If the policy is set to `PERMIT`, then the transaction is suspended[1] for the duration of the call to the `FORBIDS` object. `PERMIT` allows *non-transactional* objects to participate in a transaction.

ADAPTS

An object with an `ADAPTS` policy can be called on with or without a transaction. The OTS runtime sends the transaction context with the request if it exists.

REQUIRES

An object with the `REQUIRES` policy cannot be called outside of a transaction. If a call is made without a transaction, the OTS runtime will raise the `CORBA::TRANSACTION_REQUIRED` exception.

Object Transactional Policy

The implicit programming model applies the OTS Policy in a target object reference to all of the object's methods. This includes `CORBA::Object` methods such as `is_a` and

1. The OTS is not literally required to suspend and resume. It just must behave as if it does.

non_existent as object life cycle and supported interfaces can be transactional operations.[1] There is no individual control of a method's transactional policy.

Exceptions

One key feature of the implicit programming model is that any CORBA::SystemException raised when a request is made within a transaction will cause the transaction to be marked for rollback only.

User exceptions do not directly affect the outcome of a transaction. A client is free to take whatever action is appropriate for a user exception.

Client Use

Most operations that an application program will need such as begin and commit can be performed directly using the Current interface itself.

Server Use

A transactional server will typically require access to a transaction Coordinator. An example of obtaining a reference to the Coordinator is shown in this example for C++:

```
// C++
CosTransactions::Current_var current = .....
try
{
   CosTransactions::Control_var ctrl = current -> get_control();

   if(CORBA::is_nil(ctrl))
       throw CORBA::TRANSACTION_REQUIRED;

   CosTransactions::Coordinator_var coord =
                                  ctrl -> get_coordinator();
   ....
   //
   // Register a Resource .....
   //
}
catch(const CosTransactions::Unavailable&)
{
```

1. This is not to say that these operations are required to exhibit transactional behavior, just that the OTS policy will be enforced.

```
19    //
20    // Coordinator is unavailable to this thread.
21    //
22 }
```

7 For this example, a transaction is required. If this object has a `REQUIRES` policy, the OTS runtime would have raised the `TRANSACTION_REQUIRED` exception already, so strictly speaking this check is redundant.

17 The `get_coordinator` and `get_terminator` operations can both raise the `Unavailable` exception. An OTS implementation can restrict access to either the `Coordinator` or `Terminator` in any thread or process. ORBACUS OTS does not restrict access to these references and will not raise this exception.

Multiple Threads

A transactional application is not restricted to using a transaction in a single thread. To allow multiple threads to participate in a transaction, a reference to the transaction `Control` must be passed to any threads that will join the transaction. The threads then set their implicit transaction context by calling `CosTransactions::Current::resume`.

The following example shows the basis of a worker thread class that can participate in an implicit transaction:

```
 1 // C++
 2 #include <JTC/JTC.h>
 3
 4 class TxnWorkerThread : public JTCThread
 5 {
 6    CosTransactions::Current_var current_;
 7    CosTransactions::Control_var control_;
 8 public:
 9    TxnWorkerThread(CosTransactions::Current current,
10                    CosTransactions::Control_ptr ctrl) :
11    current_(CosTransactions::Current::_duplicate(current),
12    ctrl_(CosTransactions::Control::_duplicate(ctrl))
13    {
14    }
15    virtual void run()
16    {
17       current -> resume(ctrl);
18       ...
19       //
20       // Perform transactional work in this thread
21          //
22          ...
23          current -> resume(CosTransactions::Control::_nil());
24       }
25    }
```

4 This class makes use of the OOC's JThreads/C++ package [7] for threading support.

9 The constructor is given a reference to the `CosTransactions::Current` and the `Control` for the transaction of interest.

15 The `run` method calls `Current::resume` to implicitly associate the transaction with the thread.

23 Before exit a user created thread must release any implicit transaction it is associated with. Failure to do so will prevent the OTS runtime from reclaiming internal data associated with the transaction.

A sample use of the `TxnWorkerThread` class to create a transaction that uses two additional threads is:

```
// C++

CosTransactions::Current current =
   orb -> resolve_initial_references("TransactionCurrent");

try
{
   current -> begin();

   TxnWorkerThread t1(current, ctrl);
   TxnWorkerThread t2(current, ctrl);

   t1.start();
   t2.start();

   someObj -> operation1();
   someOtherObj -> operation2();

   do
   {
      try
      {
         t1.join();
24          t2.join();
25       catch(const JTCInterruptedException&)
26       {
27       }
28    }
29    while(t1.isAlive() || t2.isAlive());
30
31    current -> commit();
32 }
33 catch(const CORBA::TRANSACTION_ROLLEDBACK)
34 {
35    ....
36 }
```

10-11 Create the worker threads.

13-14 start creates the new threads and executes their run methods. The new threads should now resume the transaction and perform their work.

19-29 Wait for the threads to exit their run method. See the JThreads/C++ Manual [7] for details.

31 All the threads have completed their work, commit the transaction.

33 Any one of the threads can cause the transaction to rollback.

Multiple ORBs

There are situations where within a single process it may be required to involve object references from different `ORB` instances in the same transaction. The orb instances may have been created with different polices or initial references. From an implementation standpoint, this situation is very similar to involving multiple threads in a single transaction. It is up to the application to insure that the implicit context for all of the `ORB` instances is the same. The code fragment below illustrates the basics of a multi-orb helper class called `TwoOrbCurrent` and its `begin` method implementation.

```
// Java
public class TwoOrbCurrent
{
   private org.omg.CORBA.ORB orbs_[2];
   private org.omg.CosTransactions.Current current_[2];
...

public TwoOrbCurrent(org.omg.CORBA.ORB o1, org.omg.CORBA.ORB o2)
{
...
```

```
11 public begin()
12 {
13    orbs_[0].begin(0);
14    org.omg.CosTransactions.Control ctrl =
15                        current_[0].get_control();
16    current_[1].resume(ctrl);
17
18 public commit()
19 {
20    current_[0].commit(true);
21    current_[1].resume(null);
22 }
```

2 The `TwoOrbCurrent` class operations mirror those of `CosTransactions::Current` but always work with two orbs.

4-5 The class keeps track of the orbs involved and the `CosTransactions::Current` object[1] for each orb.

11 The `begin` method starts a transaction using the first orb's `Current` and then uses `get_control` and `resume` to involve the other `ORB` in the same transaction. (All exception handling has been ignored.)

18 The `commit` method commits the transaction using the first orb's `Current` and releases the second orb's association by using a `resume(nil)`. (All exception handling has been ignored.)

Using the `TwoOrbCurrent` class, a multiple orb client example is:

```
// Java
org.omg.CORBA.ORB orb1 = org.omg.CORBA.ORB_init(args1, props1);
org.omg.CORBA.ORB orb2 = org.omg.CORBA.ORB_init(args2, props2);

TwoOrbCurrent multiCurrent(orb1,orb2,....);
multiCurrent.begin();

obj1 = SomeObjectType.narrow(orb1.resolve_initial_references(...));
obj2 = SomeOtherType.narrow(orb2.resolve_initial_references(...));

obj1.operationX();
obj2.operationY();
```

1. This is a different object for each `ORB` instance.

```
13
14 multiCurrent.commit();
```

2-3 Two orbs are initialized with some different properties or policies, such as timeouts.

5 The `TwoOrbCurrent` helper is created with orb1 and orb2.

6 The `TwoOrbCurrent::begin` call implicitly associates both orbs with the transaction in this thread.

11-12 Transactional operations are performed on object references on both orbs.

14 The `TwoOrbCurrent::commit` method commits leaving both orbs with no implicit transaction association in this thread.

Checked Transactions

With the implicit propagation model, ORBACUS OTS mediates every request made on a CORBA object. This allows the ORBACUS OTS runtime to keep track of the requests made on each transaction.

If ORBacus OTS is running in checked mode it will not allow a transaction to be committed while there are still outstanding requests that have been made in the context of this transaction. Checked mode is the default with ORBACUS OTS.

The possibility of committing a transaction with outstanding requests is only possible in two scenarios. The first is if multiple threads are involved in the same transaction where one thread tries to commit while another is awaiting the reply from a `CORBA` invocation. The second is if deferred synchronous requests are made and an attempt to commit is made before the results of the requests are received.

If only normal synchronous invocations are made and a transaction is not explicitly resumed in multiple threads, the transaction will never violate the checked constraint.

4.2.2 Explicit

When using the explicit programming model, the application is responsible for using the `CosTransactions` module objects directly to use transactions. The OTS provides no runtime support for explicit programming.

Checked Transactions

There is no concept of a checked transaction when the explicit programming model is used.

Exceptions

System exceptions do not affect the outcome of a transaction using the explicit programming model. Transactions can only be rolled back by calling `Terminator::rollback` or `Coordinator::rollback_only`.

Method Transactional Policy

`CosTransaction` module objects such as `Control` and `Coordinator` must be passed as explicit parameters to interface methods with the explicit model. This allows the programmer to specify whether individual methods of an interface are transactional or not.

The OTS Policy of an object reference is not used by the explicit model.[1]

Interposition

Interposition can be used with the explicit programming model. The `TransactionFactory` interface has a `recreate` method on it that can be used to import a transaction into a specific OTS server instance. The application program must then use the returned `Control` reference to represent the transaction. This `Control` reference is an interposed representation of the original transaction. This process is similar to what the OTS implicit runtime does to implement interposition.

4.2.3 Mixed

It is possible, but not encouraged, to mix the explicit and implicit programming models. A transactional programming model should be decided for a project and then used uniformly throughout.

There are cases, such as application bridging, where mixed model programming may be unavoidable.

Implicit to Explicit

It is trivial to obtain the particular `CosTransactions` object reference required for an explicit call by starting with `CosTransactions::Current::get_control`.

1. OTS Policy checking is dependent on whether the orb was initialized for implicit propagation.

Implicit Tranaction Policy Enforcement

A problem can arise when trying to make the call using the explicit model if the OTS policy for the target object is `FORBIDS`. This can easily be the case if the object was designed for use with explicit propagation only. The trouble is that the implicit orb policy check will still be performed. This will cause the call to fail and for the transaction to be marked for rollback.

This can be avoided by setting the `NonTxTargetPolicy` on the orb to PERMIT or by creating the explicit object references on an orb that has not been registered for implicit transaction propagation, that is an orb that is instantiated before the call to `OTSInit`.

Explicit to Implicit

To go from explicit to impolite, a reference to the transaction `Control` is required. The transaction can be made implicit by calling `CosTransactions::Current::resume`. `OTSInit` must have been called to initialize the orb for implicit support.

An OTS implementation can restrict which `Control` references are valid for `resume`. ORBACUS OTS does not restrict the use of `resume`.

Checked Transactions

Once a transaction is used with the explicit programming model, it is not possible for the implicit OTS runtime to correctly maintain the outstanding request count for the explicit portions of the transaction. If mixed model programming is used, the application cannot rely on the OTS to enforce checked behavior.

4.3 Subtransactions

Subtransactions allow an application to perform portions of a transaction in an isolated manner, allowing them to be rolled back without forcing the parent transaction to rollback as well.

Any method that raises a `CORBA::SystemException` will cause a transaction to be marked for rollback. Subtransactions can be used to isolate any transactional work that can raise a `SystemException` but should not cause the rollback of the parent transaction. In this case only the subtransaction will be marked for rollback.

It is important to remember that subtransactions do not operate the same way as top-level transactions. They do not use a two-phase commit protocol and `Resources` registered with them will not receive a `prepare` or `commit` until the enclosing top-level transaction completes. The interface SubTranAwareResource is used to receive subtransaction commit and rollback notification (See “SubTransactions” on page 58.)

By default ORBACUS OTS enables the use of subtransactions. Since many `Resources` such as XA databases do not support subtransactions, they can be disabled by setting the configuration property `ooc.ots.implicit.subtransactions` to `false`.

4.3.1 Implicit Use

Both top-level and subtransactions are initiated by calling `CosTransactions::Current::begin`. A subtransaction can be started by calling `begin` inside of an active transaction as shown:

```
// C++
CosTransactions::Current_var current =
   orb -> resolve_initial_references("TransactionCurrent");

current -> begin();

obj1 -> operation1();

try
{
   current -> begin();
}
catch(const CosTransactions::SubTransactionsNotAvailable&)
{
   ...
}
obj2 -> operation2();

try{
   current -> commit(false);
}
catch(const CORBA::TRANSACTION_ROLLEDBACK&)
{
   ...
}
catch(const CORBA::SystemException&)
{
   ...
}

try
```

```
32 {
33    current -> commit(false);
34 }
35 catch(const CORBA::TRANSACTION_ROLLEDBACK&)
36 {
37    ...
38 }
39
```

5 A top-level transaction is started.

7 An invocation is performed in the context of this top-level transaction.

11 A subtransaction is started using `Current`::`begin`.

13 `CosTransactions::SubTransactionsNotAvailable` is raised if ORBACUS OTS is not configured to allow subtransactions.

17 An invocation is performed in the context of the subtransaction.

20 Commit the subtransaction.

22 Rollback of the subtransaction does not affect the parent transaction.

26 `CORBA::SystemExceptions` do not affect the parent transaction.

33 The top-level transaction is committed.

35 Any exception here is not directly related to the subtransaction raising an exception.

4.3.2 Explicit Use

The explicit use of a subtransaction is conceptually similar to the implicit case except that `CosTransactions::Coordinator::create_subtransaction` method is used to initiate the subtransaction. An example is shown below:

```
 1 // C++
 2 CosTransactions::TransactionFactory_var txnFactory =
 3    orb -> resolve_initial_references("TransactionFactory");
 4
 5 CosTransactions::Control_var top_level_txn;
 6 CosTransactions::Control_var sub_txn;
 7
 8 top_level_txn = txnFactory -> create(0);
 9 obj1 -> operation1();
10
11 try
```

```
12 {
13    CosTransactions::Coordinator_var coord =
14       top_level_txn -> get_coordinator();
15    sub_txn = coord -> create_subtransaction();
16 }
17 catch(const CosTransactions::SubTransactionsNotAvailable&)
18 {
19       // OTS not configured to allow subtransactions;
20       ...
21 }
22 catch(const CosTransactions::Inactive&)
23 {
24       // This transaction is already being prepared
25       ...
26 }
27 obj2 -> operation2();
28
29 try{
30    CosTransactions::Terminator_var term =
31       sub_txn -> get_terminator();
32    terminator -> commit(false);
33 }
34 catch(const CORBA::TRANSACTION_ROLLEDBACK)
35 {
36    //
37    // Top-level transaction can still be committed
38    //
39    ....
40 }
41 try
42 {
43    CosTransactions::Terminator_var term =
44       top_level_txn -> get_terminator();
45    term -> commit(false);
46 }
47 catch(const CORBA::TRANSACTION_ROLLEDBACK&){
48    //
49    // Top-level transaction failed to commit
50    //
51    ...
52 }
53
```

8 A top-level transaction is started using the `TransactionFactory`.

15 A subtransaction is started using `Coordinator::create_subtransaction`.

32 Commit the subtransaction using the subtransaction's `Terminator`.

43 The top-level transaction is committed with the top-level `Terminator`.

4.4 *Implementing Servers and Resources*

4.4.1 POAs and Transactional Policies

An object reference contains the transactional policy of the POA the servant was created on. For example if you wish an object to have a `REQUIRES` OTS policy, the servant must be activated on a POA that has a `REQUIRES` policy. All objects created on the same POA have the same transactional policy.

Creating a POA with a Transactional Policy

```
 1 // C++
 2
 3     PortableServer::POAManager_var manager =
 4                  rootPOA -> the_POAManager();
 5
 6     //
 7     // Create a POA with a transactional policy
 8     //
 9     CORBA::PolicyList policies(1);
10     policies.length(1);
11
12     CORBA::Any txnPolicyAny;
13     txnPolicyAny <<= CosTransactions::REQUIRES;
14     policies[1] = orb ->
15              create_policy(CosTransactions::OTS_POLICY_TYPE,
16                     txnPolicyAny);
17
18     PortableServer::POA_var txnPOA =
19         rootPOA -> create_POA("RequiresTxnPOA", manager, policies);
20
```

3 The new POA will be created as a child of the `RootPOA` and will use the `RootPOA` manager.

9-10 A `CORBA::PolicyList` is created and sized to contain the single OTS policy.

12-13 An `Any` holds the OTS policy value.

14 The orb creates the requested policy. In order for this to work the OTS policy factory must have been previously registered with the orb. The OTS policy factory is registered if `OTSInit` is called before this orb was initialized.

18 The POA is created. Any object references created using this POA will contain the OTS `REQUIRES` policy.

It is only necessary to use a POA with a transactional policy if you intend for this object to be invoked from an orb that has been registered for implicit propagation. This is so the implicit runtime will do the appropriate policy checks and pass the transaction context along with the invocation.

4.4.2 Transactional Objects

Resource Use

Application programs send requests to transactional objects during the life of a transaction. A transactional object implementation uses a `CosTransactions::Resource` object to represent itself to the transaction `Coordinator`. Client applications are unaware of the `Resource` object and should never be allowed direct access to it. A `Resource` is registered by calling `CosTransactions::Coordinator::register_resource`[1].

For the purposes of illustrating the concepts involved, the next sections assume a transactional object design in which the transactional object delegates state operations to a separate object which implements the `CosTransactions::Resource`[2] interface.

Serial Transaction Objects

The most straight forward case is where an object can only take part in one transaction at a time. In this case the transactional object implementation keeps track of, and delegates to a single transaction-specific helper object.

Concurrent Transaction Objects

For a transactional object to participate in more than one transaction at a time then the object implementation must maintain its working state for each active transaction.

1. There are different rules for subtransactions. See “SubTransactions” on page 56.
2. This is only one of many implementation choices.

SubTransactions

The behavior of a transactional object in regard to subtransactions depends on the implementation's use of `Resource` or a `SubTransactionAwareResource` objects. If a subtransaction commits, the state changes are not final until the top-level parent commits. If the top-level parent rolls back, the subtransaction's changes are undone.

4.4.3 Resources

A `Resource` is responsible for representing a transactional object during the lifetime of a transaction. A `Resource` is, by definition, associated with a single top-level transaction. Writing a `Resource` that performs correctly under all conditions, particularly system failure during the two-phase commit, is a non-trivial task.

XA Resources

Many transactional objects use a third party database to record their object state. If the database being used supports the XA interface [6] then ORBACUS OTS provides support that makes the use of the XA features of the database transparent to the application. See Chapter 5 for details.

Transactional Policy

A `Resource` should not be created with a transactional policy of `REQUIRES`. The reason being that the ORBACUS OTS server always uses the explicit programming model when it invokes a method on a `Resource` during commit or rollback.

If the `Resource` was created on an ORB instance that has implicit programming support, the OTS runtime will raise `CORBA::TRANSACTION_REQUIRED` for any request from the OTS server. This will prevent any transaction involving the `Resource` from completing.

ACID Responsibilities

A `Resource` is responsible for enforcing its portion of a transaction's ACID properties. A `Resource` must not let any of its state changes be visible outside of the transaction until it is told to `commit`. There must be no trace of any state change left if it is told to `rollback`.

Heuristic Decisions

Ideally a `Resource` will end its association with a transaction when it receives a `commit` or `rollback`. Unfortunately in the real world a `Resource` may not receive a `commit` or `rollback` in a timely manner. If it is holding locks or some other critical resource it may not be able to wait forever for transaction completion. In this case, a Resource can make a heuristic decision to commit or rollback. Generally if a `Resource` voted for rollback of the transaction, the heuristic decision will be to rollback. If the `Resource` voted to commit, it is an implementation choice as to whether a heuristic commit or rollback is more appropriate. After making a heuristic decision, the `Resource` must stay associated with the transaction.

The OTS server will be informed of the `Resource's` decision when, and if, it ultimately does invoke `commit` or `rollback` on the Resource. The Resource will raise an exception if its heuristic decision is in conflict with the requested `commit` or `rollback`. When a heuristic exception is raised by a `Resource`, administrative intervention is required to insure transactional integrity.

If a `Resource` makes a heuristic decision. It is obligated to persistently remember that decision and to remain in existence until its `forget` method is invoked by the transaction service. The `forget` method indicates that the `Resource's` heuristic decision has been accounted for.

The Resource and SubTranAware Resource Interfaces

The CosTransactions module defines two interfaces for transaction resources: `Resource` and `SubTranAwareResource`. The `SubTranAwareResource` interface is derived from Resource but has two additional methods for use with subtransactions, `commit_subtransaction` and `rollback_subtransaction`. In the rest of this chapter, when a Resource is referred to it can be an object of type `Resource` or `SubTranAwareResource`.

Registering with a Transaction

Top Level Transactions

When a `Resource` object is registered with a top-level transaction coordinator using `CosTransactions::Coordinator::register_resource`, the object will take part in the transaction completion protocol.

The `register_resource` method returns a reference to a `CosTransactions::RecoveryCoordinator`. This reference should be kept by the `Resource` for use in recovery. It does not have to be placed in stable storage until the `Resource` receives the prepare.

SubTransactions

When a `Resource` or `SubTransactionAwareResource` object is registered with a subtransaction coordinator using `CosTransactions::Coordinator::register_resource`[1], the object will take part in the transaction completion of the enclosing top-level transaction, not the subtransaction. There is no transaction completion protocol for subtransactions.

If a `SubTransactionAwareResource` object wishes to be notified at the end of the subtransaction, it must be registered with the subtransaction coordinator using `CosTransactions::Coordinator::register_subtran_aware`. Then either SubTranAwareResource::`commit_subtransaction` or `rollback_subtransaction` will be called when the subtransaction ends.

On receipt of `commit_subtransaction`, a `SubTransactionAwareResource` must not make any state changes that are visible outside the top-level transaction. The state changes can be made visible, in an application specific manner, to other members of the transaction family[2]. If the enclosing top-level transaction eventually rolls back, the subtransaction's work must be rolled back as well.

If a `SubTranAwareResource` is registered only using `register_subtran_aware` and not `register_resource`, the only method that will be called on it by the OTS server is either `commit_subtransaction` and `rollback_subtransaction`. It will not receive a `commit` or `rollback` when the top-level transaction completes. In this case, it is the application's responsibility to eventually destroy the `SubTranAwareResource` object after either of these methods is called.

The Two Phase Commit

Resources participate in the two-phase commit by voting whether to commit or abort the transaction during the prepare phase and then by finalizing the state of their associated transactional object in response to the second-phase `commit` or `rollback`. If any single `Resource` votes to abort, the transaction is rolled back.

Prepare

The Resource can respond to a `prepare` request with the following:

1. There is some ambiguity in the OTS specification [4] about whether Resource and SubTranAwareResources behave identically when registered with a subtransaction. This document describes the behavior of ORBACUS OTS as implemented in this beta release.
2. The transaction family is defined to be the top-level transaction and all of its subtransactions.

- `VoteRollback`, The `Resource` will not commit state changes associated with this transaction. This is the correct vote for a `Resource` that has forgotten about a transaction[1]. The Resource may forget the transaction or delete itself at this point.
- `VoteCommit`, The Resource will commit the state changes associated with this transaction when `commit` is called.
- `VoteReadOnly`, The transaction can be committed but there was no state change made for this Resource.

Once `prepare` is called, a `Resource` should mark itself as prepared and disallow any further operations that would affect state. Its working state should now be frozen until the transaction completes.

If a `Resource` votes `VoteRollback` or `VoteReadOnly` it can now forget about the transaction. The OTS Server will not contact this `Resource` again. The `Resource` implementation is now responsible for its own life cycle[2].

A `Resource` that votes `VoteCommit`, or has made a heuristic decision, is obligated to remain in existence until the transaction completes. It must be able to survive process termination[3] and restart. Before returning `VoteCommit` it must persistently save any state change it intends to make and the server the `Resource` resides in must be capable of restoring the `Resource` and its state after restart. This implies that the server process is maintaining a persistent list of outstanding `Resources` and their intended commit state or heuristic decisions.

Commit

A `Resource` should only receive a `commit` request if it has been previously prepared. If it has not been prepared, an error has been made in the two-phase commit protocol. In this case it should raise the `CosTransactions::NotPrepared` exception. The `Resource` must remember the transaction after raising this exception.

If a `Resource` has forgotten about the transaction or made a heuristic commit decision then it should do nothing. If it has not made a heuristic decision it should make its state

1. This can happen if the `Resource's` process is shutdown and restarted. When the OTS server then starts the two-phase commit the `Resource` may not remember the transaction. It is also possible the OTS server will receive an `CORBA::OBJECT_NOT_EXIST` when trying to invoke `prepare`. Both of these scenarios depend on how the `Resource` is implemented.
2. In some implementations this would be an appropriate point to delete the `Resource` or return it to a pool.
3. graceful or not.

changes permanent. For the preceding cases, it can now forget the transaction. The OTS server will not contact this `Resource` again.

If the Resource has made a heuristic decision to rollback it must now raise the `CosTransactions::HeuristicRollback` exception. An example of this would be a `Resource` that had a five minute timeout for transaction completion after which it had to independently roll back its state changes in order to release some critical locks. After raising a heuristic exception, the `Resource` must remain associated with the transaction.

Rollback

A Resource's rollback method is called if:

- A call to `Current::rollback` or `Terminator::rollback` on a top-level transaction is made.
- A call to `Current::commit` or `Terminator::commit` on a top-level transaction is made and it has been marked for rollback.
- One `Resource` votes `VoteRollBack` in the prepare phase of the two-phase commit.

During the two-phase commit only `Resources` that have not yet been prepared or voted `VoteCommit` should receive the rollback. Resources that voted `VoteRollback` or `VoteReadOnly` should already have forgotten about the transaction.

If a `Resource` has forgotten about the transaction or made a heuristic decision to rollback it should do nothing.

If a `Resource` has already been prepared, it may raise a heuristic exception if it is unable to `rollback`. If it does so, the `Resource` cannot forget about the transaction until `forget` is called.

4.4.4 Synchronizations

The Synchronization interface provides for taking action before the two-phase commit starts the prepare phase and at transaction completion after all `Resources` have been asked to commit or rollback. This is very useful for orchestrating activities like transaction lock management and cache flushing.

A `Synchronization` is, by definition, associated with one top-level transaction[1].

1. When created, a `Synchronization` must remember its transaction as a `Resource` does. The fact that the `Synchronization` interface derives from `TransactionalObject` is a historical hang-over and does not imply that an OTS server will provide an implicit transaction context when calling a `Synchronization` method.

A `Synchronization` is registered with a transaction by calling `CosTransactions::Coordinator::register_synchronization`.

before_completion

This method is called before the beginning of the two-phase commit protocol. The transaction may be rolled back if this method raises a `CORBA::SystemException`.

after_completion

This method is called after the two-phase commit protocol is complete, regardless of whether the transaction was ultimately committed or rolled back. The method receives the transaction's completion status: `StatusCommitted` or `StatusRolledBack`. As the transaction's `Control`, `Coordinator`, and `Terminator` references may no longer be valid at this point this method should not attempt to use them. Any exceptions raised by `after_completion` are ignored as there is no transaction to affect.

4.4.5 Recovery

Once begun, a transaction must either successfully commit or abort. A transaction must be able to finish, even if one of the participants[1] experiences a failure at any point in the life of the transaction.

OTS Server

ORBACUS OTS does not record any persistent state about a transaction until the transaction has advanced to the `StatusPrepared` state and all `Resources` have voted `VoteCommit`. If the OTS server is shut down before this, the transaction is forgotten and considered rolled back. Note that `Resources` have a responsibility to find out when the OTS server has forgotten about a transaction. If they do not they could wait forever for a `commit` that will never arrive.

Once the commit vote is determined, the OTS server saves the commit decision and list of `Resources` to stable storage. If an OTS server is shut down after this point, the OTS server will resume the two-phase commit on restart.

1. participant here means the transaction `Coordinator`, based in the OTS server, or a `Resource`.

Resource

It is the responsibility of the programmer to insure that `Resources` involved in transactions are written to support recovery.

First, a `Resource` must follow the procedures described for `register_resource` and `prepare` method to insure it can take part in recovery.

When the `Resource` is reactivated on startup, it should initiate transaction recovery by calling `replay_completion` on the `CosTransactions::RecoveryCoordinator` reference it saved when it was registered with the transaction. If `replay_completion` raises `CORBA::OBJECT_NOT_EXIST,` the `Resource` can presume the transaction has been rolled back or forgotten[1]. A robust `Resource` implementation should have some means of reporting such an occurrence to an administrator.

Any other exception from `replay_completion` indicates the `Resource` should try again after some implementation defined interval.

At some point, either in response to the `replay_completion`, or based on its own recovery, the OTS server will invoke either `commit` or `rollback` on the `Resource`. If the `Resource` has made no heuristic decision or the heuristic decision agrees with the transaction outcome, transaction completion proceeds as normal and the `Resource` forgets the transaction. Otherwise the `Resource` raises a heuristic exception and remains in existence until told to `forget`.

A `Resource` ends its association with a transaction by:

- Responding to prepare with `VoteRollback` or `VoteReadOnly`
- Raising no heuristic exception from `commit` or `rollback.`
- Having `forget` called on it.

After a `Resource` has ended it association with the transaction, the OTS server will not contact it again. An implementation can then reclaim the `Resource` object. It no longer needs to be restarted if the server is terminated and restarts.

The `Resource` and server implementation are responsible for the resource life cycle and persistent store management.

1. In addition to just at restart, a `Resource` that has not yet been prepared should query the `RecoveryCoordinator` periodically to insure that the OTS hasn't forgotten about the transaction.

Synchronizations

Specification Limitation

The OTS specification defines `Synchronizations` to be “not persistent so they are not restarted after failure and, as a result, their operations are not invoked during failure processing.” This definition makes it impossible to rely on the use of `Synchronizations` in transactions as they may or may not be involved in the two-phase commit depending on when and which participants in a transaction experience a system failure.

ORBACUS OTS Synchronization Guarantees

ORBACUS OTS saves both references to `Resources` and `Synchronizations` once a transaction is prepared with intent to commit. This insures that the `Synchronization’s` `before_completion` and `after_completion` will always be called after a restart. This is the only way to reliably allow `Synchronizations` to be used in transactions.

If a server process implementing a `Synchronization` does not restart it, any calls an OTS Server makes to `before_completion` and `after_completion` will result in an `CORBA::OBJECT_NOT_EXIST` exception being raised. It is strongly recommended that `Synchronization` implementations be made persistent and be restarted along with the `Resources` a server provides.

CHAPTER 5

X/Open XA Integration

Note: ORBACUS OTS *XA support has changed from the first BETA release. The preliminary information in this chapter is based on the RELNOTES file that accompanies the* ORBACUS OTS *distribution. Please see that file for the latest information*

5.1 Introduction

The X/Open XA specification [9] provides a standard C API to integrate database processing systems (such as Oracle) with transaction management systems (such as OMG OTS).

5.2 The XA API

The XA methods API is accessed via a C structure defined as follows:

```
struct xa_switch_t {
  char name[RMNAMESZ];
  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);
};
```

The functions can be divided as follows:

- `xa_open_entry, xa_close_entry`

  Open and close connections.

- `xa_start_entry, xa_end_entry`

  Associate the current thread of control with the given transaction.

- `xa_prepare_entry, xa_rollback_entry, xa_commit_entry, xa_forget_entry`

  Entry points into the two-phase commit protocol.

- `xa_recover_entry`

  Provide information essential for recovery after a crash.

- `xa_complete_entry`

  Used for asynchronous method completion (not used by ORBacus XA integration).

Implementations of the XA specification (provided by the database vendors) map a connection (established via `xa_open_entry` and `xa_close_entry`) with two general models:

- A process model in which the scope of the XA connection is the process.
- A threaded model in which the scope of the XA connection is the thread.

Some implementations provide both models (such as Oracle, where specifying `threads=yes` in the open string selects the threaded model).

5.3 ORBacus OTS XA Integration

To initialize an application for use with automatic XA integration, add the following to the application code:

```
#include <OB/OTSXA.h>
...
OB::OTSInit(argc, argv);
XA::OTSInit(argc, argv);
```

```
CORBA::ORB_var orb = CORBA::ORB_init(argc, argv);
```

Next, the `XA::Connector` must be resolved:

```
XA::Connector_var connector =
   XA::Connector::_narrow(CORBA::Object_var(
      orb -> resolve_initial_references("XAConnector")));
```

This connector is then used to either create a `ResourceManager`, or connect to a remote `ResourceManager`, depending on whether a threaded or process model is used, respectively.

Another choice that must be made when creating the resource manager is whether a method dispatch thread is automatically associated with XA. This is controlled by the `automatic_association` flag. If `automatic_association` is true, then `CurrentConnection::start` and `CurrentConnection::end` are called automatically for each method invocation thread (the disadvantage is a lack of flexibility).

The API for the `Connector` interface is as follows:

```
local interface Connector
{
     ResourceManager create_resource_manager(
      in string
      in resource_manager_name,
      in XASwitch xa_switch,
      in string open_string,
      in string close_string,
      in ThreadModel thread_model,
      in boolean automatic_association,
      in any optional_impl_specific_param,
      out CurrentConnection current_connection);

    CurrentConnection connect_to_resource_manager(
      in ResourceManager rm,
      in XASwitch xa_switch,
      in string open_string,
      in string close_string,
      in ThreadModel thread_model,
      in boolean automatic_association,
      in any optional_impl_specific_param);
};
```

ORBACUS OTS does not have any implementation-specific parameters.

If automatic association is not selected, then the association of the thread with XA must be handled manually through the use of the `CurrentConnection` object. The interface is:

```
/*local*/ interface CurrentConnection
{
    //
   // xa_start(TMNOFLAGS) or xa_start(TMJOIN)
   //
    void start(in CosTransactions::Coordinator tx,
             in CosTransactions::otid_t otid);

    //
    // xa_end(TMSUSPEND)
    //
    void suspend(in CosTransactions::Coordinator tx,
                 in CosTransactions::otid_t otid);

    //
    // xa_start(TMRESUME)
    //
    void resume(in CosTransactions::Coordinator tx,
       in CosTransactions::otid_t otid);

    //
   // xa_end(TMSUCCESS) or xa_end(TMFAIL)
   //
    void end(in CosTransactions::Coordinator tx,
     in CosTransactions::otid_t otid,
     in boolean success);

    ThreadModel thread_model();
    long rmid();
};
```

The `suspend` and `resume` methods can only be used from the same thread of control. That is, the thread that calls `suspend` must be the same thread that calls `resume`. Therefore this is generally only useful for those applications that are single-threaded (or those applications in which the threads are tightly controlled).

5.3.1 Oracle 8i

ORBACUS OTS has been tested with Oracle 8i 8.1.6. The example applications in `ots/xa/demo/oracle` shows how to integrate the OTS with Oracle and Pro*C. Dynamic resource registration is not supported in this beta release; the `xa_switch_t` for Oracle must be `xaosw`.

For information on Oracle XA integration, please see

```
http://technet.oracle.com/doc/oracle8i_816/appdev.816/a76939/
adga1_xa.htm#622669
```

Pay particular attention to the format of the open string, and restrictions that your application must follow. In particular the following SQL commands must not be used:

- `SQL ENABLE THREADS`
- `SQL ROLLBACK WORK`
- `SQL COMMIT WORK`
- `SET TRANSACTION`
- `SQL CONNECT`
- `SQL ALLOCATE`
- `SQL USE`

In addition:

- DDL statements are not supported.
- Select privilege to the `DBA_PENDING_TRANSACTIONS` must be granted for all Oracle server users specified in the open string.

5.3.2 XA Server Object References

X/Open XA integrated servers must create persistent object references. It is therefore important that the server either uses a static port (for direct binding) or is configured to start via the implementation repository. If this is not done, transaction recovery cannot work on restart and a strong possibility exists for database corruption.

CHAPTER 6

ORBacus OTS Console

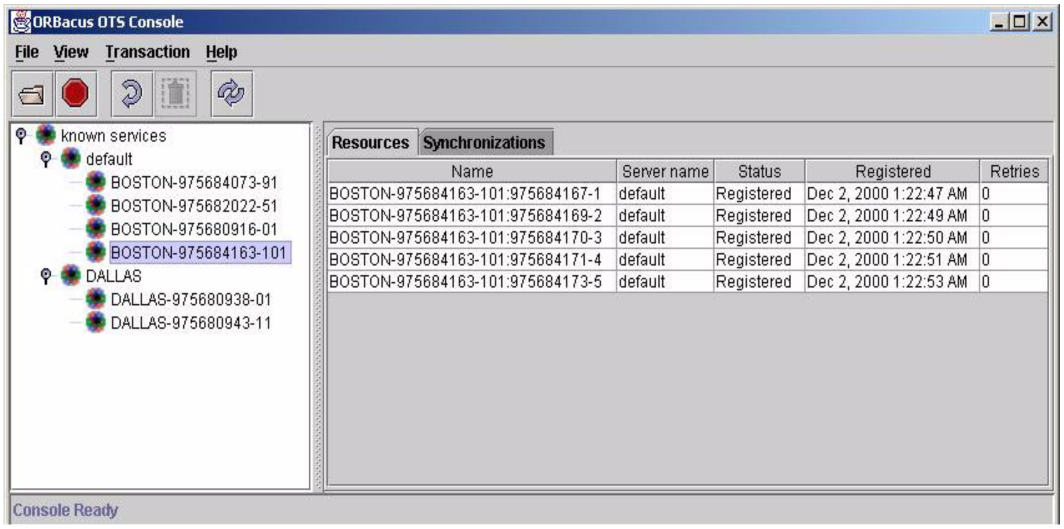
The ORBACUS OTS Console supports the management of all the transactions within an OTS domain. The ORBACUS OTS Console includes the following functionality:

- Shutdown any OTS server in the domain
- Query the state of any transaction on an OTS server
- View the status of each resource associated with a transaction
- View `Synchronizations` registered with a transaction
- Force transaction rollback
- Remove a resource from a transaction

The OMG OTS specification [4] does not define interfaces for transaction administration. Therefore the ORBACUS OTS Console relies on proprietary interfaces to obtain the information displayed.

6.1 The Main Window

The ORBACUS OTS Console main window is shown in Figure 6.1. It contains the following elements:



- known services
  - default
    - BOSTON-975684073-91
    - BOSTON-975682022-51
    - BOSTON-975680916-01
    - BOSTON-975684163-101
  - DALLAS
    - DALLAS-975680938-01
    - DALLAS-975680943-11

Name	Server name	Status	Registered	Retries
BOSTON-975684163-101:975684167-1	default	Registered	Dec 2, 2000 1:22:47 AM	0
BOSTON-975684163-101:975684169-2	default	Registered	Dec 2, 2000 1:22:49 AM	0
BOSTON-975684163-101:975684170-3	default	Registered	Dec 2, 2000 1:22:50 AM	0
BOSTON-975684163-101:975684171-4	default	Registered	Dec 2, 2000 1:22:51 AM	0
BOSTON-975684163-101:975684173-5	default	Registered	Dec 2, 2000 1:22:53 AM	0

Figure 6.1: ORBACUS OTS Console Main Window

Menu bar	Provides access to all the application features.
Toolbar	Shortcuts for the most common menu commands.
Transaction List	The left window pane displays the list of OTS servers in the domain[a]. Servers can be selected to display their active transactions. An individual transaction can be selected to display its details in the right window pane.
Resource Tab	Displays the current status of Resources registered with the selected transaction.
Synchronization Tab	Displays any Synchronizations registered with the selected transaction.
Status bar	Displays the host and port at which the console is connected to the selected ORBACUS OTS server and also displays information regarding currently executing operations.

a. The displayed server list is configurable. The OTS server configured as the `TransactionFactory` initial reference is labeled *default*. (This is subject to change in a later release.)

6.2 The ORBacus OTS Console Menus

6.2.1 The File Menu

The File Menu contains operations that operate on ORBacus OTS servers.

Switch OTS	Displays a dialog box that allows the user to change the default `OBCosTransactions::TransactionFactory`[a] from the one currently displayed. The user may either type in the IOR in URL style syntax, or specify a file containing the IOR.
Shutdown OTS	Terminates the displayed OTS Server by calling the shutdown operation on the server.
Quit	Quits the ORBACUS OTS Console application.

a. This is an OOC proprietary interface that derives from `CosTransactions::TransactionFactory` and has additional administrative methods.

6.2.2 The View Menu

This menu contains operations which allow the user to configure the console display.

Error Window	Toggles the display of the Error Window which displays CORBA exceptions received by the console from the transaction service.
Refresh	Obtains an updated list of transactions for the displayed OTS servers.

6.2.3 The Transaction Menu

This menu contains operations to administratively control a transaction.

Rollback	Rolls back the selected transaction. (This is done by a call to rollback on the transaction's `Coordinator` object.)
Remove Resource	Removes the selected resource from the transaction. (This is done by a call to `remove_resource` on `OBCosTransactions::Coordinator`[a].)

a. This is an OOC proprietary interface that derives from `CosTransactions::Coordinator`.

6.2.4 The Help Menu

This menu is used to access the on-line help facilities.

Help Contents	Displays the main help contents page. From here the user can navigate the entire on-line help system.
About...	Displays version and copyright information.

6.3 The Toolbar

The ORBACUS OTS Console toolbar is shown in Figure 6.2:. It contains buttons for the

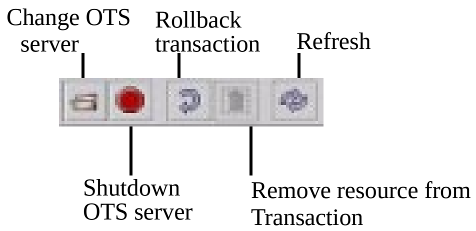


Figure 6.2: ORBACUS OTS **Console Toolbar**

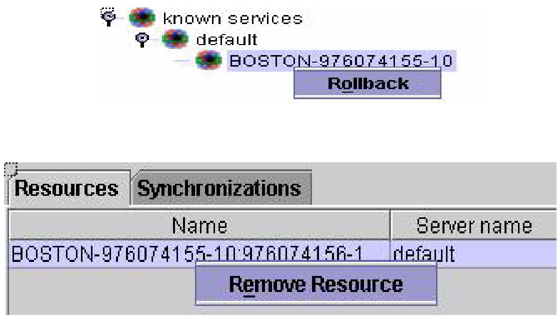
most commonly used menu commands.

6.4 The Popup Menu

Right-clicking on either transactions or `Resources` in the console displays a context sensitive popup menu, as shown in Figure 6.3. This popup menu is a shortcut to the menu commands. For transactions the shortcut is rollback, for `Resources` it is remove.

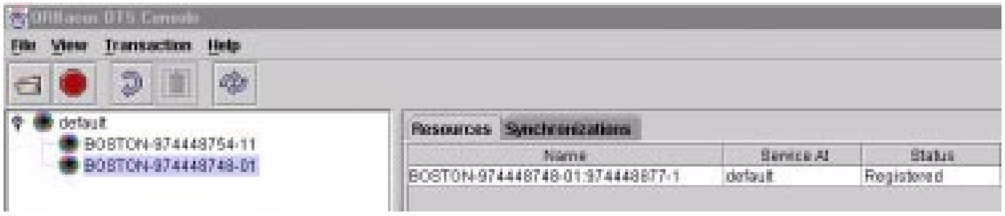
6.5 Transaction and Resource Identification

Figure 6.4 illustrates viewing a transaction and resource with the ORBACUS OTS Console. Under the default tree, there are two transactions. The second one has been highlighted and a single resource for that transaction is shown in the right hand pane.



Name	Server name
BOSTON-976074155-10:976074156-1	default

Figure 6.3: Transaction and Resource Popup Menus



Name	Service At	Status
BOSTON-974448748-01:974448877-1	default	Registered

Figure 6.4: A Transaction and a Registered Resource

6.5.1 Transaction ID

The transaction IDs generated by ORBACUS OTS consist of the originating OTS server name and a numeric component[1]. In the above example, the server name prefix is `BOSTON`. This corresponds to an OTS server that was started with the command line argument `-OTSserver_name` or the property `ooc.ots.server_name` being set to `BOSTON`. Remember every OTS server in an OTS domain must have a unique name.

1. The numeric component consists of a timestamp and a counter value. This coupled with the server name will guarantee a unique transaction id within an OTS domain.

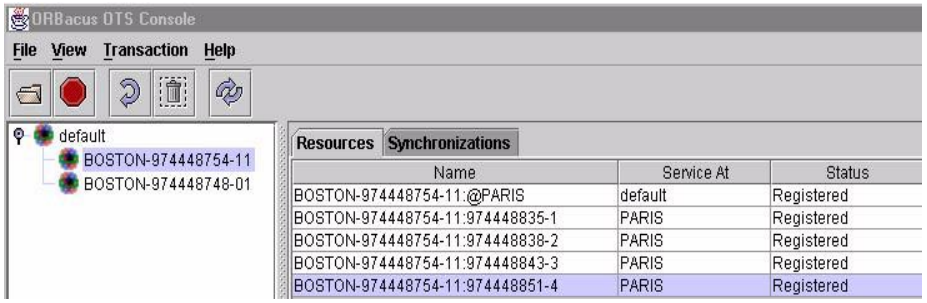
6.5.2 Resource Name

The resource name that the ORBACUS OTS Console displays consists of the enclosing transaction ID and a resource ID. In the example above, the resource name `BOSTON-9744487480-01:974448877-1` denotes a resource that is registered with the transaction `BOSTON-9744487480-01` and the resource ID is `974448877-1`. The **Service At** column shows the name of the OTS server the resource is registered with. In this case a value of `default` in this column indicates the resource has been registered with a coordinator in the `BOSTON` OTS Server.

Interposition

When interposition is used, a subordinate coordinator that is managing many resources will appear as a single resource to the originating transaction coordinator. This optimization can greatly reduce network traffic but can create administrative headaches if resources registered with subordinate coordinators were not directly visible in administrative tools.

ORBACUS OTS provides additional support to view all of the resources registered with a subcoordinator. Figure 6.5 illustrates a transaction that originated at the OTS Server `BOSTON`. The transaction propagated to an application that uses the OTS Server `PARIS`, where an interposed coordinator was created to represent the transaction. Several resources were then registered with the interposed coordinator.



Name	Service At	Status
BOSTON-974448754-11:@PARIS	default	Registered
BOSTON-974448754-11:974448835-1	PARIS	Registered
BOSTON-974448754-11:974448838-2	PARIS	Registered
BOSTON-974448754-11:974448843-3	PARIS	Registered
BOSTON-974448754-11:974448851-4	PARIS	Registered

Figure 6.5: Transaction with an Interposed Coordinator

As far as the OTS Server `BOSTON` is concerned, there is only one resource registered with the transaction `BOSTON-974448754-11` and that is `BOSTON-974448754-11:@PARIS`. The ORBACUS OTS Console displays any resources that are actually interposed coordinators with names that consist of the originating transaction name with `@<interposed-ots-server-name>` appended.

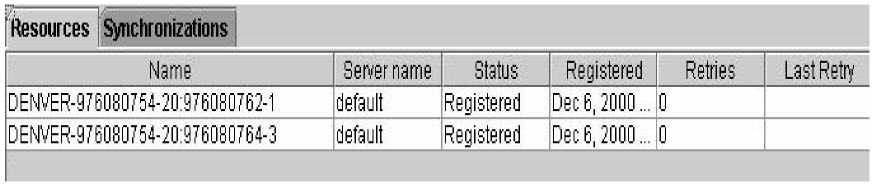
Regular resources, such as the last one `BOSTON-974448754-11:974448851-4,` are given names that identify the transaction they associated with. It is possible to tell that they are registered with the interposed coordinator at `PARIS` by looking at the **Service At** column of the Resources table, which shows PARIS for these resources. The interposed coordinator is the only resource that shows `default` (`BOSTON`) in this column

6.5.3 Resource Properties Tab

The Resource Properties tab displays the following columns:

- Name - The resource name.
- Server Name - The name of the OTS server this resource is directly registered with.
- Status - The status of the resource. This can be one of: `Registered, Preparing, Prepared, Committing, RollingBack`.
- Registered - The timestamp of when the resource was registered with the transaction.
- Retries - The number of times the OTS server has had to retry contacting the resource.
- Last Retry - The timestamp of the last retry on the resource.

These are illustrated in Figure 6.6:



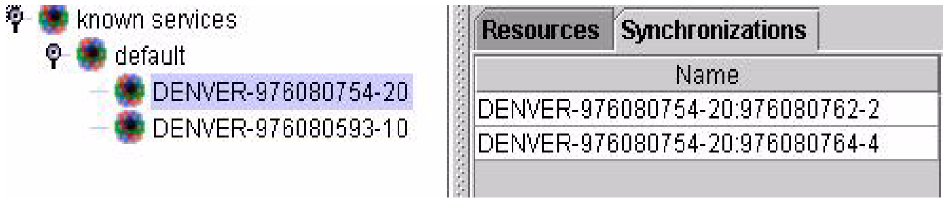
Name	Server name	Status	Registered	Retries	Last Retry
DENVER-976080754-20:976080762-1	default	Registered	Dec 6, 2000 ...	0	
DENVER-976080754-20:976080764-3	default	Registered	Dec 6, 2000 ...	0	

Figure 6.6: Resource Tab

In this figure, the OTS server has not had to retry contacting the resources.

6.5.4 Synchronization Properties Tab

The Synchronizations Properties Tab shows the name of any Synchronizations registered with the transaction as illustrated in Figure 6.7:



Name
DENVER-976080754-20:976080762-2
DENVER-976080754-20:976080764-4

Figure 6.7: Synchronizations Tab

APPENDIX A

CosTransactions Module Reference

A.1 Module CosTransactions

Enums

Status

```
enum Status
{
     StatusActive,
     StatusMarkedRollback,
     StatusPrepared,
     StatusCommitted,
     StatusRolledBack,
     StatusUnknown,
     StatusNoTransaction,
    StatusPreparing,

    StatusCommitting,

    StatusRollingBack

};
```

Status enumeration - used by resources.

Members:

`StatusActive` — Transaction is active

`StatusMarkedRollback` — Transaction is marked for rollback

`StatusPrepared` — Transaction has been prepared

`StatusCommitted` — Transaction has been committed

`StatusRolledBack` — Transaction has been rolled back

`StatusUnknown` — Transaction status is unknown

`StatusNoTransaction` — Transaction is invalid

`StatusPreparing` — Transaction is preparing

`StatusCommitting` — Transaction is committing

`StatusRollingBack` — Transaction is rolling back

Vote

```
enum Vote
{
    VoteCommit,

    VoteRollback,

    VoteReadOnly

};
```

Votes enumeration - used by resources.

Members:

`VoteCommit` — Vote for committing a transaction

`VoteRollback` — Vote for rolling back a transaction

`VoteReadOnly` — Vote in case the transaction did not change a resource

Structs

otid_t

```
struct otid_t
{
    long format_id;
    long bqual_length;
    sequence<octet> tid;
};
```

Structure for transaction identifiers.

Members:

`format_id` — '0' for OSI TP

`bqual_length` — length of identifying part

`tid` — transaction identifier

TransIdentity

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

Transaction identity structure.

Members:

`coord` — The transaction coordinators object reference

`term` — The transaction terminators object reference

`otid` — Transaction identifier structure

PropagationContext

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

Transaction propagation context structure.

Members:

`timeout` — The transactions' timeout value

`current` — The transactions' identity

`parents` — Identities of parent transactions

`implementation_specific_data` — Data needed for administrative purposes

Exceptions

HeuristicRollback

```
exception HeuristicRollback
{
};
```

HeuristicCommit

```
exception HeuristicCommit
{
};
```

HeuristicMixed

```
exception HeuristicMixed
{
};
```

HeuristicHazard

```
exception HeuristicHazard
{
};
```

SubtransactionsUnavailable

```
exception SubtransactionsUnavailable
{
};
```

NotSubtransaction

```
exception NotSubtransaction
{
};
```

Inactive

```
exception Inactive
{
};
```

NotPrepared

```
exception NotPrepared
{
};
```

NoTransaction

```
exception NoTransaction
{
};
```

InvalidControl

```
exception InvalidControl
{
};
```

Unavailable

```
exception Unavailable
```

```
{
};
```

SynchronizationUnavailable

```
exception SynchronizationUnavailable
{
};
```

A.2 Interface CosTransactions::Current

interface Current

```
interface Current
   inherits from CORBA::Current
```

Operations

begin

```
   void begin()
        raises(SubtransactionsUnavailable);
```

commit

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

rollback

```
   void rollback()
        raises(NoTransaction);
```

rollback_only

```
   void rollback_only()
        raises(NoTransaction);
```

get_status

```
Status get_status();
```

get_transaction_name

```
string get_transaction_name();
```

set_timeout

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

get_timeout

```
unsigned long get_timeout();
```

get_control

```
Control get_control();
```

suspend

```
Control suspend();
```

A.3 Interface CosTransactions::TransactionFactory

interface **TransactionFactory**.

Factory interface for transaction objects.

Operations

create

```
Control create(in unsigned long time_out);
```

Create a transaction object.

Parameters:

`time_out` — Timeout in seconds for transaction completion.

Returns:

Transaction `Control` object.

recreate

```
Control recreate(in PropagationContext ctx);
```

Recreate a transaction object from a propagation context.

Parameters:

`ctx` — Propagation context.

Returns:

Transaction `Control` object.

A.4 Interface CosTransactions::Control

`interface` **`Control`**

Transaction `Control` interface. Used to retrieve References for a transactions' `Terminator` or `Coordinator`.

Operations

get_terminator

```
Terminator get_terminator()
    raises(Unavailable);
```

Get the reference of a transactions' `Terminator`.

Returns:

Terminator object reference.

Raises:

`Unavailable` — Raised if no `Terminator` available.

get_coordinator

```
Coordinator get_coordinator()
    raises(Unavailable);
```

Get the reference of a transactions' `Coordinator`.

Returns:

Coordinator object reference.

Raises:

`Unavailable` — Raised if no `Coordinator` available.

A.5 Interface CosTransactions::Terminator

`interface` **`Terminator`**

Transaction `Terminator` interface. Responsible for commiting or rolling back a `Transaction`.

Operations

commit

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

Commit transaction - make any changes to resources permanent.

Parameters:

`report_heuristic` — Flag to indicate if heuristic decision information shall be reported.

Raises:

`HeuristicMixed` — Raised if part of the resources implicated voted `VoteCommit` and others voted `VoteRollback`.

`HeuristicHazard` — Raised if a hazard condition occured - from some resources implicated it could not be determined how they reacted.

rollback

```
void rollback();
```

Roll back the transaction - undo any changes in implicated resources.

A.6 Interface CosTransactions::Coordinator

`interface` **`Coordinator`**

Transaction `Coordinator` interface. Responsible for `Resource` management and `Status` information.

Operations

get_status

`Status` **`get_status`**`();`

Get `Status` of the `Transaction`.

Returns:

`Status` of the `Transaction`.

get_parent_status

`Status` **`get_parent_status`**`();`

Get `Status` of the associated parent `Transaction`.

Returns:

`Status` of the associated parent `Transaction`.

get_top_level_status

`Status` **`get_top_level_status`**`();`

Get `Status` of the associated top level `Transaction`.

Returns:

`Status` of the associated top level `Transaction`.

is_same_transaction

```
boolean is_same_transaction(in Coordinator tc);
```

Check if `Transaction` is the same than the `Transaction` associated with the given `Coordinator` object reference.

Parameters:

`tc` — Object reference of a `Transaction` `Coordinator` object.

Returns:

`true` if `Transaction` is the same.

is_related_transaction

```
boolean is_related_transaction(in Coordinator tc);
```

Check if `Transaction` is related to the `Transaction` associated with the given `Coordinator` object reference.

Parameters:

`tc` — Object reference of a `Transaction` `Coordinator` object.

Returns:

`true` if `Transaction` is related.

is_ancestor_transaction

```
boolean is_ancestor_transaction(in Coordinator tc);
```

Check if `Transaction` is an anchestor of the `Transaction` associated with the given `Coordinator` object reference.

Parameters:

`tc` — Object reference of a `Transaction` `Coordinator` object.

Returns:

TRUE if `Transaction` is an anchestor.

is_descendant_transaction

```
boolean is_descendant_transaction(in Coordinator tc);
```

Check if `Transaction` is a descendant of the `Transaction` associated with the given `Coordinator` object reference.

Parameters:

`tc` — Object reference of a `Transaction` `Coordinator` object.

Returns:

TRUE if `Transaction` is a descandant.

is_top_level_transaction

```
boolean is_top_level_transaction();
```

Check if `Transaction` is a top level `Transaction`.

Returns:

TRUE if `Transaction` is top level, `false` if is a child `Transaction`.

hash_transaction

```
unsigned long hash_transaction();
```

Get unique `hash` value of the `Transaction`.

Returns:

unique `hash` value.

hash_top_level_tran

```
unsigned long hash_top_level_tran();
```

Get unique `hash` value of the corresponding top level `Transaction`.

Returns:

unique `hash` value.

register_resource

```
RecoveryCoordinator register_resource(in Resource r)
    raises(Inactive);
```

Register a `Resource` object with the `Transaction`.

Parameters:

`r` — Object reference of a `Resource` object.

Raises:

`Inactive` — Raised if `Transaction` inactive.

register_synchronization

```
void register_synchronization(in Synchronization sync)
    raises(Inactive,
        SynchronizationUnavailable);
```

Register a `Synchronization` object with the `Transaction`.

Parameters:

`sync` — Object reference of a `Synchronization` object.

Raises:

`Inactive` — Raised if `Transaction` inactive.

`SynchronizationUnavailable` — Raised if no `Synchronization` available.

register_subtran_aware

```
void register_subtran_aware(in SubtransactionAwareResource r)
    raises(Inactive,
       NotSubtransaction);
```

Register a `SubtransactionAwareResource` with the `Transaction`.

Raises:

`Inactive` — Raised if `Transaction` inactive.

`NotSubtransaction` — Raised if `Transaction` is not a child transaction.

rollback_only

```
void rollback_only()
    raises(Inactive);
```

Mark a `Transaction` for rollback as the only valid completion. XXX

Raises:

`Inactive` — Raised if `Transaction` inactive.

get_transaction_name

```
string get_transaction_name();
```

Get the name of a `Transaction`.

Returns:

Name string of the `Transaction`.

create_subtransaction

```
Control create_subtransaction()
    raises(SubtransactionsUnavailable,
       Inactive);
```

Get the object reference of the `Control` of a `Transaction`.

Raises:

`SubtransactionsUnavailable` — Raised if no subtransacions available.

`Inactive` — Raised if `Transaction` is inactive.XXX

get_txcontext

```
PropagationContext get_txcontext()
    raises(Unavailable);
```

Get the `PropagationContext` of a `Transaction`.

Raises:

`Unavailable` — Raised if no `PropagationContext` available.

A.7 Interface CosTransactions::RecoveryCoordinator

`interface` **`RecoveryCoordinator`**

Transaction `RecoveryCoordinator` interface. Responsible for recovery of resources

Operations

replay_completion

```
Status replay_completion(in Resource r)
    raises(NotPrepared);
```

Replay the completion sequence of an implicated `Resource`.

Parameters:

`r` — Object reference of implicated `Resource`.

Raises:

`NotPrepared` — Raised if `Resource` was not prepared for completion.

A.8 Interface CosTransactions::TransactionalObject

```
interface TransactionalObject
```

A.9 Interface CosTransactions::Resource

```
interface Resource
```

Operations

prepare

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

rollback

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

commit

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

commit_one_phase

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

forget

```
void forget();
```

A.10 Interface CosTransactions::SubtransactionAwareResource

```
interface SubtransactionAwareResource
   inherits from CosTransactions::Resource
```

Operations

commit_subtransaction

```
   void commit_subtransaction(in Coordinator parent);
```

rollback_subtransaction

```
   void rollback_subtransaction();
```

A.11 Interface CosTransactions::Synchronization

```
interface Synchronization
   inherits from CosTransactions::TransactionalObject
```

Operations

before_completion

```
   void before_completion();
```

after_completion

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

References

[1] Gorton, I. 2000. *Enterprise Transaction Processing Systems*. Harlow, England: Addison-Wesley Pearson Education.

[2] Gray, J. and A. Reuter. 1993. *Transaction Processing: Concepts and Techniques*. San Francisco: Morgan Kaufmann.

[3] Henning, M., and S. Vinoski. 1999. *Advanced CORBA Programming with C++*. Reading, MA: Addison-Wesley

[4] Object Management Group. 1999. *Object Transaction Service*. Waltham, MA: Object Management Group

This is the OTS 1.1 specification. It is available for download at ftp://ftp.omg.org/pub/docs/ptc/99-10-07.pdf

[5] Object Management Group. 2000. *Object Transaction Service*. Waltham, MA: Object Management Group

This is the current draft OTS 1.2 specification. It is available for download at ftp://ftp.omg.org/pub/docs/ptc/00-09-04.pdf. This document should be superceded by another draft by early 2001.

[6] Object Oriented Concepts. 2000. *ORBacus 4 User's Manual*. Billerica, MA: ObjectOriented Concepts.

This is available for download at http://www.ooc.com/ob

[7] Object Oriented Concepts. 2000. *ORBacus JThreads/C++ Manual*. Billerica, MA: ObjectOriented Concepts.

This is available for download at http://www.ooc.com/jtc

[8] Slama, D., et al. 1999. *Enterprise CORBA*. Upper Saddle River, NJ: Prentice Hall PTR.

[9] X/Open Company Ltd. 1991. *Distributed Transaction Processing: The XA Specification.* Berkshire, UK: X/Open Company Ltd.

Details are available at http://www.opengroup.org/pubs/catalog/c193.htm.