BEA Tuxedo®

Using CORBA Transactions

Version 10.0
Document Released: September 28, 2007

Contents

3. Transactions in CORBA Server Applications

4. Transactions in CORBA Client Applications

5. Administering Transactions

CHAPTER 1

Introducing Transactions

This topic includes the following sections:

- Overview of Transactions in BEA Tuxedo CORBA Applications
- When to Use Transactions
- How to Use Transactions in BEA Tuxedo CORBA Applications
- Writing a Transactions Sample Application

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Overview of Transactions in BEA Tuxedo CORBA Applications

This topic includes the following sections:

ACID Properties of Transactions

One of the most fundamental features of the BEA Tuxedo system is transaction management. Transactions are a means to guarantee that database transactions are completed accurately and that they take on all the ACID properties (atomicity, consistency, isolation, and durability) of a high-performance transaction. BEA Tuxedo protects the integrity of your transactions by providing a complete infrastructure for ensuring that database updates are done accurately, even across a variety of resource managers (RMs). If any one of the operations fails, the entire set of operations is rolled back.

Resource Manager

A Resource Manager (RM) is a data repository, such as a database management system or the BEA Tuxedo system's Application Queuing Manager, with tools for accessing the data. The BEA Tuxedo system uses one or more RMs to maintain the state of an application. For example, bank records in which account balances are maintained are kept in an RM. When the state of the application changes through a service that allows a customer to withdraw money from an account, the new balance in the account is recorded in the appropriate RM.

The BEA Tuxedo system helps you manage transactions involving resource managers that support the XA interface. To coordinate all the operations performed and all the modules affected by a transaction, the BEA Tuxedo system plays the role of the Transaction Manager (TM).

The TM coordinates global transactions involving system-wide resources. Local resource managers (RMs) are responsible for individual resources. The Transaction Manager Server

(TMS) begins, commits, and aborts transactions involving multiple resources. The application code uses the normal embedded SQL interface to the RM to perform reads and updates. The TMS uses the XA interface to the RM to perform the work of a global transaction.

Supported Programming Model

BEA Tuxedo supports the Object Management Group Common Object Request Broker (CORBA) in C++, in compliance with the *The Common Object Request Broker: Architecture and Specification*, Revision 2.4.2, January 2001.

Supported API Model

BEA Tuxedo supports the CORBA services Object Transaction Service (OTS). BEA Tuxedo provides a C++ interface to the OTS and is based on the OTS. The OTS is accessed through the `org.omg.CosTransactions.Current` environmental object. For information about using the `TransactionCurrent` environmental object, see the "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

Note: BEA Tuxedo also supports use of the CORBA Interoperable Naming Service (INS) bootstrapping mechanism. For information on INS, see the "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

Support for Business Transactions

OTS provides the following support for your business transactions:

- Creates a global transaction identifier when a client application initiates a transaction.
- Works with the BEA Tuxedo infrastructure to track objects that are involved in a transaction and, therefore, need to be coordinated when the transaction is ready to commit.
- Notifies the resource managers—which are, most often, databases—when they are accessed on behalf of a transaction. Resource managers then lock the accessed records until the end of the transaction.
- Orchestrates the two-phase commit when the transaction completes, which ensures that all the participants in the transaction commit their updates simultaneously. It coordinates the commit with any databases that are being updated using Open Group's XA protocol. Almost all relational databases support this standard.
- Executes the rollback procedure when the transaction must be stopped.

- Executes a recovery procedure when failures occur. It determines which transactions were active in the machine at the time of the crash, and then determines whether the transaction should be rolled back or committed.

Distributed Transactions and the Two-Phase Commit Protocol

BEA Tuxedo CORBA supports distributed transactions and the two-phase commit protocol for enterprise applications. A *distributed transaction* is a transaction that updates multiple resource managers (such as databases) in a coordinated manner. The *two-phase commit protocol (2PC)* is a method of coordinating a single transaction across one or more resource managers. It guarantees data integrity by ensuring that transactional updates are committed in all of the participating databases, or are fully rolled back out of all the databases, reverting to the state prior to the start of the transaction.

When to Use Transactions

Transactions are appropriate in the situations described in the following list. Each situation describes a transaction model supported by BEA Tuxedo CORBA.

- The client application needs to make invocations on several objects, which may involve write operations to one or more databases. If any one invocation is unsuccessful, any state that is written (either in memory or, more typically, to a database) must be rolled back.

  For example, consider a travel agent application. The client application needs to arrange for a journey to a distant location; for example, from Strasbourg, France, to Alice Springs, Australia. Such a journey would inevitably require multiple individual flight reservations. The client application works by reserving each individual segment of the journey in sequential order; for example, Strasbourg to Paris, Paris to New York, New York to Los Angeles. However, if any individual flight reservation cannot be made, the client application needs a way to cancel all the flight reservations made up to that point.

- The client application needs a conversation with an object managed by the server application, and the client application needs to make multiple invocations on a specific object instance. The conversation may be characterized by one or more of the following:

  – Data is cached in memory or written to a database during or after each successive invocation.

  – Data is written to a database at the end of the conversation.

  - The client application needs the object to maintain an in-memory context between each invocation; that is, each successive invocation uses the data that is being maintained in memory across the conversation.
  - At the end of the conversation, the client application needs the ability to cancel all database write operations that may have occurred during or at the end of the conversation.

  For example, consider an Internet-based online shopping cart application. Users of the client application browse through an online catalog and make multiple purchase selections. When the users are done choosing all the items they want to buy, they proceed to check out and enter their credit card information to make the purchase. If the credit card check fails, the shopping application needs a way to cancel all the pending purchase selections in the shopping cart, or roll back any purchase transactions made during the conversation.

- Within the scope of a single client invocation on an object, the object performs multiple edits to data in a database. If one of the edits fails, the object needs a mechanism to roll back all the edits. (In this situation, the individual database edits are not necessarily CORBA.)

  For example, consider a banking application. The client invokes the transfer operation on a teller object. The transfer operation requires the teller object to make the following invocations on the bank database:

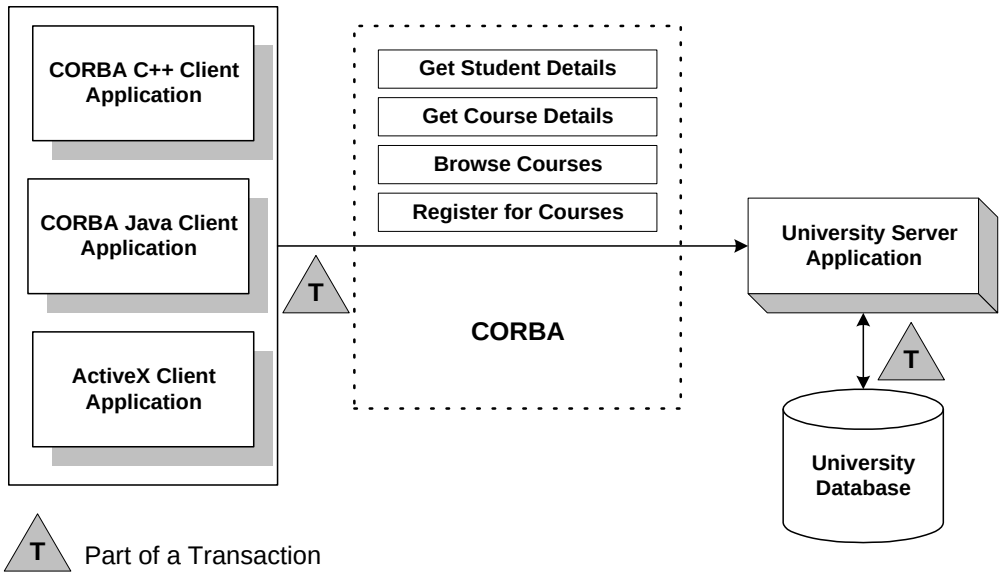
  - Invoking the debit method on one account.
  - Invoking the credit method on another account.

  If the credit invocation on the bank database fails, the banking application needs a way to roll back the previous debit invocation.

How to Use Transactions in BEA Tuxedo CORBA Applications

Figure 1-1 illustrates transactions in a BEA Tuxedo CORBA application.

Figure 1-1 Transactions in a BEA Tuxedo CORBA Application



The way you use transactions differs depending on whether you use the BEA bootstrapping mechanism or the Interoperable Naming Service (INS) bootstrapping mechanism.

Note: You should use the BEA bootstrapping mechanism if you are using BEA Tuxedo CORBA client software. You should use the INS bootstrapping mechanism if you are using a third-party client.

How to Use Transactions When Using the BEA Bootstrapping Mechanism

When the BEA proprietary Bootstrapping mechanism is used, you use a basic transaction in the following way:

1. The client application uses the Bootstrap object to return an object reference to the TransactionCurrent object for the BEA Tuxedo domain.
2. A client application begins a transaction using the `Tobj::TransactionCurrent::begin()` operation, and issues a request to the CORBA interface through the TP Framework. All operations on the CORBA interface execute within the scope of a transaction.

- If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught.
- If all the changes that need to occur have taken place successfully, and the state of the database (or objects) is consistent, then the transaction should be committed; otherwise, the transaction should be rolled back.
- The client application commits the current transaction using the `Tobj::TransactionCurrent::commit()` operation. This operation ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

3. The `Tobj::TransactionCurrent:commit()` operation causes the TP Framework to call the transaction manager to complete the transaction.
4. The transaction manager is responsible for coordinating with the resource managers to update the database.

How to Use Transactions When Using the INS Bootstrapping Mechanism

When you use CORBA services Interoperable Naming Service (INS) bootstrapping mechanism is used, you use a basic transaction in the following way:

1. The client application uses the `ORB::resolve_initial_references()` operation to get a FactoryFinder object for the BEA Tuxedo domain.
2. The client application uses the FactoryFinder to get a TransactionFactory.

   Note: The TransactionFactory returns objects that adhere to the standard CORBA Services Transaction Service interfaces instead of the BEA delegated interfaces. This means that a third-party client can use their ORB's `resolve_initial_references()` function to get the TransactionFactory from a BEA Tuxedo CORBA server and use stubs generated from standard OMG IDL to act on the instances returned.

3. The client application then uses the `create()` operation on the TransactionFactory to begin a transaction and issues a request to the CORBA interface through the TP Framework.
4. From the Control object returned from the `create()` operation, the client application uses the `get_terminator()` operation to get the transaction Terminator interface.
5. The client application then uses the `commit()` or `rollback()` operation on the Terminator interface to end or abort the transaction. The `commit()` operation causes the TP Framework to call the transaction manager to complete the transaction.

6. The transaction manager is responsible for coordinating with the resource managers to update the database.

Note: All operations on the CORBA interface execute within the scope of a transaction.

- If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught.
- If all the changes that need to occur have taken place successfully, and the state of the database (or objects) is consistent, then the transaction should be committed; otherwise, the transaction should be rolled back.
- The client application commits the current transaction using the `Terminator::commit()` operation. This operation ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

Note: For more information on INS, see the "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

Writing a Transactions Sample Application

This topic includes the following sections:

Workflow for the Transactions Sample Application

In the Transactions sample CORBA application, the operation of registering for courses is executed within the scope of a transaction. The transaction model used in the Transactions sample application is a combination of the conversational model and the model in which a single client invocation makes multiple individual operations on a database.

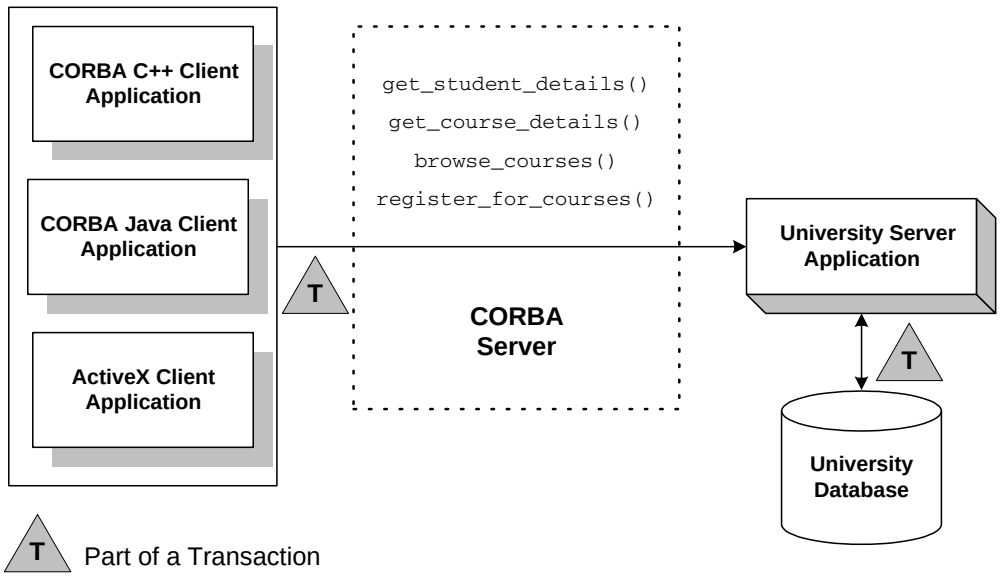
The Transactions sample application works in the following way:

1. Students submit a list of courses for which they want to be registered.
2. For each course in the list, the server application checks whether:
   - The course is in the database.
   - The student is already registered for a course.
   - The student exceeds the maximum number of credits the student can take.

3. One of the following occurs:
   - If the course meets all the criteria, the server application registers the student for the course.
   - If the course is not in the database or if the student is already registered for the course, the server application adds the course to a list of courses for which the student could not be registered. After processing all the registration requests, the server application returns the list of courses for which registration failed. The client application can then choose to either commit the transaction (thereby registering the student for the courses for which registration request succeeded) or to roll back the transaction (thus, not registering the student for any of the courses).
   - If the student exceeds the maximum number of credits the student can take, the server application returns a `TooManyCredits` user exception to the client application. The client application provides a brief message explaining that the request was rejected. The client application then rolls back the transaction.

Figure 1-2 illustrates how the Transactions sample application works.

Figure 1-2 Transactions Sample Application



The Transactions sample application shows two ways in which a transaction can be rolled back:

- **Nonfatal**. If the registration for a course fails because the course is not in the database, or because the student is already registered for the course, the server application returns the numbers of those courses to the client application. The decision to roll back the transaction lies with the user of the client application (and the Transaction client application code rolls back the transaction automatically in this case).
- **Fatal**. If the registration for a course fails because the student exceeds the maximum number of credits he or she can take, the server application generates a CORBA exception and returns it to the client. The decision to roll back the transaction also lies with the client application.

  Thus, the Transactions sample application also shows how to implement user-defined CORBA exceptions. For example, if the student tries to register for a course that would exceed the maximum number of courses for which the student can register, the server application returns the `TooManyCredits` exception. When the client application receives this exception, the client application rolls back the transaction automatically.

Note: For information about how transactions are implemented in BEA Tuxedo CORBA applications, see the Transactions Sample in the BEA Tuxedo online documentation.

Development Steps

This topic describes the following development steps for writing a BEA Tuxedo application that contains transaction processing code:

- Step 1: Writing the OMG IDL
- Step 2: Defining Transaction Policies for the Interfaces
- Step 3: Writing the Server Application
- Step 4: Writing the Client Application
- Step 5: Creating a Configuration File

The Transactions sample application is used to demonstrate these development steps. The source files for the Transactions sample application are located in the `\samples\corba\university` directory of the BEA Tuxedo software. For information about building and running the Transactions sample application, see the Transactions Sample in the BEA Tuxedo online documentation.

Step 1: Writing the OMG IDL

You need to specify interfaces involved in transactions in Object Management Group (OMG) Interface Definition Language (IDL) just as you would any other CORBA interface. You must also specify any user exceptions that might occur from using the interface.

For the Transactions sample application, you would define in OMG IDL the `Registrar` interface and the `register_for_courses()` operation. The `register_for_courses()` operation has a parameter, `NotRegisteredList`, which returns to the client application the list of courses for which registration failed. If the value of `NotRegisteredList` is empty, then the client application commits the transaction. You also need to define the `TooManyCredits` user exception.

Listing 1-1 includes the OMG IDL for the Transactions sample application.

Listing 1-1 OMG IDL for the Transactions Sample Application

```
#pragma prefix "beasys.com"
module UniversityT

{
       typedef unsigned long CourseNumber;
       typedef sequence<CourseNumber> CourseNumberList;

       struct CourseSynopsis
       {
              CourseNumber   course_number;
              string         title;
       };
       typedef sequence<CourseSynopsis> CourseSynopsisList;

       interface CourseSynopsisEnumerator
       {
       //Returns a list of length 0 if there are no more entries
       CourseSynopsisList get_next_n(
              in  unsigned long number_to_get, // 0 = return all
              out unsigned long number_remaining
       );
```

```
        void destroy();
        };
        typedef unsigned short Days;
        const Days MONDAY    =  1;
        const Days TUESDAY   =  2;
        const Days WEDNESDAY =  4;
        const Days THURSDAY  =  8;
        const Days FRIDAY    = 16;
    }
    //Classes restricted to same time block on all scheduled days,
    //starting on the hour

    struct ClassSchedule
    {
        Days           class_days; // bitmask of days
        unsigned short start_hour; // whole hours in military time
        unsigned short duration;   // minutes
    };

    struct CourseDetails
    {
        CourseNumber   course_number;
        double         cost;
        unsigned short number_of_credits;
        ClassSchedule  class_schedule;
        unsigned short number_of_seats;
        string         title;
        string         professor;
        string         description;
    };
        typedef sequence<CourseDetails> CourseDetailsList;
        typedef unsigned long StudentId;

    struct StudentDetails
    {
        StudentId         student_id;
        string            name;
        CourseDetailsList registered_courses;
    };
```

```
enum NotRegisteredReason
{
       AlreadyRegistered,
       NoSuchCourse
};

struct NotRegistered
{
       CourseNumber        course_number;
       NotRegisteredReason not_registered_reason;
};
       typedef sequence<NotRegistered> NotRegisteredList;

exception TooManyCredits
{
       unsigned short maximum_credits;
};

//The Registrar interface is the main interface that allows
//students to access the database.
interface Registrar
{
       CourseSynopsisList
       get_courses_synopsis(
              in string                    search_criteria,
              in unsigned long             number_to_get,
              out unsigned long            number_remaining,
              out CourseSynopsisEnumerator rest
);

       CourseDetailsList get_courses_details(in CourseNumberList
        courses);
       StudentDetails get_student_details(in StudentId student);
       NotRegisteredList register_for_courses(
              in StudentId        student,
              in CourseNumberList courses
       ) raises (
              TooManyCredits
       );

};
```

```
// The RegistrarFactory interface finds Registrar interfaces.

interface RegistrarFactory
{
       Registrar find_registrar(
       );
};
```

Step 2: Defining Transaction Policies for the Interfaces

Transaction policies are used on a per-interface basis. During design, it is decided which interfaces within a BEA Tuxedo application will handle transactions. Table 1-1 describes the CORBA transaction policies.

Table 1-1 CORBA Transaction Policies

Transaction Policy	Description
`always`	The interface must always be part of a transaction. If the interface is not part of a transaction, a transaction will be automatically started by the TP Framework.
`ignore`	The interface is not transactional. However, requests made to this interface within a scope of a transaction are allowed. The `AUTOTRAN` parameter, specified in the `UBBCONFIG` file for this interface, is ignored.
`never`	The interface is not transactional. Objects created for this interface can never be involved in a transaction. The BEA Tuxedo system generates an exception (`INVALID_TRANSACTION`) if an interface with this policy is involved in a transaction.
`optional`	The interface may be transactional. Objects can be involved in a transaction if the request is transactional. This transaction policy is the default.

During development, you decide which interfaces will execute in a transaction by assigning transaction policies. You specify transaction policies in the Implementation Configuration File (ICF). A template ICF file is created by the `genicf` command. For more information about the ICFs, see "Implementation Configuration File (ICF)" in the *CORBA Programming Reference*.

In the Transactions sample application, the transaction policy of the `Registrar` interface is set to `always`.

Step 3: Writing the Server Application

When using transactions in server applications, you need to write methods that implement the interface's operations. In the Transactions sample application, you would write a method implementation for the `register_for_courses()` operation.

If your BEA Tuxedo application uses a database, you need to include in the server application code that opens and closes an XA resource manager. These operations are included in the `Server::initialize()` and `Server::release()` operations of the Server object. Listing 1-2 shows the portion of the code for the Server object in the Transactions sample application that opens and closes the XA resource manager.

Note: For a complete example of a CORBA server application that implements transactions, see the Transactions Sample in the BEA Tuxedo online documentation.

Listing 1-2 C++ Server Object in Transactions Sample Application

```
CORBA::Boolean Server::initialize(int argc, char* argv[])
{
       TRACE_METHOD("Server::initialize");
       try {
              open_database();
              begin_transactional();
              register_fact();
              return CORBA_TRUE;
}
       catch (CORBA::Exception& e) {
              LOG("CORBA exception : " <<e);
       }
       catch (SamplesDBException& e) {
              LOG("Can't connect to database");
       }
       catch (...) {
              LOG("Unexpected database error : " <<e);
       }
       catch (...) {
```

```
                LOG(“Unexpected exception”);
        }
        cleanup();
        return CORBA_FALSE;
}

void Server::release()
{
        TRACE_METHOD(“Server::release”);
        cleanup();
}
static void cleanup()
{
        unregister_factory();
        end_transactional();
        close_database();
}
//Utilities to manage transaction resource manager

CORBA::Boolean s_became_transactional = CORBA_FALSE;
static void begin_transactional()
{
        TP::open_xa_rm();
        s_became_transactional = CORBA_TRUE;
}
static void end_transactional()
{
        if(!s_became_transactional){
        return//cleanup not necessary
}
try {
        TP::close_xa_rm ();
}
        catch (CORBA::Exception& e) {
                LOG(“CORBA Exception : “ << e);
        }
        catch (...) {
```

```
            LOG("unexpected exception");
        }
        s_became_transactional = CORBA_FALSE;
    }
```

Step 4: Writing the Client Application

The client application needs code that performs the following tasks:

1. Obtains a reference to the TransactionCurrent object from the Bootstrap object.
2. Begins a transaction by invoking the `Tobj::TransactionCurrent::begin()` operation on the TransactionCurrent object.
3. Invokes operations on the object. In the Transactions sample application, the client application invokes the `register_for_courses()` operation on the Registrar object, passing a list of courses.

Listing 1-3 shows the portion of the CORBA C++ client applications in the Transactions sample application that illustrates the development steps for transactions.

Note: The sample code shown in Listing 1-3 illustrates how to use the BEA bootstrapping mechanism. For information on how to use the INS bootstrapping mechanism, see the "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

Listing 1-3 Transactions Code for CORBA C++ Client Applications

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var transaction_current_oref=
     CosTransactions::Current::_narrow(var_transaction_current_oref.in());
//Begin the transaction
var_transaction_current_oref->begin();
try {
//Perform the operation inside the transaction
    pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     ...
//If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     var_transaction_current_ref->commit(report_heuristics);
    }
```

```
catch (...) {
//If the operation has problems executing, rollback the
//transaction. Then throw the original exception again.
//If the rollback fails, ignore the exception and throw the
//original exception again.
try {
     var_transaction_current_ref->rollback();
    }
catch (...) {
            TP::userlog("rollback failed");
            }
throw;
}
```

Step 5: Creating a Configuration File

You need to add the following information to the configuration file for a transactional BEA Tuxedo application:

- In the `GROUPS` section:
  - In the `OPENINFO` parameter, include the information needed to open the resource manager for the database. You obtain this information from the product documentation for your database. Note that the default version of the `com.beasys.Tobj.Server.initialize` method automatically opens the resource manager.
  - In the `CLOSEINFO` parameter, include the information needed to close the resource manager for the database. By default, the `CLOSEINFO` parameter is empty.
  - Specify the `TMSNAME` and `TMSCOUNT` parameters to associate the XA resource manager with a specified server group.
- In the `SERVERS` section, define a server group that includes both the server application that includes the interface and the server application that manages the database. This server group needs to be specified as transactional.
- Include the pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter. For more information about the transaction log, see Chapter 5, "Administering Transactions."

Listing 1-4 includes the portions of the configuration file that define this information for the Transactions sample application.

Listing 1-4 Configuration File for Transactions Sample Application

```
*RESOURCES
       IPCKEY    55432
       DOMAINID  university
       MASTER    SITE1
       MODEL     SHM
       LDBAL     N
       SECURITY  APP_PW

*MACHINES
       BLOTTO
       LMID = SITE1
       APPDIR = C:\TRANSACTION_SAMPLE
       TUXCONFIG=C:\TRANSACTION_SAMPLE\tuxconfig
       TLOGDEVICE=C:\APP_DIR\TLOG
       TLOGNAME=TLOG
       TUXDIR="C:\tuxdir"
       MAXWSCLIENTS=10

*GROUPS
       SYS_GRP
         LMID      = SITE1
         GRPNO     = 1
       ORA_GRP
         LMID      = SITE1
         GRPNO     = 2

       OPENINFO  = "ORACLE_XA:Oracle_XA+SqlNet=ORCL+Acc=P
       /scott/tiger+SesTm=100+LogDir=.+MaxCur=5"
       CLOSEINFO = ""
       TMSNAME   = "TMS_ORA"
       TMSCOUNT  = 2

*SERVERS
       DEFAULT:
       RESTART = Y
       MAXGEN  = 5

       TMSYSEVT
```

```
    SRVGRP  = SYS_GRP
    SRVID   = 1

  TMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 2
    CLOPT   = "-A -- -N -M"

  TMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 3
    CLOPT   = "-A -- -N"

  TMFFNAME
    SRVGRP  = SYS_GRP
    SRVID   = 4
    CLOPT   = "-A -- -F"

  TMIFRSVR
    SRVGRP  = SYS_GRP
    SRVID   = 5


  UNIVT_SERVER
    SRVGRP  = ORA_GRP
    SRVID   = 1
    RESTART = N

  ISL
    SRVGRP  = SYS_GRP
    SRVID   = 6
    CLOPT   = -A -- -n //MACHINENAME:2500
*SERVICES
```

For information about the transaction log and defining parameters in the Configuration file, see Chapter 5, “Administering Transactions.”

CHAPTER 2

Transaction Service

This topic includes the following sections:

This topic provides the information that programmers need to write transactional CORBA applications for the BEA Tuxedo system. Before you begin, you should read Chapter 1, "Introducing Transactions."

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

About the Transaction Service

BEA Tuxedo provides a Transaction Service that supports transactions in CORBA applications. The Transaction Service provides an implementation of the CORBA Services Transaction Service that is described in the OMG CORBA Services *Transaction Service Specification*. This specification defines the interfaces for an object service that provides transactional functions.

Capabilities and Limitations

This topic includes the following sections:

- Lightweight Clients with Delegated Commit
- Support for Third-Party Clients Using INS
- Multithreaded Transaction Client Support
- Transaction Integrity
- Transaction Termination
- Flat Transactions
- Interoperability Between CORBA Remote Clients and the BEA Tuxedo Domain
- Intradomain and Interdomain Interoperability
- Network Interoperability
- Relationship of the Transaction Service to Transaction Processing
- Process Failure
- General Constraints

These sections describe the capabilities and limitations of the Transaction Service that supports CORBA applications.

Lightweight Clients with Delegated Commit

A *lightweight client* runs on a single-user, unmanaged desktop system that has irregular availability. Owners may turn their desktop systems off when they are not in use. These single-user, unmanaged desktop systems should not be required to perform network functions such as transaction coordination. In particular, unmanaged systems should not be responsible for

ensuring atomicity, consistency, isolation, and durability (ACID) properties across failures for transactions involving server resources. BEA Tuxedo CORBA remote clients are lightweight clients.

The Transaction Service allows lightweight clients to do a delegated commit, which means that the Transaction Service allows lightweight clients to begin and terminate transactions while the responsibility for transaction coordination is delegated to a transaction manager running on a server machine. Client applications do not require a local transaction server. The remote TransactionCurrent implementation that CORBA clients use delegates the actual responsibility of transaction coordination to transaction manager on the server.

Support for Third-Party Clients Using INS

In BEA Tuxedo release 8.0 and later, the CORBA Interoperable Naming Service (INS) is supported. Therefore, clients that implement the CORBA services Object Transaction Service (OTS) can communicate with BEA Tuxedo CORBA servers and initiate and terminate transactions. Using INS, any third-party client ORB that can compile the standard OTS IDL files and produce usable stub files can interact with the BEA Tuxedo CORBA transaction manager. However, such interaction is limited because the transaction coordination interfaces that would allow a third-party ORB to become a resource manager are not supported. Only BEA provided resource managers and/or XA compliant resource managers can participate in the coordination of a transaction. Further, the BEA provided and XA compliant resource managers can participate in transaction coordination only if they use the XA protocols—not the CORBA services OTS protocols—for transaction coordination.

In summary, a third-party client ORB can be used to initiate a transaction, and the client can request the rollback or commit of the transaction, however, the client ORB cannot participate in the coordination of the two-phase commit protocol using the CORBA services OTS.

Multithreaded Transaction Client Support

In release 8.0, BEA Tuxedo CORBA supports multithreaded clients for nontransactional clients and transactional clients.

Transaction Propagation (CORBA Only)

For CORBA applications, the OMG CORBA Services *Transaction Service specification* states that a client can choose to propagate a transaction context either implicitly or explicitly. BEA Tuxedo *provides* implicit propagation. Explicit propagation *is strongly discouraged*.

Objects that are related to transaction contexts that are passed around using explicit transaction propagation *should not* be mixed with implicit transaction propagation APIs. It should be noted, however, that explicit propagation does not place any constraints on when transactional methods can be processed. There is no guarantee that all transactional methods will be completed before the transaction is committed.

Transaction Integrity

Checked transaction behavior provides transaction integrity by guaranteeing that a `commit` will not succeed unless all transactional objects involved in the transaction have completed the processing of their transactional requests. If implicit transaction propagation is used, the Transaction Service *provides* checked transaction behavior that is equivalent to that provided by the request/response interprocess communication models defined by The Open Group. For CORBA applications, for example, the Transaction Service performs `reply` checks, `commit` checks, and `resume` checks, as described in the OMG CORBA Services *Transaction Service Specification.*

Unchecked transaction behavior relies completely on the application to provide transaction integrity. If explicit propagation is used, the Transaction Service *does not* provide checked transaction behavior and transaction integrity *is not* guaranteed.

Transaction Termination

BEA Tuxedo CORBA allows transactions to be terminated *only* by the client that created the transaction.

Note: The client may be a server object that requests the services of another object.

Flat Transactions

BEA Tuxedo CORBA implements the flat transaction model. Nested transactions are *not* supported.

Interoperability Between CORBA Remote Clients and the BEA Tuxedo Domain

BEA Tuxedo CORBA supports remote clients invoking methods on server objects in *different* BEA Tuxedo domains in the *same* transaction.

Remote CORBA clients with multiple connections to the same BEA Tuxedo domain *may* make invocations to server objects on these separate connections within the same transaction.

Intradomain and Interdomain Interoperability

BEA Tuxedo CORBA supports native clients invoking methods on server objects in the BEA Tuxedo domain. In addition, BEA Tuxedo supports server objects invoking methods on other objects in the same or in different processes within the same BEA Tuxedo domain.

In BEA Tuxedo applications, transactions can span multiple domains as long as factory-based routing is properly configured across multiple domains. To support transactions across multiple domains, you must configure the `factory_finder.ini` file to identify factory objects that are used in the current (local) domain but that are resident in a different (remote) domain. For more information, see *Using the BEA Tuxedo Domains Component*.

Network Interoperability

A client application can have only one active Bootstrap object and TransactionCurrent object within a single domain. BEA Tuxedo CORBA does *not* support exporting or importing transactions to or from remote BEA Tuxedo domains.

However, transactions can encompass multiple domains in a serial fashion. For example, a server with a transaction active in Domain A can communicate with a server in Domain B within the context of that same transaction.

Relationship of the Transaction Service to Transaction Processing

The Transaction Service relates to various transaction processing servers, interfaces, protocols, and standards in the following ways:

- Support for BEA Tuxedo ATMI servers. Servers using the BEA Tuxedo CORBA Transaction Service can make invocations on other BEA Tuxedo Application-to-Transaction Monitor Interface (ATMI) server processes in the same domain. In addition, ATMI services can invoke CORBA objects in both transactional and nontransactional contexts, both within the same domain and across domains via a BEA Tuxedo Domains gateway. However, BEA Tuxedo CORBA *does not* support remote clients or native clients invoking ATMI services in the BEA Tuxedo domain.
- Support for The Open Group XA interface. The Open Group resource managers are resource managers that can be involved in a distributed transaction by allowing their

two-phase commit protocol to be controlled via The Open Group XA interface. BEA Tuxedo supports interaction with The Open Group resource managers.

- Support for the OSI TP protocol. Open Systems Interconnect Transaction Processing (OSI TP) is the transactional protocol defined by the International Organization for Standardization (ISO). BEA Tuxedo CORBA *does not* support interactions with OSI TP transactions.
- Support for the LU 6.2 protocol. Systems Network Architecture (SNA) LU 6.2 is a transactional protocol defined by IBM. BEA Tuxedo CORBA *does not* support interactions with LU 6.2 transactions.
- Support for the ODMG standard. ODMG-93 is a standard defined by the Object Database Management Group (ODMG) that describes a portable interface to access Object Database Management Systems. BEA Tuxedo CORBA *does not* support interactions with ODMG transactions.

Process Failure

The Transaction Service monitors the participants in a transaction for failures and inactivity. The BEA Tuxedo system provides management tools for keeping the application running when failures occur. Because BEA Tuxedo CORBA is built upon the BEA Tuxedo transaction management system, it inherits the BEA Tuxedo capabilities for keeping applications running.

General Constraints

The following constraints apply to the Transaction Service:

- In BEA Tuxedo CORBA, a client or a server object *cannot* invoke methods on an object that is infected with (or participating in) another transaction. The method invocation issued by the client or the server will return an exception.
- For CORBA applications, a server application object using transactions from the BEA Tuxedo CORBA Transaction Service library *requires* the TP Framework functionality. For more information about the TP Framework, see "TP Framework" in the *CORBA Programming Reference*.
- For CORBA applications, a return from the `rollback` method on the `Current` object is asynchronous.

  As a result, the objects that were infected by (or participating in) the rolled back transaction get their states cleared by BEA Tuxedo *a little later.* Therefore, *no* other client can infect these objects with a different transaction until BEA Tuxedo clears the states of

these objects. This condition exists for a very short amount of time and is typically not noticeable in a production application. A simple workaround for this race condition is to try the appropriate operation after a short (typically a 1-second) delay.

- In BEA Tuxedo CORBA applications, clients may not make one-way method invocations within the context of a transaction to server objects having the `NEVER`, `OPTIONAL`, or `ALWAYS` transaction policies.

  No error or exception will be returned to the client because it is a one-way method invocation. However, the method on the server object will not be executed, and an appropriate error message will be written to the log. Clients may make one-way method invocations within the context of a transaction to server objects with the `IGNORE` transaction policy. In this case, the method on the server object will be executed, but not in the context of a transaction. For more information about the transaction policies, see "Implementation Configuration File (ICF)" in the *CORBA Programming Reference*.

Transaction Service in CORBA Applications

This topic includes the following sections:

- Getting Initial References to the TransactionCurrent Object Using the Bootstrap Object
- Getting Initial References to the TransactionFactory Object Using INS
- CORBA Transaction Service API
- CORBA Transaction Service API Extensions
- Notes on Using Transactions in BEA Tuxedo CORBA Applications

These sections describe how BEA Tuxedo implements the OTS, with particular emphasis on the portion of the CORBAservices Object Transaction Service that is described as implementation-specific. They describe the OTS application programming interface (API) that you use to begin or terminate transactions, suspend or resume transactions, and get information about transactions.

Getting Initial References to the TransactionCurrent Object Using the Bootstrap Object

To use the TransactionCurrent object to access the Transaction Service API and the extension to the Transaction Service API as described later in this chapter, an application needs to complete the following operations:

1. Create a Bootstrap object. For more information about creating a Bootstrap object, see the "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.
2. Invoke the `resolve_initial_reference("TransactionCurrent")` method on the Bootstrap object. The invocation returns a standard CORBA object pointer. For a description of this Bootstrap object method, see the *CORBA Programming Reference*.
3. If an application requires only the Transaction Service APIs, it should issue a `CosTransactionsCurrent::_narrow()` (in C++) on the object pointer returned from step 2 above.

   If an application requires the Transaction Service APIs with the extensions, it should issue a `Tobj::TransactionCurrent::_narrow()` (in C++) on the object pointer returned from step 2 above.

Getting Initial References to the TransactionFactory Object Using INS

BEA Tuxedo also supports the use of the CORBA Interoperable Naming Service (INS) by third-party clients to obtain initial transaction object references. INS uses the `ORB::resolve_initial_references()` operation.

Listing 2-1 shows an example of how a client application, using INS, gets an object reference to the TransactionFactory object. For a complete code example, see the client application in the University Sample.

Listing 2-1 Code Example for a Client Application that Uses INS

```
// Get the factory finder from the ORB:
CORBA::Object_var v_fact_finder_oref =
   orb->resolve_initial_references("FactoryFinder");

// Narrow the factory finder :
Tobj::FactoryFinder_var v_fact_finder_ref =
   Tobj::FactoryFinder::_narrow(v_fact_finder_oref.in());

// Get the TransactionFactory from the FactoryFinder
CORBA::Object_var v_txn_fac_oref =
   v_fact_finder_ref->find_one_factory_by_id(
             "IDL:omg.org/CosTransactions/TransactionFactory:1.0");
```

```
// Narrow the TransactionFactory object reference
CosTransactions::TransactionFactory_var v_txn_fac_ref =
   CosTransactions::TransactionFactory::_narrow(
                                        v_txn_fac_oref.in());
```

For more information about using the `ORB::resolve_initial_references()` operation, see "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

CORBA Transaction Service API

This topic includes the following sections:

- Data Types
- Exceptions
- Current Interface
- Control Interface
- TransactionalObject Interface

These sections describe the CORBA-based components of the `CosTransactions` modules that BEA Tuxedo implements to support the Transaction Service. For more information about these components, see the OMG CORBA Services *Transaction Service Specification,* Version 1.1, May 2000.

Data Types

Listing 2-2 shows the supported data types.

Listing 2-2 Data Types Supported by the Transaction Service

```
enum Status {
       StatusActive,
       StatusMarkedRollback,
       StatusPrepared,
       StatusCommitted,
       StatusRolledBack,
```

```
        StatusUnknown,
        StatusNoTransaction,
        StatusPreparing,
        StatusCommitting,
        StatusRollingBack
};
// This information comes from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

Exceptions

Listing 2-3 shows the supported exceptions in IDL code.

Listing 2-3 Exceptions Supported by the Transaction Service

```
// Heuristic exceptions
exception HeuristicMixed {};
exception HeuristicHazard {};

// Other transaction-specific exceptions
exception SubtransactionsUnavailable {};
exception NoTransaction {};
exception InvalidControl {};
exception Unavailable {};
```

Table 2-1 describes the exceptions.

Note: This information comes from the OMG CORBA Services *Transaction Service Specification*, Version 1.1, May 2000. Used with permission of the OMG.

Table 2-1 Exceptions Supported by the Transaction Service

Exception	Description
`HeuristicMixed`	A request raises this exception to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back.
`HeuristicHazard`	A request raises this exception to report that a heuristic decision was made, that the disposition of all relevant updates is not known, and that for those updates whose disposition is known, either all have been committed or all have been rolled back. Therefore, the `HeuristicMixed` exception takes priority over the `HeuristicHazard` exception.
`SubtransactionsUnavailable`	This exception is raised for the `Current` interface `begin` method if the client already has an associated transaction.
`NoTransaction`	This exception is raised for the `Current` interface `rollback` and `rollback_only` methods if there is no transaction associated with the client application.
`InvalidControl`	This exception is raised for the `Current` interface `resume` method if the parameter is not valid in the current execution environment.
`Unavailable`	This exception is raised for the `Control` interface `get_terminator` and `get_coordinator` methods if the `Control` interface cannot provide the requested object.

Current Interface

The `Current` interface defines methods that allow a client of the Transaction Service to explicitly manage the association between threads and transactions. The `Current` interface also defines methods that simplify the use of the Transaction Service for most applications. These methods can be used to begin and end transactions, to suspend and resume transactions, and to obtain information about the current transaction.

The `CosTransactions` module defines the `Current` interface (shown in Listing 2-4).

Listing 2-4 Current Interface IDL

```
// Current transaction
interface Current : CORBA::Current {
       void begin()
              raises(SubtransactionsUnavailable);
       void commit(in boolean report_heuristics)
              raises(
                     NoTransaction,
                     HeuristicMixed,
                     HeuristicHazard
             );
       void rollback()
              raises(NoTransaction);
       void rollback_only()
              raises(NoTransaction);
       Status get_status();
       string get_transaction_name();
       void set_timeout(in unsigned long seconds);
       Control get_control();
       Control suspend();
       void resume(in Control which)
              raises(InvalidControl);

};

// This information comes from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

Table 2-2 provides a description of the `Current` transaction methods.

Note: This information was taken from the OMG CORBA Services *Transaction Service Specification*, Version 1.1, May 2000. Used with permission of the OMG.

Table 2-2 Transaction Methods in the Current Object

Method	Description
`begin`	Creates a new transaction. The transaction context of the client application is modified so that the thread is associated with the new transaction. If the client application is currently associated with a transaction, the `SubtransactionsUnavailable` exception is raised. If the client application cannot be placed in transaction mode due to an error while starting the transaction, the standard system exception `INVALID_TRANSACTION` is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised.
`commit`	If there is no transaction associated with the client application, the `NoTransaction` exception is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the system decides to roll back the transaction, the standard exception `TRANSACTION_ROLLEDBACK` is raised and the thread's transaction context is set to NULL. A `HeuristicMixed` exception is raised to report that a heuristic decision was made and that some relevant updates have been committed and others have been rolled back. A `HeuristicHazard` exception is raised to report that a heuristic decision was made, and that the disposition of all relevant updates is not known; for those updates whose disposition is known, either all have been committed or all have been rolled back. The `HeuristicMixed` exception takes priority over the `HeuristicHazard` exception. If a heuristic exception is raised or the operation completes normally, the thread's transaction exception context is set to NULL. If the operation completes normally, the thread's transaction context is set to NULL.

Table 2-2 Transaction Methods in the Current Object (Continued)

Method	Description
`rollback`	If there is no transaction associated with the client application, the `NoTransaction` exception is raised. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the operation completes normally, the thread's transaction context is set to NULL.
`rollback_only`	If there is no transaction associated with the client application, the `NoTransaction` exception is raised. Otherwise, the transaction associated with the client application is modified so that the only possible outcome is to roll back the transaction.
`get_status`	If there is no transaction associated with the client application, the `StatusNoTransaction` value is returned. Otherwise, this method returns the status of the transaction associated with the client application.
`get_transaction_name`	If there is no transaction associated with the client application, an empty string is returned. Otherwise, this method returns a printable string describing the transaction (specifically, the `XID` as specified by The Open Group). The returned string is intended to support debugging.

Table 2-2 Transaction Methods in the Current Object (Continued)

Method	Description
`set_timeout`	This method modifies a state variable associated with the target object that affects the timeout period associated with transactions created by subsequent invocations of the `begin` method. The initial transaction timeout value is 300 seconds. Calling `set_timeout()` with an argument value larger than zero specifies a new timeout value. Calling `set_timeout()` with a zero argument sets the timeout value back to the default of 300 seconds. After calling `set_timeout()`, transactions created by subsequent invocations of `begin` are subject to being rolled back if they do not complete before the specified number of seconds after their creation. **Note:** The initial transaction timeout value is 300 seconds. If a transaction is started via `AUTOTRAN` instead of the `begin` method, then the timeout value is determined by the `TRANTIME` value in the BEA Tuxedo configuration file. For more information, see Chapter 5, "Administering Transactions."
`get_control`	If the client is not associated with a transaction, a NULL object reference is returned. Otherwise, a `Control` object is returned that represents the transaction context currently associated with the client application. This object may be given to the `resume` method to reestablish this context.

Table 2-2 Transaction Methods in the Current Object (Continued)

Method	Description
`suspend`	If the client application is not associated with a transaction, a NULL object reference is returned. If the associated transaction is in a state such that the only possible outcome of the transaction is to be rolled back, the standard system exception `TRANSACTION_ROLLEDBACK` is raised and the client application becomes associated with no transaction. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. The caller's state with respect to the transaction is not changed. Otherwise, an object is returned that represents the transaction context currently associated with the client application. The same client can subsequently give this object to the `resume` method to reestablish this context. In addition, the client application becomes associated with no transaction. **Note:** As defined in The Common Object Request Broker: Architecture and Specification, Revision 2.4, the standard system exception `TRANSACTION_ROLLEDBACK` indicates that the transaction associated with the request has already been rolled back or has been marked to roll back. Thus, the requested method either could not be performed or was not performed because further computation on behalf of the transaction would be fruitless.

Table 2-2 Transaction Methods in the Current Object (Continued)

Method	Description
`resume`	If the client application is already associated with a transaction which is in a state such that the only possible outcome of the transaction is to be rolled back, the standard system exception `TRANSACTION_ROLLEDBACK` is raised and the client application becomes associated with no transaction. If the call was made in an improper context, the standard system exception `BAD_INV_ORDER` is raised. If the system is unable to resume the global transaction because the caller is currently participating in work outside any global transaction with one or more resource managers, the standard system exception `INVALID_TRANSACTION` is raised. If the parameter is a NULL object reference, the client application becomes associated with no transaction. If the parameter is valid in the current execution environment, the client application becomes associated with that transaction (in place of any previous transaction). Otherwise, the `InvalidControl` exception is raised. **Note:** See `suspend` for a definition of the standard system exception `TRANSACTION_ROLLEDBACK`.

Control Interface

The `Control` interface allows a program to explicitly manage or propagate a transaction context. An object that supports the `Control` interface is implicitly associated with one specific transaction.

Listing 2-5 shows the `Control` interface, which is defined in the `CosTransactions` module.

Listing 2-5 Control Interface

```
interface Control {
       Terminator get_terminator()
              raises(Unavailable);
       Coordinator get_coordinator()
```

```
            raises(Unavailable);
};

// This information comes from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

The `Control` interface is used only in conjunction with the `suspend` and `resume` methods.

Terminator Interface

The Terminator interface supports operations to commit or roll back a transaction. Typically, these operations are used by the transaction originator. An implementation of the Transaction Service may restrict the scope in which a Terminator can be used; at a minimum, it can be used within a single thread.

Listing 2-6 shows the Terminator interface.

Listing 2-6 Terminator Interface

```
interface Terminator {
     void commit(in boolean report_heuristics)
         raises(
             HeuristicMixed,
             HeuristicHazard
);
    void rollback();
};

// This information was taken from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

Table 2-3 describes the Terminator interface methods.

Table 2-3 Termination Interface Methods

Method	Description
`commit`	If the transaction has not been marked rollback only, and all of the participants in the transaction agree to commit, the transaction is committed and the operation terminates normally. Otherwise, the transaction is rolled back (as described below for the `rollback` method) and the `TRANSACTION_ROLLEDBACK` standard exception is raised. If the `report_heuristics` parameter is true, the Transaction Service will report inconsistent or possibly inconsistent outcomes using the `HeuristicMixed` and `HeuristicHazard` exceptions. A Transaction Service implementation may optionally use the CORBA Notification Service to report heuristic decisions. The `commit` operation may roll back the transaction if there are subtransactions of the transaction that have not themselves been committed or rolled back or if there are existing or potential activities associated with the transaction that have not completed. The nature and extent of such error checking is implementation-dependent. When a top-level transaction is committed, all changes to recoverable objects made in the scope of this transaction are made permanent and visible to other transactions or clients. When a subtransaction is committed, the changes are made visible to other related transactions as appropriate to the degree of isolation enforced by the resources.
`rollback`	The transaction is rolled back. When a transaction is rolled back, all changes to recoverable objects made in the scope of this transaction (including changes made by descendant transactions) are rolled back. All resources locked by the transaction are made available to other transactions as appropriate to the degree of isolation enforced by the resources.

TransactionalObject Interface

In BEA Tuxedo release 8.0 and later, the `CosTransactions::TransactionalObject` is no longer used by an object to indicate that it is transactional. If an interface inherits from a TransactionalObject and the ICF indicates a different transaction policy, a warning is issued. The TransactionalObject is not used for any other purpose. For details on transaction policies that need to be set to infect objects with transactions, see “Implementation Configuration File (ICF)” in the *CORBA Programming Reference*.

The CosTransactions module defines the TransactionalObject interface (shown in Listing 2-7). This interface defines no methods; it is simply a marker.

Listing 2-7 TransactionalObject Interface

```
interface TransactionalObject {
};

// This information was taken from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

TransactionFactory Interface

The TransactionFactory interface is provided to allow the transaction originator to begin a transaction. This interface defines two operations, create and recreate, which create a new representation of a top-level transaction. A TransactionFactory is located using the FactoryFinder interface of the life cycle service and not by the resolve_initial_reference() operation on the ORB interface.

Listing 2-8 shows the TransactionFactory interface.

Note: The Control recreate method of the TransactionFactory interface is not supported.

Listing 2-8 TransactionFactory Interface

```
interface TransactionFactory {
     Control create(in unsigned long time_out);
     Control recreate(in PropagationContext ctx);
};

// This information was taken from the OMG Transaction Service
// Specification, Version 1.1, May 2000. Used with permission
// of the OMG.
```

Table 2-4 describes the `TransactionFactory` interface methods.

Table 2-4 TransactionFactory Interface Methods

Method	Description
create	A new top-level transaction is created and a `Control` object is returned. The `Control` object can be used to manage or to control participation in the new transaction. An implementation of the Transaction Service may restrict the ability for the `Control` object to be transmitted to or used in other execution environments; at a minimum, it can be used by the client application. If the parameter has a nonzero value `n`, then the new transaction will be subject to being rolled back if it does not complete before `n` seconds have elapsed. If the parameter is zero, then no application specified timeout is established.
recreate	Not supported.

Other CORBAservices Object Transaction Service Interfaces

All other CORBAservices Object Transaction Service interfaces are *not* supported. Note that the `Current` interface described earlier is supported only if it has been obtained from the Bootstrap object. The `Control` interface described earlier is supported only if it has been obtained using the `get_control` and the `suspend` methods on the Current object.

CORBA Transaction Service API Extensions

This topic describes specific extensions to the CORBAservices Transaction Service API described earlier. The APIs in this topic enable an application to open or close an Open Group resource manager.

The following APIs help facilitate participation of resource managers in a distributed transaction by allowing their two-phase commit protocol to be controlled via The Open Group XA interface.

The following definitions and interfaces are defined in the `Tobj` module.

Exception

The following exception is supported:

```
exception RMfailed {};
```

A request raises this exception to report that an attempt to open or close a resource manager failed.

TransactionCurrent Interface

This interface supports all the methods of the `Current` interface in the `CosTransactions` module and is described in "C++ Bootstrap Object Programming Reference" in the *CORBA Programming Reference*. Additionally, this interface supports APIs to open and close the resource manager.

Listing 2-9 shows the `TransactionCurrent` interface, which is defined in the `Tobj` module.

Listing 2-9 TransactionCurrent Interface

```
Interface TransactionCurrent: CosTransactions::Current {
      void open_xa_rm()
             raises(RMfailed);
      void close_xa_rm()
             raises(Rmfailed);
}
```

Table 2-5 describes APIs that are specific to the resource manager. For more information about these APIs, see the *CORBA Programming Reference*.

Table 2-5 Resource Manager APIs for the Current Interface

Method	Description
`open_xa_rm`	This method opens The Open Group resource manager to which this process is linked. A `RMfailed` exception is raised if there is a failure while opening the resource manager. Any attempts to invoke this method by remote clients or the native clients raises the standard system exception `NO_IMPLEMENT`.
`close_xa_rm`	This method closes The Open Group resource manager to which this process is linked. An `RMfailed` exception is raised if there is a failure while closing the resource manager. A `BAD_INV_ORDER` standard system exception is raised if the function was called in an improper context (for example, the caller is in transaction mode). Any attempts by the remote clients or the native clients to invoke this method raises the standard system exception `NO_IMPLEMENT`.

Notes on Using Transactions in BEA Tuxedo CORBA Applications

Consider the following guidelines when integrating transactions into your BEA Tuxedo CORBA client/server applications:

- Nested transactions are not permitted in the BEA Tuxedo system. You cannot start a new transaction if an existing transaction is already active. (You may start a new transaction if you first suspend the existing one; however, the object that suspends the transaction is the only object that can subsequently resume the transaction.)
- The object that starts a transaction is the only entity that can end the transaction. (In a strict sense, the object can be the client application, the TP Framework, or an object managed by the server application.) An object that is invoked within the scope of a transaction may suspend and resume the transaction (and while the transaction is suspended, the object can start and end other transactions). However, you cannot end a transaction in an object unless you began the transaction there.
- BEA Tuxedo does not support concurrent transactions. Objects can be involved with only one transaction at one time. An object is involved in a transaction for the duration of the entire transaction, and is available to be involved in a different transaction only after the current transaction is completed.

- BEA Tuxedo does not queue requests to objects that are currently involved in a transaction. If a nontransactional client application attempts to invoke an operation on an object that is currently in a transaction, the client application receives the following error message:

  C++

  ```
  CORBA::OBJ_ADAPTER
  ```

  If a client that is in a transaction attempts to invoke an operation on an object that is currently in a different transaction, the client application receives the following error message:

  C++

  ```
  CORBA::INVALID_TRANSACTION
  ```

- For transaction-bound objects, consider doing all state handling in the `Tobj_ServantBase::deactivate_object()` operation. This makes it easier for the object to handle its state properly, because the outcome of the transaction is known at the time that `deactivate_object()` is invoked.

- For method-bound objects that have several operations, but only a few that affect the object's durable state, consider doing the following:

  – Assign the `optional` transaction policy.

  – Scope any write operations within a transaction, by making invocations on the TransactionCurrent object.

  If the object is invoked outside a transaction, the object does not incur the overhead of scoping a transaction for reading data. This way, regardless of whether the object is invoked within a transaction, all the object's write operations are handled transactionally.

- Transaction rollbacks are asynchronous. Therefore, it is possible for an object to be invoked while its transactional context is still active. If you try to invoke such an object, you receive an exception.

- If an object with the `always` transaction policy is involved in a transaction that is started by the BEA Tuxedo system, and not the client application, note the following:

  – If the server application marks the transaction for rollback only and the server throws a CORBA exception, the client application receives the CORBA exception.

  – If the server application marks the transaction for rollback only and the server does *not* throw a CORBA exception, the client application receives the `OBJ_ADAPTER` exception. In this case, the BEA Tuxedo system automatically rolls back the transaction. However,

the client application is completely unaware that a transaction has been scoped in the BEA Tuxedo domain.

- If the client application initiates a transaction, and the server application marks the transaction for a rollback, one of the following occurs:
  - If the server throws a CORBA exception, the client application receives a CORBA exception.
  - If the server does *not* throw a CORBA exception, the client application receives the `TRANSACTION_ROLLEDBACK` exception.

UserTransaction API

This topic includes the following sections:

UserTransaction Methods

Table 2-6 describes the methods in the UserTransaction object.

Table 2-6 Methods in the UserTransaction Object

Method Name	Description
`begin`	Starts a transaction on the current thread.
`commit`	Commits the transaction associated with the current thread.

Table 2-6 Methods in the UserTransaction Object (Continued)

Method Name	Description
`getStatus`	Returns the transaction status, or `STATUS_NO_TRANSACTION` if no transaction is associated with the current thread. One of the following values: • `STATUS_ACTIVE` • `STATUS_COMMITTED` • `STATUS_COMMITTING` • `STATUS_MARKED_ROLLBACK` • `STATUS_NO_TRANSACTION` • `STATUS_PREPARED` • `STATUS_PREPARING` • `STATUS_ROLLEDBACK` • `STATUS_ROLLING_BACK` • `STATUS_UNKNOWN`
`rollback`	Rolls back the transaction associated with the current thread.
`setRollbackOnly`	Marks the transaction associated with the current thread so that the only possible outcome of the transaction is to roll it back.
`setTransactionTimeout`	Specifies the timeout value for the transactions started by the current thread with the `begin` method. If an application has not called the `begin` method, then the Transaction Service uses a default value for the transaction timeout.

Exceptions Thrown by UserTransaction Methods

Table 2-7 describes exceptions thrown by methods of the UserTransaction object.

Table 2-7 Exceptions Thrown by UserTransaction Methods

Exception	Description
`HeuristicMixedException`	Thrown to indicate that a heuristic decision was made and that some relevant updates have been committed while others have been rolled back.
`HeuristicRollbackException`	Thrown to indicate that a heuristic decision was made and that some relevant updates have been rolled back.
`NotSupportedException`	Thrown when the requested operation is not supported (such as a nested transaction).
`RollbackException`	Thrown when the transaction has been marked for rollback only or the transaction has been rolled back instead of committed.
`IllegalStateException`	Thrown if the current thread is not associated with a transaction.
`SecurityException`	Thrown to indicate that the thread is not allowed to commit the transaction.
`SystemException`	Thrown by the transaction manager to indicate that it has encountered an unexpected error condition that prevents future transaction services from proceeding.

CHAPTER 3

Transactions in CORBA Server Applications

This topic includes the following sections:

- Integrating Transactions in a BEA Tuxedo Client and Server Application
- Transactions and Object State Management
- User-defined Exceptions

These sections describe how to integrate transactions into a BEA Tuxedo server application. Before you begin, you should read Chapter 1, "Introducing Transactions."

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Integrating Transactions in a BEA Tuxedo Client and Server Application

This topic includes the following sections:

Transaction Support in CORBA Applications

BEA Tuxedo supports transactions in the following ways:

- The client or the server application can begin and end transactions explicitly by using calls on the TransactionCurrent object. For details about the TransactionCurrent object, see Chapter 4, "Transactions in CORBA Client Applications."
- You can assign transactional policies to an object's interface so that when the object is invoked, the BEA Tuxedo system can start a transaction automatically for that object, if a transaction has not already been started, and commit or roll back the transaction when the method invocation is complete. You use transactional policies on objects in conjunction with an XA resource manager and database when you want to delegate all the transaction commit and rollback responsibilities to that resource manager.
- Objects involved in a transaction can force a transaction to be rolled back. That is, after an object has been invoked within the scope of a transaction, the object can invoke `rollback_only()` on the TransactionCurrent object to mark the transaction for rollback only. This prevents the current transaction from being committed. An object may need to mark a transaction for rollback if an entity, typically a database, is otherwise at risk of being updated with corrupt or inaccurate data.

- Objects involved in a transaction can be kept in memory from the time they are first invoked until the moment when the transaction is ready to be committed or rolled back. In the case of a transaction that is about to be committed, these objects are polled by the BEA Tuxedo system immediately before the resource managers prepare to commit the transaction. In this sense, polling means invoking the object's `Tobj_ServantBase::deactivate_object()` operation and passing a reason value.

  When an object is polled, the object may veto the current transaction by invoking `rollback_only()` on the TransactionCurrent object. In addition, if the current transaction is to be rolled back, objects have an opportunity to skip any writes to a database. If no object vetoes the current transaction, the transaction is committed.

The following sections explain how you can use object activation policies and transaction policies to determine the transactional behavior you want in your objects. Note that these policies apply to an interface and, therefore, to all operations on all objects implementing that interface.

Note: If a server application manages an object that you want to be able to participate in a transaction, the Server object for that application must invoke the `TP::open_xa_rm()` and `TP::close_xa_rm()` operations. For more information about database connections, see "Opening an XA Resource Manager" on page 3-7.

Making an Object Automatically Transactional

The BEA Tuxedo system provides the `always` transactional policy, which you can define on an object's interface to have the BEA Tuxedo system start a transaction automatically when that object is invoked and a transaction has not already been scoped. When an invocation on that object is completed, the BEA Tuxedo system commits or rolls back the transaction automatically. Neither the server application, nor the object implementation, needs to invoke the TransactionCurrent object in this situation; the BEA Tuxedo system automatically invokes the TransactionCurrent object on behalf of the server application.

Assign the `always` transactional policy to an object's interface when:

- The object writes to a database and you want all the database commit or rollback responsibilities delegated to an XA resource manager whenever this object is invoked.
- You want to give the client application the opportunity to include the object in a larger transaction that encompasses invocations on multiple objects, and the invocations must all succeed or be rolled back if any one invocation fails.

If you want an object to be automatically transactional, assign the following policies to that object's interface in the Implementation Configuration File:

Activation Policies	Transaction Policy
• process • method • transaction	always

Note: Database cursors cannot span transactions. However, in C++, the CourseSynopsisEnumerator object in the BEA Tuxedo University sample applications uses a database cursor to find matching course synopses from the University database. Because database cursors cannot span transactions, the `activate_object()` operation on the CourseSynopsisEnumerator object reads all matching course synopses into memory. Note that the cursor is managed by an iterator class and is thus not visible to the CourseSynopsisEnumerator object.

Enabling an Object to Participate in a Transaction

If you want an object to be able to be invoked within the scope of a transaction, you can assign the `optional` transaction policies to that object's interface. The `optional` transaction policy may be appropriate for an object that does not perform any database write operations, but that you want to have the ability to be invoked during a transaction.

You can use the following policies, when they are specified in the Implementation Configuration File for that object's interface, to make an object optionally transactional:

Activation Policies	Transaction Policy
• process • method • transaction	optional

When the transaction policy is `optional`, if the `AUTOTRAN` parameter is enabled in the application's `UBBCONFIG` file, the implementation is transactional. Servers containing transactional objects must be configured within a group associated with an XA-compliant resource manager.

If the object does perform database write operations, and you want the object to be able to participate in a transaction, assigning the `always` transactional policy is generally a better choice. However, if you prefer, you can use the `optional` policy and encapsulate any write operations

within invocations on the TransactionCurrent object. That is, within your operations that write data, scope a transaction around the write statements by invoking the TransactionCurrent object to, respectively, begin and commit or roll back the transaction, if the object is not already scoped within a transaction. This ensures that any database write operations are handled transactionally. This also introduces a performance efficiency: if the object is not invoked within the scope of a transaction, all the database read operations are nontransactional, and, therefore, more streamlined.

Note: When choosing the transaction policies to assign to your objects, make sure you are familiar with the requirements of the XA resource manager you are using. For example, some XA resource managers (such as the Oracle 7 Transaction Manager Server) require that any object participating in a transaction scope their database read operations, in addition to write operations, within a transaction (you can still scope your own transactions, however). Other resource managers, such as Oracle8i, do not require a transaction context for read and write operations. If an application attempts a write operation without a transaction context, Oracle8i will start a local transaction implicitly, in which case the application needs to commit the local transaction explicitly.

Preventing an Object from Being Invoked While a Transaction Is Scoped

In many cases, it may be critical to exclude an object from a transaction. If such an object is invoked during a transaction, the object returns an exception, which may cause the transaction to be rolled back. BEA Tuxedo CORBA provides the `never` transaction policy, which you can assign to an object's interface to specifically prevent that object from being invoked within the course of a transaction, even if the current transaction is suspended.

This transaction policy is appropriate for objects that write durable state to disk that cannot be rolled back, such as for an object that writes data to a disk that is not managed by an XA resource manager. Having this capability in your client/server application is crucial if the client application does not or cannot know if some of its invocations are causing a transaction to be scoped. Therefore, if a transaction is scoped, and an object with this policy is invoked, the transaction can be rolled back.

To prevent an object from being invoked while a transaction is scoped, assign the following policies to that object's interface in the Implementation Configuration File:

Activation Policies	Transaction Policy
• process • method	never

Excluding an Object from an Ongoing Transaction

In some cases, it may be appropriate to permit an object to be invoked during the course of a transaction but also keep that object from being a part of the transaction. If such an object is invoked during a transaction, the transaction is automatically suspended. After the invocation on the object is completed, the transaction is automatically resumed. BEA Tuxedo CORBA provides the `ignore` transaction policy for this purpose.

The `ignore` transaction policy may be appropriate for an object such as a factory that typically does not write data to disk. By excluding the factory from the transaction, the factory can be available to other client invocations during the course of a transaction. In addition, using this policy can introduce an efficiency into your server application because it minimizes the overhead of invoking objects transactionally.

To prevent any transaction from being propagated to an object, assign the following policies to that object's interface in the Implementation Configuration File:

Activation Policies	Transaction Policy
• process • method	ignore

Assigning Policies

For information about how to create an Implementation Configuration File and specify policies on objects, see "Step 4: Define the in-memory behavior of objects" in "Steps for Creating a BEA Tuxedo CORBA Server Application" in the *CORBA Programming Reference*.

Using an XA Resource Manager

The Transaction Manager Server (TMS) handles object state data automatically. For an example, the University sample C++ application in the

drive:\TUX8\samples\corba\university\transactions directory uses the Oracle TMS as an example of a relational database management service (RDBMS).

Using any XA resource manager imposes specific requirements on how different objects managed by the server application may read and write data to that database, including the following:

- Some XA resource managers, such as Oracle7, require that all database operations be scoped within a transaction. This means that all method invocations on the DBaccess object need to be scoped within a transaction because this object reads from a database. The transaction can be started either by the client or by the BEA Tuxedo system.

  Other XA resource managers, such as Oracle8i, do not require a transaction context for read and write operations. If an application attempts a write operation without a transaction context, Oracle8i will start a local transaction implicitly, in which case the application needs to commit the local transaction explicitly.

- When a transaction is committed or rolled back, the XA resource manager automatically handles the durable state implied by the commit or rollback. That is, if the transaction is committed, the XA resource manager ensures that all database updates are made permanent. Likewise, if there is a rollback, the XA resource manager automatically restores the database to its state prior to the beginning of the transaction.

  This characteristic of XA resource managers actually makes the design problems associated with handling object state data in the event of a rollback much simpler. Transactional objects can always delegate the commit and rollback responsibilities to the XA resource manager, which greatly simplifies the task of implementing a server application.

Opening an XA Resource Manager

If an object's interface has the `always` or `optional` transaction policy, you must invoke the `TP::open_xa_rm()` operation in the `Server::initialize()` operation in the Server object. The resource manager is opened using the information provided in the `OPENINFO` parameter, which is in the `GROUPS` section of the `UBBCONFIG` file. Note that the default version of the `Server::initialize()` operation automatically opens the resource manager.

If you have an object that does not write data to disk and that participates in a transaction—the object typically has the `optional` transaction policy—you still need to include an invocation to the `TP::open_xa_rm()` operation. In that invocation, specify the NULL resource manager.

Closing an XA Resource Manager

If your Server object's `Server::initialize()` operation opens an XA resource manager, you must include the following invocation in the `Server::release()` operation:

```
TP::close_xa_rm();
```

Transactions and Object State Management

This topic includes the following sections:

- Delegating Object State Management to an XA Resource Manager
- Waiting Until Transaction Work Is Complete Before Writing to the Database

If you need transactions in your BEA Tuxedo CORBA client and server application, you can integrate transactions with object state management in a few different ways. In general, BEA Tuxedo CORBA can automatically scope the transaction for the duration of an operation invocation without requiring you to make any changes to your application's logic or the way in which the object writes durable state to disk.

Delegating Object State Management to an XA Resource Manager

Using an XA resource manager, such as Oracle, generally simplifies the design problems associated with handling object state data in the event of a rollback. (The Oracle resource manager is used in the BEA Tuxedo CORBA University sample C++ applications). Transactional objects can always delegate the commit and rollback responsibilities to the XA resource manager, which greatly simplifies the task of implementing a server application. This means that process- or method-bound objects involved in a transaction can write to a database during transactions, and can depend on the resource manager to undo any data written to the database in the event of a transaction rollback.

Waiting Until Transaction Work Is Complete Before Writing to the Database

The `transaction` activation policy is a good choice for objects that maintain state in memory that you do not want written, or that cannot be written, to disk until the transaction work is complete. When you assign the `transaction` activation policy to an object, the object:

- Is brought into memory when it is first invoked within the scope of a transaction.
- Remains in memory until the transaction is either committed or rolled back.

When the transaction work is complete, BEA Tuxedo CORBA invokes each transaction-bound object's `Tobj_ServantBase::deactivate_object()` operation passing a `reason` code that can be either `DR_TRANS_COMMITTING` or `DR_TRANS_ABORTED`. If the variable is `DR_TRANS_COMMITTING`, the object can invoke its database write operations. If the variable is `DR_TRANS_ABORTED`, the object skips its write operations.

When to Assign the Transaction Activation Policy

Assigning the `transaction` activation policy to an object may be appropriate in the following situations:

- You want the object to write its persistent state to disk at the time that the transaction work is complete.

  This introduces a performance efficiency because it reduces the number of database write operations that may need to be rolled back.

- You want to provide the object with the ability to veto a transaction that is about to be committed.

  If BEA Tuxedo CORBA passes the reason `DR_TRANS_COMMITTING`, the object can, if necessary, invoke `rollback_only()` on the TransactionCurrent object. Note that if you do make an invocation to `rollback_only()` from within the `Tobj_ServantBase::deactivate_object()` operation, then `deactivate_object()` is not invoked again.

- You want to provide the object with the ability to perform batch updates.
- You have an object that is likely to be invoked multiple times during the course of a single transaction, and you want to avoid the overhead of continually activating and deactivating the object during that transaction.

Transaction Policies to Use with the Transaction Activation Policy

To give an object the ability to wait until the transaction is committing before writing to a database, assign the following policies to that object's interface in the Implementation Configuration File:

Activation Policy	Transaction Policy
`transaction`	`always` or `optional`

Note: Transaction-bound objects cannot start a transaction or invoke other objects from inside the `Tobj_ServantBase::deactivate_object()` operation. The only valid invocations transaction-bound objects can make inside `deactivate_object()` are write operations to the database.

Also, if you have an object that is involved in a transaction, the Server object that manages that object must include invocations to open and close the XA resource manager, even if the object does not write any data to disk. (If you have a transactional object that does not write data to disk, you specify the NULL resource manager.) For more information about opening and closing an XA resource manager, see "Opening an XA Resource Manager" on page 3-7 and "Closing an XA Resource Manager" on page 3-8.

User-defined Exceptions

This topic includes the following sections:

About User-defined Exceptions

Including a user-defined exception in a BEA Tuxedo CORBA client/server application involves the following steps:

1. In your OMG IDL file, define the exception and specify the operations that can use it.
2. In the implementation file, include code that throws the exception.
3. In the client application source file, include code that catches and handles the exception.

For example, the Transactions sample C++ application includes an instance of a user-defined exception, `TooManyCredits`. This exception is thrown by the server application when the client application tries to register a student for a course, and the student has exceeded the maximum

number of courses for which he or she can register. When the client application catches this exception, the client application rolls back the transaction that registers a student for a course. This section explains how you can define and implement user-defined exceptions in your BEA Tuxedo CORBA client/server application, using the `TooManyCredits` exception as an example.

Defining the Exception

In the OMG IDL file for your client/server application:

1. Define the exception and define the data sent with the exception. For example, the `TooManyCredits` exception is defined to pass a short integer representing the maximum number of credits for which a student can register. Therefore, the definition for the `TooManyCredits` exception contains the following OMG IDL statements:

   ```
   exception TooManyCredits
   {
       unsigned short maximum_credits;
   };
   ```

2. In the definition of the operations that throw the exception, include the exception. The following example shows the OMG IDL statements for the `register_for_courses()` operation on the `Registrar` interface:

   ```
   NotRegisteredList register_for_courses(
       in StudentId        student,
       in CourseNumberList courses
   ) raises (
       TooManyCredits
   );
   ```

Throwing the Exception

In the implementation of the operation that uses the exception, write the code that throws the exception, as in the following C++ example.

```
if ( ... ) {
    UniversityZ::TooManyCredits e;
    e.maximum_credits = 18;
    throw e;
```

How the Transactions University Sample Application Works

This topic includes the following sections:

About the Transactions University Sample Application

To implement the student registration process, the Transactions sample application does the following:

- The client application obtains a reference to the TransactionCurrent object from the Bootstrap object.
- When the student submits the list of courses for which he or she wants to register, the client application:
  a. Begins a transaction by invoking the `Current::begin()` operation on the TransactionCurrent object.
  b. Invokes the `register_for_courses()` operation on the Registrar object, passing a list of courses.
- The `register_for_courses()` operation on the Registrar object processes the registration request by executing a loop that does the following iteratively for each course in the list:
  a. Checks to see how many credits the student is already registered for.
  b. Adds the course to the list of courses for which the student is registered.

  The Registrar object checks for the following potential problems, which prevent the transaction from being committed:
  – The student is already registered for the course.
  – A course in the list does not exist.

  - The student exceeds the maximum credits allowed.

- As defined in the application's OMG IDL, the `register_for_courses()` operation returns a parameter to the client application, `NotRegisteredList`, which contains a list of the courses for which the registration failed.

  If the `NotRegisteredList` value is empty, the client application commits the transaction.

  If the `NotRegisteredList` value contains any courses, the client application queries the student to indicate whether he or she wants to complete the registration process for the courses for which the registration succeeded. If the user chooses to complete the registration, the client application commits the transaction. If the user chooses to cancel the registration, the client application rolls back the transaction.

- If the registration for a course has failed because the student exceeds the maximum number of credits he or she can take, the Registrar object returns a `TooManyCredits` exception to the client application, and the client application rolls back the entire transaction.

Transactional Model Used by the Transactions University Sample Application

The basic design rationale for the Transactions sample application is to handle course registrations in groups, as opposed to one at a time. This design helps to minimize the number of remote invocations on the Registrar object.

In implementing this design, the Transactions sample application shows one model of the use of transactions, which were described in "Integrating Transactions in a BEA Tuxedo Client and Server Application" on page 3-2. The model is as follows:

- The client begins a transaction by invoking the `begin()` operation on the TransactionCurrent object, followed by making an invocation to the `register_for_courses()` operation on the Registrar object.

  The Registrar object registers the student for the courses for which it can, and then returns a list of courses for which the registration process was unsuccessful. The client application can choose to commit the transaction or roll it back. The transaction encapsulates this conversation between the client and the server application.

- The `register_for_courses()` operation performs multiple checks of the University database. If any one of those checks fail, the transaction can be rolled back.

Object State Considerations for the University Server Application

Because the Transactions University sample application is transactional, the University server application generally needs to consider the implications on object state, particularly in the event of a rollback. In cases where there is a rollback, the server application must ensure that all affected objects have their durable state restored to the proper state.

Because the Registrar object is being used for database transactions, a good design choice for this object is to make it transactional (assign the `always` transaction policy to this object's interface). If a transaction has not already been scoped when this object is invoked, the BEA Tuxedo system will start a transaction automatically.

By making the Registrar object automatically transactional, all database write operations performed by this object will always be done within the scope of a transaction, regardless of whether the client application starts one. Since the server application uses an XA resource manager, and since the object is guaranteed to be in a transaction when the object writes to a database, the object does not have any rollback or commit responsibilities because the XA resource manager takes responsibility for these database operations on behalf of the object.

The RegistrarFactory object, however, can be excluded from transactions because this object does not manage data that is used during the course of a transaction. By excluding this object from transactions, you minimize the processing overhead implied by transactions.

Object Policies Defined for the Registrar Object

To make the Registrar object transactional, the ICF file specifies the `always` transaction policy for the `Registrar` interface. Therefore, in the Transaction sample application, the ICF file specifies the following object policies for the `Registrar` interface:

Activation Policy	Transaction Policy
`process`	`always`

Object Policies Defined for the RegistrarFactory Object

To exclude the RegistrarFactory object from transactions, the ICF file specifies the `ignore` transaction policy for the `Registrar` interface. Therefore, in the Transaction sample application, the ICF file specifies the following object policies for the `RegistrarFactory` interface:

Activation Policy	Transaction Policy
process	ignore

Using an XA Resource Manager in the Transactions Sample Application

The Transactions sample application uses the Oracle Transaction Manager Server (TMS), which handles object state data automatically. Using any XA resource manager imposes specific requirements on how different objects managed by the server application may read and write data to that database, including the following:

- Some XA resource managers, such as Oracle7, require that all database operations be scoped within a transaction. This means that the CourseSynopsisEnumerator object needs to be scoped within a transaction because this object reads from a database.
- When a transaction is committed or rolled back, the XA resource manager automatically handles the durable state implied by the commit or rollback. That is, if the transaction is committed, the XA resource manager ensures that all database updates are made permanent. Likewise, if there is a rollback, the XA resource manager automatically restores the database to its state prior to the beginning of the transaction.

  This characteristic of XA resource managers actually makes the design problems associated with handling object state data in the event of a rollback much simpler. Transactional objects can always delegate the commit and rollback responsibilities to the XA resource manager, which greatly simplifies the task of implementing a server application.

Configuration Requirements for the Transactions Sample Application

The University sample applications use an Oracle Transaction Manager Server (TMS). To use the Oracle database, you must include specific Oracle-provided files in the server application build process. For more information about building, configuring, and running the Transactions sample application, see *The Transaction Sample Application* in the BEA Tuxedo online documentation. For more information about the configurable settings in the `UBBCONFIG` file, see "Modifying the UBBCONFIG File to Accommodate Transactions" on page 5-2.

CHAPTER 4

Transactions in CORBA Client Applications

This topic includes the following sections:

This topic describes how to use transactions in CORBA C++ client applications for the BEA Tuxedo CORBA software. Before you begin, you should read Chapter 1, "Introducing Transactions."

Notes: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

For an example of how transactions are implemented in working client applications, see the *The Transaction Sample Application* in the BEA Tuxedo online documentation. For an overview of the TransactionCurrent object, see "Client Application Development Concepts" in *Creating CORBA Client Applications*.

Overview of BEA Tuxedo CORBA Transactions

Client applications use transaction processing to ensure that data remains correct, consistent, and persistent. The transactions in the BEA Tuxedo software allow client applications to begin and terminate transactions and to get the status of transactions. The BEA Tuxedo software uses transactions as defined in the CORBA services Object Transaction Service, with extensions for ease of use.

Transactions are defined on *interfaces*. The application designer decides which interfaces within a BEA Tuxedo client/server application will handle transactions. Transaction policies are defined in the Implementation Configuration File (ICF) for server applications. Generally, the ICF file for the available interfaces is provided to the client programmer by the application designer.

Summary of the Development Process for Transactions

To add transactions to a client application, complete the following steps:

- Step 1: Using the Bootstrap Object to Obtain the TransactionCurrent Object
- JStep 2: Using the TransactionCurrent Methods

The rest of this topic describes these steps using portions of the client applications in the Transactions University sample application. For information about the Transactions University sample application, see *The Transactions Sample Application* in the BEA Tuxedo online documentation.

The Transactions University sample application is located in the following directory on the BEA Tuxedo software kit:

- For Microsoft Windows systems:
  `drive:\tuxdir\samples\corba\university\transactions`
- For UNIX systems:
  `drive:/tuxdir/samples/corba/university/transactions`

Step 1: Using the Bootstrap Object to Obtain the TransactionCurrent Object

If you are using the BEA Tuxedo CORBA client software, you should use the Bootstrap object to obtain an object reference to the TransactionCurrent object for the specified BEA Tuxedo

domain. For more information about the TransactionCurrent object, see "Client Application Development Concepts" in *Creating CORBA Client Applications*.

Note: If you are using a third-party client ORB, you should the CORBA Interoperable Naming Service (INS) `CORBA::ORB::resolve_initial_references` operation to obtain an object reference to the FactoryFinder object for the specified BEA Tuxedo domain. For information on how to use INS to get initial object references for transaction clients, see "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

The following C++ examples illustrate how the Bootstrap object is used to return the TransactionCurrent object.

C++ Example

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var transaction_current_oref=
     CosTransactions::Current::_narrow(
              var_transaction_current_oref.in());
```

JStep 2: Using the TransactionCurrent Methods

The TransactionCurrent object has *methods* that allow a client application to manage transactions. These methods can be used to begin and end transactions and to obtain information about the current transaction.

Table 4-1 describes the methods in the TransactionCurrent object.

Table 4-1 Methods in the TransactionCurrent Object

Method	Description
`begin`	Creates a new transaction. Future operations take place within the scope of this transaction. When a client application begins a transaction, the default transaction timeout is 300 seconds. You can change this default, using the `set_timeout` method.
`commit`	Ends the transaction successfully. Indicates that all operations on this client application have completed successfully.

Table 4-1 Methods in the TransactionCurrent Object (Continued)

Method	Description
`rollback`	Forces the transaction to roll back.
`rollback_only`	Marks the transaction so that the only possible action is to roll back. Generally, this method is used only in server applications.
`suspend`	Suspends participation in the current transaction. This method returns an object that identifies the transaction and allows the client application to resume the transaction later.
`resume`	Resumes participation in the specified transaction.
`get_status`	Returns the status of a transaction with a client application.
`get_transaction_name`	Returns a printable string describing the transaction.
`set_timeout`	Modifies the timeout period associated with transactions. The default transaction timeout value is 300 seconds. If a transaction is automatically started instead of explicitly started with the `begin` method, the timeout value is determined by the value of the `TRANTIME` parameter in the `UBBCONFIG` file. For more information about setting the `TRANTIME` parameter, see Chapter , "Administering Transactions."
`get_control`	Returns a control object that represents the transaction.

A basic transaction works in the following way:

1. A client application begins a transaction using the `Tobj::TransactionCurrent::begin` method. This method does not return a value.
2. The operations on the CORBA interface execute within the scope of a transaction. If a call to any of these operations raises an exception (either explicitly or as a result of a communications failure), the exception can be caught and the transaction can be rolled back.
3. Use the `Tobj::TransactionCurrent::commit` method to commit the current transaction. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

The association between the transaction and the client application ends when the client application calls the `Tobj::TransactionCurrent:commit` method or the `Tobj::TransactionCurrent:rollback` method.The following C++ examples illustrate using a transaction to encapsulate the operation of a student registering for a class.

C++ Example

```
//Begin the transaction
transaction_current_oref->begin();
try {
//Perform the operation inside the transaction
    pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     ...
//If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     transaction_current_ref->commit(report_heuristics);
}
catch (CORBA::Exception &) {
//If the operation has problems executing, rollback the
//transaction. Then throw the original exception again.
//If the rollback fails, ignore the exception and throw the
//original exception again.
try {
     transaction_current_ref->rollback();
}
catch (CORBA::Exception &) {
            TP::userlog("rollback failed");

throw;
}
```

CHAPTER 5

Administering Transactions

This topic includes the following sections:

- Modifying the UBBCONFIG File to Accommodate Transactions
- Modifying the Domain Configuration File to Support Transactions (BEA Tuxedo CORBA Servers)
- Sample Distributed Application Using Transactions

Before you begin, you should read Chapter 1, "Introducing Transactions."

Notes: The administrative information applies whether you are using the Bootstrap object or the CORBA interoperable Naming Service (INS) to obtain initial object references to the BEA Tuxedo ORB.

The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party Java ORB libraries, and for programmer reference only.

Technical support for third party CORBA Java ORBs should be provided by their respective vendors. BEA Tuxedo does not provide any technical support or documentation for third party CORBA Java ORBs.

Modifying the UBBCONFIG File to Accommodate Transactions

This topic includes the following sections:

Summary of Steps

To accommodate transactions, you must modify the `RESOURCES`, `MACHINES`, `GROUPS`, and the `INTERFACES` or `SERVICES` sections of the application's `UBBCONFIG` file in the following ways:

- In the `RESOURCES` section, specify the application-wide number of allowed transactions and the value of the commit control flag.
- In the `MACHINES` section, create the `TLOG` information for each machine.
- In the `GROUPS` section, indicate information about each resource manager and about the Transaction Manager Server.
- In the `INTERFACES` section (for BEA Tuxedo CORBA applications only) or the `SERVICES` section (for BEA Tuxedo ATMI applications only), enable the automatic transaction option.

For instructions about modifying these sections in the `UBBCONFIG` file, see "Creating a Configuration File" in the *Setting Up a BEA Tuxedo Application*.

Step 1: Specify Application-wide Transactions in the RESOURCES Section

Table 5-1 provides a description of transaction-related parameters in the `RESOURCES` section of the configuration file.

Table 5-1 Transaction-related Parameters in the RESOURCES Section

Parameter	Meaning
MAXGTT	Limits the total number of global transaction identifiers (GTRIDs) allowed on one machine at one time. The maximum value allowed is 2048, the minimum is 0, and the default is 100. You can override this value on a per-machine basis in the MACHINES section. Entries remain in the table only while the global transaction is active, so this parameter has the effect of setting a limit on the number of simultaneous transactions.
CMTRET	Specifies the initial setting of the TP_COMMIT_CONTROL characteristic. The default is COMPLETE. Following are its two settings: • LOGGED—the TP_COMMIT_CONTROL characteristic is set to TP_CMT_LOGGED, which means that tpcommit() returns when all the participants have successfully precommitted. • COMPLETE—the TP_COMMIT_CONTROL characteristic is set to TP_CMT_COMPLETE, which means that tpcommit() will not return until all the participants have successfully committed. **Note:** You should consult with the RM vendors to determine the appropriate setting. If any RM in the application uses the *late commit* implementation of the XA standard, the setting should be COMPLETE. If all the resource managers use the *early commit* implementation, the setting should be LOGGED for performance reasons. (You can override this setting with tpscmt().)

Step 2: Create a Transaction Log (TLOG)

This section discusses creating a transaction log (TLOG), which refers to a log in which information on transactions is kept until the transaction is completed.

Creating the UDL

The Universal Device List (UDL) is like a map of the BEA Tuxedo file system. The UDL gets loaded into shared memory when an application is booted. To create an entry in the UDL for the TLOG device, create the UDL on each machine using global transactions. If the TLOGDEVICE is mirrored between two machines, it is unnecessary to do this on the paired machine. The Bulletin Board Liaison (BBL) then initializes and opens the TLOG during the boot process.

To create the UDL, enter a command using the following format, before the application has been booted:

```
tmadmin -c crdl -z config -b blocks
```

where:

`-z config`	Specifies the full pathname for the device where you should create the UDL.
`-b blocks`	Specifies the number of blocks to be allocated on the device.
config	Should match the value of the `TLOGDEVICE` parameter in the `MACHINES` section of the `UBBCONFIG` file.

Note: In general, the value that you supply for `blocks` should not be less than the value for `TLOGSIZE`. For example, if `TLOGSIZE` is specified as 200 blocks, specifying `-b 500` would not cause a degradation.

For more information about storing the `TLOG`, see *Installing the BEA Tuxedo System*.

Defining Transaction-related Parameters in the MACHINES Section

You can define a global transaction log (`TLOG`) using several parameters in the `MACHINES` section of the `UBBCONFIG` file. You must manually create the device list entry for the `TLOGDEVICE` on each machine where a `TLOG` is needed. You can do this either before or after `TUXCONFIG` has been loaded, but it must be done before the system is booted.

Note: If you are not using transactions, the `TLOG` parameters are not required.

Table 5-2 provides a description of transaction-related parameters in the `MACHINES` section of the configuration file.

Table 5-2 Transaction-related Parameters in the MACHINES Section

Parameter	Meaning
`TLOGNAME`	The name of the `DTP` transaction log for this machine.
`TLOGDEVICE`	Specifies the BEA Tuxedo or BEA Tuxedo file system that contains the `DTP` transaction log (`TLOG`) for this machine. If this parameter is not specified, the machine is assumed not to have a `TLOG`. The maximum string value length is 64 characters.

Table 5-2 Transaction-related Parameters in the MACHINES Section (Continued)

Parameter	Meaning
TLOGSIZE	The size of the TLOG file in physical pages. Its value must be between 1 and 2048, and its default is 100. The value should be large enough to hold the number of outstanding transactions on the machine at a given time. One transaction is logged per page. The default should suffice for most applications.
TLOGOFFSET	Specifies the offset in pages from the beginning of TLOGDEVICE to the start of the VTOC that contains the transaction log for this machine.The number must be greater than or equal to 0 and less than the number of pages on the device. The default is 0. TLOGOFFSET is rarely necessary. However, if two VTOCs share the same device or if a VTOC is stored on a device (such as a file system) that is shared with another application, you can use TLOGOFFSET to indicate a starting address relative to the address of the device.

Creating the Domains Transaction Log (BEA Tuxedo ATMI Servers Only)

This section applies to the ATMI servers only.

You can create the Domains transaction log before starting the Domains gateway group by using the following command:

```
dmadmin(1) crdmlog (crdlog) -d local_domain_name
```

Create the Domains transaction log for the named local domain on the current machine (the machine on which dmadmin is running). The command uses the parameters specified in the DMCONFIG file. This command fails if the named local domain is active on the current machine or if the log already exists. If the transaction log has not been created, the Domains gateway group creates the log when it starts up.

Step 3: Define Each Resource Manager (RM) and the Transaction Manager Server in the GROUPS Section

Additions to the GROUPS section fall into two categories:

- Defining the Transaction Manager Servers that perform most of the work that controls global transactions:
  - The TMSNAME parameter specifies the name of the server executable.

– The TMSCOUNT parameter specifies the number of such servers to boot (the minimum is 2, the maximum is 10, and the default is 3).

A NULL Transactional Manager Server does not communicate with any resource manager. It is used to exercise an application's use of the transactional primitives before actually testing the application in a recoverable, *real* environment. This server is named TMS and it simply begins, commits, or terminates without talking to any resource manager.

- Defining opening and closing information for each resource manager:
  – OPENINFO is a string with information used to open a resource manager.
  – CLOSEINFO is used to close a resource manager.

Sample GROUPS Section

The following sample GROUPS section derives from the bankapp banking application:

```
BANKB1 GRPNO=1 TMSNAME=TMS_SQL TMSCOUNT=2
OPENINFO="TUXEDO/SQL:<APPDIR>/bankdl1:bankdb:readwrite"
```

Table 5-3 describes the transaction values specified in this sample GROUPS section.

Table 5-3 Transaction Values in the GROUPS Section of a Sample UBBCONFIG File

Transaction Value	Meaning
BANKB1 GRPNO=1 TMSNAME=TMS_SQL\ TMSCOUNT=2	Contains the name of the Transaction Manager Server (TMS_SQL) and the number (2) of these servers to be booted in the group BANKB1
TUXEDO/SQL	Published name of the resource manager
<APPDIR>/bankdl1	Includes a device name
bankdb	Database name
readwrite	Access mode

Characteristics of the TMSNAME, TMSCOUNT, OPENINFO, and CLOSEINFO Parameters

Table 5-4 lists the characteristics of the TMSNAME, TMSCOUNT, OPENINFO, and CLOSEINFO parameters.

Table 5-4 Characteristics of the TMSNAME, TMSCOUNT, OPENINFO, and CLOSEINFO Parameters

Parameter	Characteristics
TMSNAME	Name of the Transaction Manager Server executable. Required parameter for transactional configurations. TMS is a NULL Transactional Manager Server.
TMSCOUNT	Number of Transaction Manager Servers (must be between 2 and 10). Default is 3.
OPENINFO CLOSEINFO	Represents information to open or close a resource manager. Content depends on the specific resource manager. Starts with the name of the resource manager. Omission means the resource manager needs no information to open.

Step 4: Enable an Interface to Begin a Transaction

To enable an interface to begin a transaction, you change different sections in the UBBCONFIG file, depending on whether you are configuring a BEA Tuxedo CORBA server or BEA Tuxedo ATMI server.

- Changing the INTERFACES Section (BEA Tuxedo CORBA Servers)
- Changing the SERVICES Section (BEA Tuxedo ATMI Servers)

Changing the INTERFACES Section (BEA Tuxedo CORBA Servers)

The INTERFACES section in the UBBCONFIG file supports BEA Tuxedo CORBA interfaces:

- For each CORBA interface, set AUTOTRAN to Y if you want a transaction to start automatically when an operation invocation is received. AUTOTRAN=Y has no effect if the interface is already in transaction mode. The default is N. The effect of specifying a value for AUTOTRAN depends on the transactional policy specified by the developer in the Implementation Configuration File (ICF) for the interface. This transactional policy will become the transactional policy attribute of the associated T_IFQUEUE MIB object at run time. The only time this value affects the behavior of the application is if the developer specified a transaction policy of optional.

Note: To work properly, this feature depends on collaboration between the system designer and the administrator. If the administrator sets this value to `Y` without prior knowledge of the transaction policy defined by the developer in the interface's ICF, the actual run time effect of the parameter might be unknown.

- If `AUTOTRAN` is set to `Y`, you must set the `TRANTIME` parameter, which specifies the transaction timeout, in seconds, for the transactions to be created. The value must be greater than or equal to zero and must not exceed 2,147,483,647 (2^{31} - 1, or about 70 years). A value of zero implies there is no timeout for the transaction. (The default is 30 seconds.)

Table 5-5 describes the characteristics of the `AUTOTRAN`, `TRANTIME`, and `FACTORYROUTING` parameters.

Table 5-5 Characteristics of the AUTOTRAN, TRANTIME, and FACTORYROUTING Parameters

Parameter	Characteristics
`AUTOTRAN`	• Makes an interface the initiator of a transaction. • To work properly, it is dependent on collaboration between the system designer and the system administrator. If the administrator sets this value to `Y` without prior knowledge of the ICF transaction policy set by the developer, the actual run-time effort of the parameter might be unknown. • The only time this value affects the behavior of the application is if the developer specified a transaction policy of `optional`. • If a transaction already exists, a new one is not started. • Default is `N`.
`TRANTIME`	• Represents the timeout for the `AUTOTRAN` transactions. • Valid values are between 0 and 2^{31} - 1, inclusive. • Zero (0) represents no timeout. • Default is 30 seconds.
`FACTORYROUTING`	• Specifies the name of the routing criteria to be used for factory-based routing for this CORBA interface. • You must specify a `FACTORYROUTING` parameter for interfaces requesting factory-based routing.

Changing the SERVICES Section (BEA Tuxedo ATMI Servers)

The following are three transaction-related features in the SERVICES section:

- If you want a service (instead of a client) to begin a transaction, you must set the AUTOTRAN flag to Y. This is useful if the service is not needed as part of any larger transaction, and if the application wants to relieve the client of making transaction decisions. If the service is called when there is already an existing transaction, this call becomes part of it. (The default is N.)

  Note: Generally, clients are the best initiators of transactions because a service has the potential of participating in a larger transaction.

- If AUTOTRAN is set to Y, you must set the TRANTIME parameter, which is the transaction timeout, in seconds, for the transactions to be created. The value must be greater than or equal to 0 and must not exceed 2,147,483,647 (2^{31} - 1, or about 70 years). A value of zero implies there is no timeout for the transaction. (The default is 30 seconds.)

- You must specify a ROUTING parameter for transactions that request data-dependent routing.

Table 5-6 describes the characteristics of the AUTOTRAN, TRANTIME, and ROUTING parameters:

Table 5-6 Characteristics of the AUTOTRAN, TRANTIME, and ROUTING Parameters

Parameter	Characteristics
AUTOTRAN	Makes a service the initiator of a transaction. Relieves the client of the transactional burden. If a transaction already exists, a new one is not started. Default is N.
TRANTIME	Represents the timeout for the AUTOTRAN transactions. Valid values are between 0 and 2^{31} - 1, inclusive. 0 represents no timeout. Default is 30 seconds.
ROUTING	Points to an entry in the ROUTING section where data-dependent routing is specified for transactions that request this service.

Modifying the Domain Configuration File to Support Transactions (BEA Tuxedo CORBA Servers)

This topic includes the following sections:

To enable transactions across domains, you need to set parameters in both the `DM_LOCAL_DOMAINS` and the `DM_REMOTE_SERVICES` sections of the Domains configuration file (`DMCONFIG`). Entries in the `DM_LOCAL_DOMAINS` section define local domain characteristics. Entries in the `DM_REMOTE_SERVICES` section define information on services that are *imported* and that are available on remote domains.

Characteristics of the DMTLOGDEV, DMTLOGNAME, DMTLOGSIZE, MAXRDTRAN, and MAXTRAN Parameters

The `DM_LOCAL_DOMAINS` section of the Domains configuration file identifies local domains and their associated gateway groups. This section must have an entry for each gateway group (local domain). Each entry specifies the parameters required for the Domains gateway processes running in that group.

Table 5-7 provides a description of the five transaction-related parameters in this section: `DMTLOGDEV`, `DMTLOGNAME`, `DMTLOGSIZE`, `MAXRDTRAN`, and `MAXTRAN`.

Table 5-7 Characteristics of the DMTLOGDEV, DMTLOGNAME, DMTLOGSIZE, MAXRDTRAN, and MAXTRAN Parameters

Parameter	Characteristics
DMTLOGDEV	Specifies the BEA Tuxedo file system that contains the Domains transaction log (DMTLOG) for this machine. The DMTLOG is stored as a BEA Tuxedo VTOC table on the device. If this parameter is not specified, the Domains gateway group is not allowed to process requests in transaction mode. Local domains running on the same machine can share the same DMTLOGDEV file system, but each local domain must have its own log (a table in the DMTLOGDEV) named as specified by the DMTLOGNAME keyword.
DMTLOGNAME	Specifies the name of the Domains transaction log for this domain. This name must be unique when the same DMTLOGDEV is used for several local domains. If a value is not specified, the value defaults to the string DMTLOG. The name must contain 30 characters or less.
DMTLOGSIZE	Specifies the numeric size of the Domains transaction log for this machine (in pages). It must be greater than zero and less than the amount of available space on the BEA Tuxedo file system. If a value is not specified, the value defaults to 100 pages. **Note:** The number of domains in a transaction determine the number of pages you must specify in the DMTLOGSIZE parameter. One transaction does not necessarily equal one log page.
MAXRDTRAN	Specifies the maximum number of domains that can be involved in a transaction. It must be greater than zero and less than 32,768. If a value is not specified, the value defaults to 16.
MAXTRAN	Specifies the maximum number of simultaneous global transactions allowed on this local domain. It must be greater than or equal to zero, and less than or equal to the MAXGTT parameter specified in the TUXCONFIG file. If not specified, the default is the value of MAXGTT.

Characteristics of the AUTOTRAN and TRANTIME Parameters (BEA Tuxedo CORBA and ATMI Servers)

The DM_REMOTE_SERVICES section of the Domains configuration file identifies information on services *imported* and available on remote domains. Remote services are associated with a particular remote domain.

Table 5-8 describes the two transaction-related parameters in this section: AUTOTRAN and TRANTIME.

Table 5-8 Characteristics of the AUTOTRAN and TRANTIME Parameters

Parameter	Characteristics
AUTOTRAN	Used by gateways to automatically start/terminate transactions for remote services. This capability is required if you want to enforce reliable network communication with remote services. You specify this capability by setting the AUTOTRAN parameter to Y in the corresponding remote service definition.
TRANTIME	Specifies the default timeout value in seconds for a transaction automatically started for the associated service. The value must be greater than or equal to zero, and less than 2147483648. The default is 30 seconds. A value of zero implies the maximum timeout value for the machine.

Sample Distributed Application Using Transactions

This topic includes the following sections:

This topic describes a sample configuration file for a sample CORBA application that enables transactions and distributes the application over three sites. The application includes the following features:

- Data-dependent routing on `ACCOUNT_ID`.
- Data distributed over three databases.
- `BRIDGE` processes communicating with the system via the `ATMI` interface.
- System administration from one site.

The configuration file includes seven sections: `RESOURCES`, `MACHINES`, `GROUPS`, `NETWORK`, `SERVERS`, `SERVICES`, and `ROUTING`.

RESOURCES Section

The `RESOURCES` section shown in Listing 5-1 specifies the following parameters:

- `MAXSERVERS`, `MAXSERVICES`, and `MAXGTT` are less than the defaults. This makes the Bulletin Board smaller.
- `MASTER` is `SITE3` and the backup master is `SITE1`.
- `MODEL` is set to `MP` and `OPTIONS` is set to `LAN`, `MIGRATE`. This allows a networked configuration with migration.
- `BBLQUERY` is set to 180 and `SCANUNIT` is set to 10. This means that `DBBL` checks of the remote `BBLs` are done every 1800 seconds (one half hour).

Listing 5-1 Sample RESOURCES Section

```
*RESOURCES
#
IPCKEY       99999
UID          1
GID          0
PERM         0660
MAXACCESSERS 25
MAXSERVERS   25
MAXSERVICES  40
MAXGTT       20
MASTER       SITE3, SITE1
SCANUNIT     10
SANITYSCAN   12
BBLQUERY     180
BLOCKTIME    30
DBBLWAIT     6
OPTIONS      LAN, MIGRATE
MODEL        MP
LDBAL         Y
```

MACHINES Section

The MACHINES section shown in Listing 5-2 specifies the following parameters:

- TLOGDEVICE and TLOGNAME are specified, which indicate that transactions will be done.
- The TYPE parameters are all different, which indicates that encode/decode will be done on all messages sent between machines.

Listing 5-2 Sample MACHINES Section

```
*MACHINES
Gisela       LMID=SITE1
              TUXDIR="/usr/tuxedo"
              APPDIR="/usr/home"
```

```
                ENVFILE="/usr/home/ENVFILE"
                TLOGDEVICE="/usr/home/TLOG"
                TLOGNAME=TLOG
                TUXCONFIG="/usr/home/tuxconfig"
                TYPE="3B600"

romeo           LMID=SITE2
                TUXDIR="/usr/tuxedo"
                APPDIR="/usr/home"
                ENVFILE="/usr/home/ENVFILE"
                TLOGDEVICE="/usr/home/TLOG"
                TLOGNAME=TLOG
                TUXCONFIG="/usr/home/tuxconfig"
                TYPE="SEQUENT"

juliet          LMID=SITE3
                TUXDIR="/usr/tuxedo"
                APPDIR='/usr/home"
                ENVFILE="/usr/home/ENVFILE"
                TLOGDEVICE="/usr/home/TLOG"
                TLOGNAME=TLOG
                TUXCONFIG="/usr/home/tuxconfig"
                TYPE="AMDAHL"
```

GROUPS and NETWORK Sections

The `GROUPS` and `NETWORK` sections shown in Listing 5-3 specify the following parameters:

- The `TMSCOUNT` is set to 2, which means that only two `TMS_SQL` transaction manager servers will be booted per group.
- The `OPENINFO` string indicates that the application will perform database access.

Listing 5-3 Sample GROUPS and NETWORK Sections

```
*GROUPS
DEFAULT:         TMSNAME=TMS_SQL       TMSCOUNT=2
BANKB1           LMID=SITE1            GRPNO=1
 OPENINFO="TUXEDO/SQL:/usr/home/bankdl1:bankdb:readwrite"
BANKB2           LMID=SITE2            GRPNO=2
 OPENINFO="TUXEDO/SQL:/usr/home/bankdl2:bankdb:readwrite"
BANKB3            LMID=SITE3            GRPNO=3
 OPENINFO="TUXEDO/SQL:/usr/home/bankdl3:bankdb:readwrite"

*NETWORK
SITE1             NADDR="0X0002ab117B2D4359"
                  BRIDGE="/dev/tcp"
                  NLSADDR="0X0002ab127B2D4359"

SITE2             NADDR="0X0002ab117B2D4360"
                  BRIDGE="/dev/tcp"
                  NLSADDR="0X0002ab127B2D4360"

SITE3             NADDR="0X0002ab117B2D4361"
                  BRIDGE="/dev/tcp"
                  NLSADDR="0X0002ab127B2D4361"
```

SERVERS, SERVICES, and ROUTING Sections

The SERVERS, SERVICES, and ROUTING sections shown in Listing 5-4 specify the following parameters:

- The TLR servers have a `-T number` passed to their *`tpsrvrinit()`* functions.
- All requests for the services are routed on the ACCOUNT_ID field.
- None of the services will be performed in AUTOTRAN mode.

Listing 5-4 Sample SERVERS, SERVICES, and ROUTING Sections

```
*SERVERS
DEFAULT: RESTART=Y MAXGEN=5 REPLYQ=N CLOPT="-A"
TLR      SRVGRP=BANKB1    SRVID=1    CLOPT="-A -- -T 100"
TLR      SRVGRP=BANKB2    SRVID=3    CLOPT="-A -- -T 400"
TLR      SRVGRP=BANKB3    SRVID=4    CLOPT="-A -- -T 700"
XFER     SRVGRP=BANKB1    SRVID=5    REPLYQ=Y
XFER     SRVGRP=BANKB2    SRVID=6    REPLYQ=Y
XFER     SRVGRP=BANKB3    SRVID=7    REPLYQ=Y

*SERVICES
DEFAULT:    AUTOTRAN=N
WITHDRAW        ROUTING=ACCOUNT_ID
DEPOSIT         ROUTING=ACCOUNT_ID
TRANSFER        ROUTING=ACCOUNT_ID
INQUIRY         ROUTING=ACCOUNT_ID

*ROUTING
ACCOUNT_ID      FIELD=ACCOUNT_ID     BUFTYPE="FML"
                     RANGES="MON - 9999:*,
                     10000 - 39999:BANKB1
                     40000 - 69999:BANKB2
                     70000 - 100000:BANKB3
                        ""
```