BEA Tuxedo®

Creating CORBA Server Applications

Version 10.0
Document Released:August 28, 2007

Contents

1. CORBA Server Application Concepts

The Entities You Create to Build a CORBA Server Application 1-1
The Implementation of the CORBA Objects for Your Server Application 1-2
How Interface Definitions Establish the Operations on a CORBA Object . . . 1-2
How You Implement the Operations on a CORBA Object 1-3
How Client Applications Access and Manipulate Your Application's CORBA Objects 1-4
How You Instantiate a CORBA Object at Run Time 1-6
The Server Object 1-7
Process for Developing CORBA Server Applications 1-8
Generating Object References 1-8
How Client Applications Find Your Server Application's Factories 1-9
Creating an Active Object Reference 1-9
Managing Object State 1-9
About Object State 1-10
Object Activation Policies 1-11
Application-controlled Deactivation 1-13
Reading and Writing an Object's Data 1-14
Available Mechanisms for Reading and Writing an Object's Durable State . 1-15
Reading State at Object Activation 1-18
Reading State Within Individual Operations on an Object 1-18
Stateless Objects and Durable State 1-19

Stateful Objects and Durable State 1-20
Your Responsibilities for Object Deactivation 1-20
Avoiding Unnecessary I/O 1-20
Sample Activation Walkthrough 1-21
Using Design Patterns 1-21
Process-Entity Design Pattern 1-22
List-Enumerator Design Pattern 1-22

2. Steps for Creating a BEA Tuxedo CORBA Server Application

Summary of the CORBA Server Application Development Process 2-2
Step 1: Compile the OMG IDL File for the Server Application 2-2
Using the IDL Compiler 2-4
Generating the Skeleton and Implementation Files 2-5
Generating Tie Classes 2-5
Step 2: Write the Methods That Implement Each Interface's Operations 2-5
The Implementation File Generated by the IDL Compiler 2-6
Implementing a Factory 2-6
Step 3: Create the Server Object 2-7
Initializing the Server Application 2-8
Writing the Code That Creates and Registers a Factory 2-9
Creating Servants 2-10
Releasing the Server Application 2-11
Step 4: Define the In-memory Behavior of Objects 2-12
Specifying Object Activation and Transaction Policies in the ICF File 2-13
Step 5: Compile and Link the Server Application 2-16
Step 6: Deploy the Server Application 2-16
Development and Debugging Tips 2-17
Use of CORBA Exceptions and the User Log 2-17

Client Application View of Exceptions 2-18
Server Application View of Exceptions 2-18
Detecting Error Conditions in the Callback Methods 2-22
Common Pitfalls of OMG IDL Interface Versioning and Modification 2-23
Caveat for State Handling in Tobj_ServantBase::deactivate_object() 2-24
Servant Pooling 2-25
How Servant Pooling Works 2-25
How You Implement Servant Pooling 2-25
Delegation-based Interface Implementation 2-26
About Tie Classes in the BEA Tuxedo System 2-26
When to Use Tie Classes 2-29
How to Create Tie Classes in a CORBA Application 2-29

3. Designing and Implementing a Basic CORBA Server Application

How the Basic University Sample Application Works 3-2
The Basic University Sample Application OMG IDL 3-2
Application Startup 3-3
Browsing Course Synopses 3-4
Browsing Course Details 3-6
Design Considerations for the University Server Application 3-6
Design Considerations for Generating Object References 3-7
Design Considerations for Managing Object State 3-9
The RegistrarFactory Object 3-9
The Registrar Object 3-9
The CourseSynopsisEnumerator Object 3-9
Basic University Sample Application ICF File 3-10
Design Considerations for Handling Durable State Information 3-11

The Registrar Object 3-11
The CourseSynopsisEnumerator Object 3-12
Using the University Database. 3-12
How the Basic Sample Application Applies Design Patterns. 3-13
Process-Entity Design Pattern 3-13
List-Enumerator Design Pattern 3-14
Additional Performance Efficiencies Built into the BEA Tuxedo System 3-15
Preactivating an Object with State 3-15
How You Preactivate an Object with State 3-16
Usage Notes for Preactivated Objects 3-16

4. Creating Multithreaded CORBA Server Applications

Overview 4-2
Introduction. 4-2
Requirements, Goals, and Concepts 4-3
Threading Models 4-5
Reentrant Servants. 4-7
The Current Object 4-7
Mechanisms for Supporting Multithreaded CORBA Servers. 4-8
Context Services 4-8
Classes and Methods in the TP Framework. 4-9
Capabilities in the Build Commands 4-10
Tools for Administration 4-10
Running Single-threaded Server Applications in a Multithreaded System 4-11
Developing and Building Multithreaded CORBA Server Applications 4-12
Using the buildobjserver Command 4-12
Platform-specific Thread Libraries 4-12
Specifying Multithreaded Support. 4-12

Specifying an Alternate Server Class 4-13
Using the buildobjclient Command 4-14
Creating Non-reentrant Servants 4-14
Creating Reentrant Servants 4-15
Considerations for Client Applications 4-15
Building and Running the Multithreaded Simpapp Sample Application 4-16
About the Simpapp Multithreaded Sample 4-16
How the Sample Application Works 4-16
OMG IDL Code for the Simpapp Multithreaded Sample Application 4-18
How to Build and Run the Sample Application 4-19
Setting the TUXDIR Environment Variable 4-20
Verifying the TUXDIR Environment Variable 4-20
Changing the Setting of the Environment Variable 4-20
Creating a Working Directory for the Sample Application 4-21
Checking Permissions on All the Files 4-24
Executing the runme Command 4-25
Running the Sample Application Step-by-Step 4-29
Shutting Down the Sample Application 4-32
Multithreaded CORBA Server Application Administration 4-33
Specifying Thread Pool Size 4-33
MAXDISPATCHTHREADS 4-33
MINDISPATCHTHREADS 4-34
Specifying a Threading Model 4-35
Specifying the Number of Active Objects 4-35
Sample UBBCONFIG File 4-36

5. Security and CORBA Server Applications

Overview of Security and CORBA Server Applications 5-1

Design Considerations for the University Server Application 5-2
How the Security University Sample Application Works. 5-2
Design Considerations for Returning Student Details to the Client Application . . 5-5

6. Integrating Transactions into a CORBA Server Application

Overview of Transactions in the BEA Tuxedo System 6-2
Designing and Implementing Transactions in a CORBA Server Application 6-3
How the Transactions University Sample Application Works 6-5
Transactional Model Used by the Transactions University Sample Application . . 6-6
Object State Considerations for the University Server Application 6-7
Object Policies Defined for the Registrar Object 6-7
Object Policies Defined for the RegistrarFactory Object 6-7
Using an XA Resource Manager in the Transactions Sample Application . . . 6-8
Configuration Requirements for the Transactions Sample Application 6-8
Integrating Transactions in a CORBA Client and Server Application 6-9
Making an Object Automatically Transactional 6-10
Enabling an Object to Participate in a Transaction 6-10
Preventing an Object from Being Invoked While a Transaction Is Scoped 6-11
Excluding an Object from an Ongoing Transaction 6-12
Assigning Policies 6-12
Opening an XA Resource Manager 6-13
Closing an XA Resource Manager 6-13
Transactions and Object State Management 6-13
Delegating Object State Management to an XA Resource Manager 6-13
Waiting Until Transaction Work Is Complete Before Writing to the Database . . 6-14
Notes on Using Transactions in the BEA Tuxedo System. 6-15
User-defined Exceptions 6-17
Defining the Exception 6-17

Throwing the Exception 6-18

7. Wrapping a BEA Tuxedo Service in a CORBA Object

Overview of Wrapping a BEA Tuxedo Service 7-2
Designing the Object That Wraps the BEA Tuxedo Service. 7-3
Creating the Buffer in Which to Encapsulate BEA Tuxedo Service Calls 7-4
Implementing the Operations That Send Messages to and from the BEA Tuxedo Service 7-5
Restrictions 7-7
Design Considerations for the Wrapper Sample Application. 7-7
How the Wrapper University Sample Application Works 7-8
Interface Definitions for the Billing Server Application 7-9
Additional Design Considerations for the Wrapper Sample Application . . . 7-10

8. Scaling a BEA Tuxedo CORBA Server Application

Overview of the Scalability Features Available in the BEA Tuxedo System 8-2
Scaling a BEA Tuxedo Server Application 8-2
OMG IDL Changes for the Production Sample Application 8-3
Replicating Server Processes and Server Groups 8-4
Replicated Server Processes. 8-4
Replicated Server Groups. 8-6
Configuring Replicated Server Processes and Groups. 8-7
Scaling the Application Via Object State Management 8-9
Factory-based Routing. 8-11
How Factory-based Routing Works 8-12
Configuring for Factory-based Routing in the UBBCONFIG file. 8-12
Implementing Factory-based Routing in a Factory 8-15
What Happens at Run Time 8-16
Additional Design Considerations for the Registrar and Teller Objects 8-17

Instantiating the Registrar and Teller Objects 8-17
Ensuring That Student Registration Occurs in the Correct Server Group . . . 8-18
Ensuring That the Teller Object Is Instantiated in the Correct Server Group 8-20
How the Production Server Application Can Be Scaled Further 8-20
Choosing Between Stateless and Stateful Objects 8-21
When You Want Stateless Objects 8-22
When You Want Stateful Objects 8-23

CHAPTER 1

CORBA Server Application Concepts

This topic includes the following sections:

- The Entities You Create to Build a CORBA Server Application:
  - The Implementation of the CORBA Objects for Your Server Application
  - The Server Object
- Process for Developing CORBA Server Applications:
  - Generating Object References
  - Managing Object State
  - Reading and Writing an Object's Data
  - Using Design Patterns

Each of the chapters in this book gives procedures for and examples of building CORBA server applications that take advantage of various BEA Tuxedo software features. For background information about BEA Tuxedo CORBA server applications and how they work, see *Getting Started with BEA Tuxedo CORBA Applications*.

The Entities You Create to Build a CORBA Server Application

To build a CORBA server application, you create the following two entities:

- The implementation of the CORBA objects that execute your server application's business logic.
- The Server object, which implements the operations that initialize and release the server application and instantiate the CORBA objects needed to satisfy client requests.

There are also a number of files that you work with that are generated by the IDL compiler and that you build into a CORBA server application. These files are listed and described in Chapter 2, "Steps for Creating a BEA Tuxedo CORBA Server Application."

The sections that follow provide introductory information about these entities. For complete details about how to generate these components, see Chapter 2, "Steps for Creating a BEA Tuxedo CORBA Server Application."

The Implementation of the CORBA Objects for Your Server Application

Having a clear understanding of what CORBA objects are, and how they are defined, implemented, instantiated, and managed is critical for the person who is designing or creating a CORBA server application.

The CORBA objects for which you have defined interfaces in the Object Management Group Interface Definition Language (OMG IDL) contain the business logic and data for your CORBA server applications. All client application requests involve invoking an operation on a CORBA object. The code you write that implements the operations defined for an interface is called an object implementation. For example, in C++, the object implementation is a C++ class.

This topic includes the following sections:

- How OMG IDL interface definitions establish the operations that can be invoked on a CORBA object
- How you implement the operations on a CORBA object
- How client applications access and manipulate your application's CORBA objects
- How you instantiate a CORBA object with code and data at run time in response to a client request

How Interface Definitions Establish the Operations on a CORBA Object

A CORBA object's interface identifies the operations that can be performed on it. A distinguishing characteristic of CORBA objects is that an object's interface definition is separate

from its implementation. The definition for the interface establishes how the operations on the interface must be implemented, including what the valid parameters are that can be passed to and returned from an operation.

An interface definition, which is expressed in OMG IDL, establishes the client/server contract for an application. That is, for a given interface, the server application is bound to do the following:

- Implement the operations defined for that interface
- Always use the parameters defined with each operation

How the server application implements the operations may change over time. This is acceptable behavior as long as the server application continues to meet the requirement of implementing the defined interface and using the defined parameters. In this way, the client stub is always a reliable proxy for the object implementation on the server machine. This underscores one of the key architectural strengths of CORBA—that you can change how a server application implements an object over time without requiring the client application to be modified or even to be aware that the object implementation has changed.

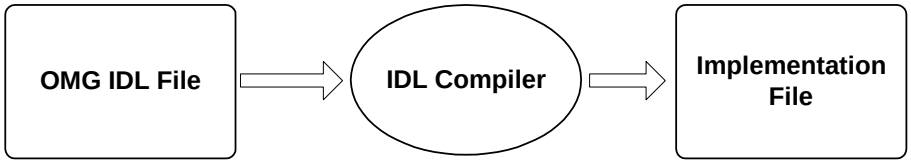
The interface definition also determines the content of both the client stub and the skeleton in the server application; these two entities, in combination with the ORB and the Portable Object Adapter (POA), ensure that a client request for an operation on an object can be routed to the code in the server application that can satisfy the request.

Once the system designer has specified the interfaces of the business objects in the application, the programmer's job is to implement those interfaces. This book explains how.

For more information about OMG IDL, see *Creating CORBA Client Applications*.

How You Implement the Operations on a CORBA Object

As stated earlier, the code that implements the operations defined for a CORBA object's interface is called an object implementation. For C++, this code consists of a set of methods, one for each of the operations defined for the interfaces in your application's OMG IDL file. The file containing the set of object implementations for your application is known as an implementation file. The BEA Tuxedo system provides an IDL compiler, which compiles your application's OMG IDL file to produce several files, one being an implementation file, shown in the following figure.



The generated implementation file contains method templates, method declarations, object constructors and destructors, and other data that you can use as a starting place for writing your application's object implementations. For example, in C++, the generated implementation file contains signatures for each interface's methods. You enter the business logic for each method in this file, and then provide this file as input to the command that builds the executable server application file.

How Client Applications Access and Manipulate Your Application's CORBA Objects

Client applications access and manipulate the CORBA objects managed by the server application via ***object references*** to those objects. Client applications invoke operations (that is, requests) on an object reference. These requests are sent as messages to the server application, which invokes the appropriate operations on CORBA objects. The fact that these requests are sent to the server application and invoked in the server application is completely transparent to the client; client applications appear simply to be making invocations on the client stub.

Client applications may manipulate a CORBA object only by means of an object reference. One primary design consideration is how to create object references and return them to the client applications that need them in a way that is appropriate for your application.

Typically, object references to CORBA objects are created in the BEA Tuxedo system by ***factories***. A factory is any CORBA object that returns, as one of its operations, a reference to another CORBA object. You implement your application's factories the same way that you implement other CORBA objects defined for your application. You can make your factories widely known to the BEA Tuxedo domain, and to clients connected to the BEA Tuxedo domain, by registering them with the FactoryFinder. Registering a factory is an operation typically performed by the Server object, which is described in the section "The Server Object" on page 1-7. For more information about designing factories, see the section "Generating Object References" on page 1-8.

The Content of an Object Reference

From the client application's perspective, an object reference is opaque; it is like a black box that client applications use without having to know what is inside. However, object references contain all the information needed for the BEA Tuxedo system to locate a specific object instance and to locate any state data that is associated with that object.

An object reference contains the following information:

- The interface name

  This is the Interface Repository ID of the object's OMG IDL interface.

- The object ID (OID)

  The OID uniquely identifies the instance of the object to which the reference applies. If the object has data in external storage, the OID also typically includes a key that the server machine can use to locate the object's data.

- Group ID

  The group ID identifies the server group to which the object reference is routed when a client application makes a request using that object reference. Generating a nondefault group ID is part of a key BEA Tuxedo feature called factory-based routing, which is described in the section "Factory-based Routing" on page 8-11.

Note: The combination of the three items in the preceding list uniquely identifies the CORBA object. It is possible for an object with a given interface and OID to be simultaneously active in two different groups, if those two groups both contain the same object implementation. If you need to guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, either: use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or configure your domain so that a given object implementation is in only one group. This assures that if multiple clients have an object reference containing a given interface name and OID, the reference is always routed to the same object instance.

For more information about factory-based routing, see the section "Factory-based Routing" on page -11.

The Lifetime of an Object Reference

Object references created by server applications running in a BEA Tuxedo domain typically have a usable lifespan that extends beyond the life of the server process that creates them. BEA Tuxedo object references can be used by client applications regardless of whether the server processes

that originally created them are still running. In this way, object references are not tied to a specific server process.

An object reference created with the `TP::create_active_object_reference()` operation is valid only for the lifetime of the server process in which it was created. For more information, see the section "Preactivating an Object with State" on page 3-15.

Passing Object Instances

The ORB cannot marshal an object instance as an object reference. For example, passing a factory reference in the following code fragment generates a CORBA marshal exception in the BEA Tuxedo system:

```
connection::setFactory(this);
```

To pass an object instance, you should create a proxy object reference and pass the proxy instead, as in the following example:

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

How You Instantiate a CORBA Object at Run Time

When a server application receives a request for an object that is not mapped in the server machine's memory (that is, the object is not active), the TP Framework invokes the `Server::create_servant()`operation. The `Server::create_servant()`operation is implemented in the Server object, which, as mentioned in the section "The Implementation of the CORBA Objects for Your Server Application" on page 1-2, is a component of a CORBA server application that you create.

The `Server::create_servant()`operation causes an instance of the CORBA object implementation to be mapped into the server machine's memory. This instance of the object's implementation is called the object's ***servant***. Formally speaking, a servant is an instance of the C++ class that implements an interface defined in the application's OMG IDL file. The servant is generated via the C++ `new` statement that you write in the `Server::create_servant()`operation.

After the object's servant has been created, the TP Framework invokes the `Tobj_ServantBase::activate_object()` operation on the servant. The `Tobj_ServantBase::activate_object()` operation is a virtual operation that is defined on the `Tobj_ServantBase` base class, from which all object implementation classes inherit. The TP Framework invokes the `Tobj_ServantBase::activate_object()` operation to tie the servant

to an object ID (OID). (Conversely, when the TP Framework invokes the `Tobj_ServantBase::deactivate_object()` operation on an object, the servant's association with the OID is broken.)

If your object has data on disk that you want to read into memory when the CORBA object is activated, you can have that data read by defining and implementing the `Tobj_ServantBase::activate_object()` operation on the object. The `Tobj_ServantBase::activate_object()` operation can contain the specific read operations required to bring an object's durable state into memory. (There are circumstances in which you may prefer instead to have an object's disk data read into memory by one or more separate operations on the object that you may have coded in the implementation file. For more information, see the section "Reading and Writing an Object's Data" on page 1-14.) After the invocation of the `Tobj_ServantBase::activate_object()` operation is complete, the object is said to be active.

This collection of the object's implementation and data compose the run-time, active instance of the CORBA object.

Servant Pooling

Servant pooling provides your CORBA server application the opportunity to keep a servant in memory after the servant's association with a specific OID has been broken. When a client request that can be satisfied with a pooled servant arrives, the TP Framework bypasses the `TP::create_servant` operation and creates a link between the pooled servant and the OID provided in the client request.

Servant pooling thus provides the CORBA server application with a means to minimize the costs of reinstantiating a servant each time a request arrives for an object that can be satisfied by that servant. For more information about servant pooling and how to use it, see the section "Servant Pooling" on page 2-25.

Note: Servant pooling was first introduced in release 4.2 of the WebLogic Enterprise product.

The Server Object

The Server object is the other programming code entity that you create for a CORBA server application. The Server object implements operations that execute the following tasks:

- Performing basic server application initialization operations, which may include registering factories managed by the server application and allocating resources needed by the server application. If the server application is transactional, the Server object also implements the code that opens an XA resource manager.

- Instantiating the CORBA objects needed to satisfy client requests.
- Performing server process shutdown and cleanup procedures when the server application has finished servicing requests. For example, if the server application is transactional, the Server object also implements the code that closes the XA resource manager.

You create the Server object from scratch, using a common text editor. You then provide the Server object as input into the server application build command, `buildobjserver`. For more information about creating the Server object, see Chapter 2, "Steps for Creating a BEA Tuxedo CORBA Server Application."

Process for Developing CORBA Server Applications

This section presents important background information about the following topics, which have a major influence on how you design and implement CORBA server applications:

- Generating Object References
- Managing Object State
- Reading and Writing an Object's Data
- Using Design Patterns

It is not essential that you read these topics before proceeding to the next chapter; however, this information is located here because it applies broadly to fundamental design and implementation issues for all CORBA server applications.

Generating Object References

One of the most basic functions of a CORBA server application is providing client applications with object references to the objects they need to execute their business logic. CORBA client applications typically get object references to the initial CORBA objects they use from the following two sources:

- The **Bootstrap object**
- **Factories** managed in the BEA Tuxedo domain

Client applications use the Bootstrap object to resolve initial references to a specific set of objects in the BEA Tuxedo domain, such as the FactoryFinder and the SecurityCurrent objects. The Bootstrap object is described in *Getting Started with BEA Tuxedo CORBA Applications* and *Creating CORBA Client Applications*.

Factories, however, are designed, implemented and registered by you, and they provide the means by which client applications get references to objects in the CORBA server application, particularly the initial server application object. At its simplest, a factory is a CORBA object that returns an object reference to another CORBA object. The client application typically invokes an operation on a factory to obtain an object reference to a CORBA object of a specific type. Planning and implementing your factories carefully is an important task when developing CORBA server applications.

How Client Applications Find Your Server Application's Factories

Client applications are able to locate via the FactoryFinder the factories managed by your server application. When you develop the Server object, you typically include code that registers with the FactoryFinder any factories managed by the server application. It is via this registration operation that the FactoryFinder keeps track of your server application's factories and can provide object references to them to the client applications that request them. We recommend that you use factories and register them with the FactoryFinder; this model makes it simple for client applications to find the objects in your CORBA server application.

Creating an Active Object Reference

An active object reference is a feature that gives an alternate means through which your server application can generate object references. Active object references are not typically created by factories as described in the previous section, and active object references are meant for preactivating objects with state. The next section discusses object state in more detail.

While an object associated with a conventional object reference is not instantiated until a client application makes an invocation on the object, the object associated with an active object reference is created and activated at the time the active object reference is created. Active object references are especially convenient for specific purposes, such as iterator objects. The section "Preactivating an Object with State" on page 3-15 provides more information about active object references.

Note: The active object reference feature was first introduced in WebLogic Enterprise version 4.2.

Managing Object State

Object state management is a fundamentally important concern of large-scale client/server systems, because it is critical that such systems optimize throughput and response time. The majority of high-throughput applications, such as applications you run in a BEA Tuxedo domain,

tend to be stateless, meaning that the system flushes state information from memory after a service or an operation has been fulfilled.

Managing state is an integral part of writing CORBA-based server applications. Typically, it is difficult to manage state in these server applications in a way that scales and performs well. The BEA Tuxedo software provides an easy way to manage state and simultaneously ensure scalability and high performance.

The scalability qualities that you can build into a CORBA server application help the server application function well in an environment that includes hundreds or thousands of client applications, multiple machines, replicated server processes, and a proportionately greater number of objects and client invocations on those objects.

About Object State

In a BEA Tuxedo domain, ***object state*** refers specifically to the process state of an object across client invocations on it. The BEA Tuxedo software uses the following definitions of stateless and stateful objects:

Object	Behavior Characteristics
Stateless	The object is mapped into memory only for the duration of an invocation on one of the object's operations, and is deactivated and has its process state flushed from memory after the invocation is complete; that is, the object's state is not maintained in memory after the invocation is complete.
Stateful	The object remains activated between invocations on it, and its state is maintained in memory across those invocations. The state remains in memory until a specific event occurs, such as: • The server process in which the object exists is stopped or is shut down • The transaction in which the object is participating is either committed or rolled back • The object invokes the `TP::deactivateEnable()` operation on itself. Each of these events is discussed in more detail in this section.

Both stateless and stateful objects have data; however, stateful objects may have nonpersistent data in memory that is required to maintain context (state) between operation invocations on those objects. Thus, subsequent invocations on such a stateful object always go to the same servant. Conversely, invocations on a stateless object can be directed by the BEA Tuxedo system to any available server process that can activate the object.

State management also involves how long an object remains active, which has important implications on server performance and the use of machine resources. The duration of an active object is determined by ***object activation policies*** that you assign to an object's interface, described in the section that follows.

Object state is transparent to the client application. Client applications implement a conversational model of interaction with distributed objects. As long as a client application has an object reference, it assumes that the object is always available for additional requests, and the object appears to be maintained continuously in memory for the duration of the client application interaction with it.

To achieve optimal application performance, you need to carefully plan how your application's objects manage state. Objects are required to save their state to durable storage, if applicable, before they are deactivated. Objects must also restore their state from durable storage, if applicable, when they are activated. For more information about reading and writing object state information, see the section "Reading and Writing an Object's Data" on page 1-14.

Note: BEA Tuxedo Release 8.0 or later provides support for parallel objects, as a performance enhancement. This feature allows you to designate all business objects in a particular application as stateless objects. For complete information, see Chapter 3, "TP Framework," in the *CORBA Programming Reference*.

Object Activation Policies

The BEA Tuxedo software provides three object activation policies that you can assign to an object's interface to determine how long an object remains in memory after it has been invoked by a client request. These policies determine whether the object to which they apply is generally stateless or stateful.

The three policies are listed and described in the following table.

Policy	Description
`Method`	Causes the object to be active only for the duration of the invocation on one of the object's operations; that is, the object is activated at the beginning of the invocation, and is deactivated at the end of the invocation. An object with this activation policy is called a *method-bound object.* The `method` activation policy is associated with stateless objects. This activation policy is the default.
`Transaction`	Causes the object to be activated when an operation is invoked on it. If the object is activated within the scope of a transaction, the object remains active until the transaction is either committed or rolled back. If the object is activated outside the scope of a transaction, its behavior is the same as that of a method-bound object. An object with this activation policy is called a *transaction-bound object.* For more information about object behavior within the scope of a transaction, and general guidelines about using this policy, see Chapter 6, "Integrating Transactions into a CORBA Server Application." The `transaction` activation policy is associated with stateful objects for a limited time and under specific circumstances.
`Process`	Causes the object to be activated when an operation is invoked on it, and to be deactivated only under the following circumstances: • The server process that manages this object is shut down. • An operation on this object invokes the `TP::deactivateEnable()` operation, which causes this object to be deactivated. (This is part of a key BEA Tuxedo feature called application-controlled deactivation, which is described in the section "Application-controlled Deactivation" on page -13. An object with this activation policy is called a *process-bound object.* The `process` activation policy is associated with stateful objects.

You determine what events cause an object to be deactivated by assigning object activation policies. For more information about how you assign object activation policies to an object's interface, see the section "Step 4: Define the In-memory Behavior of Objects" on page 2-12.

Application-controlled Deactivation

The BEA Tuxedo software also provides a feature called ***application-controlled deactivation***, which provides a means for an application to deactivate an object during run time. The BEA Tuxedo software provides the `TP::deactivateEnable()` operation, which a process-bound object can invoke on itself. When invoked, the `TP::deactivateEnable()` operation causes the object in which it exists to be deactivated upon completion of the current client invocation on that object. An object can invoke this operation only on itself; you cannot invoke this operation on any object but the object in which the invocation is made.

The application-controlled deactivation feature is particularly useful when you want an object to remain in memory for the duration of a limited number of client invocations on it, and you want the client application to be able to tell the object that the client is finished with the object. At this point, the object takes itself out of memory.

Application-controlled deactivation, therefore, allows an object to remain in memory in much the same way that a process-bound object can: the object is activated as a result of a client invocation on it, and it remains in memory after the initial client invocation on it is completed. You can then deactivate the object without having to shut down the server process in which the object exists.

An alternative to application-controlled deactivation is to scope a transaction to maintain a conversation between a client application and an object; however, transactions are inherently more costly, and transactions are generally inappropriate in situations where the duration of the transaction may be indefinite.

A good rule of thumb to use when choosing between application-controlled deactivation and transactions for a conversation is whether there are any disk writing operations involved. If the conversation involves read-only operations, or involves maintaining state only in memory, then application-controlled deactivation is appropriate. If the conversation involves writing data to disk during or at the end of the conversation, transactions may be more appropriate.

Note: If you use application-controlled deactivation to implement a conversational model between a client application and an object managed by the server application, make sure that the object eventually invokes the `TP::deactivateEnable()` operation. Otherwise, the object remains idle in memory indefinitely. (Note that this can be a risk if the client application crashes before the `TP::deactivateEnable()` operation is invoked. Transactions, on the other hand, implement a timeout mechanism to prevent the situation

in which the object remains idle for an indefinite period. This may be another consideration when choosing between the two conversational models.)

You implement application-controlled deactivation in an object using the following procedure:

1. In the implementation file, insert an invocation to the `TP::deactivateEnable()` operation at the appropriate location within the operation of the interface that uses application-controlled deactivation.
2. In the Implementation Configuration File (ICF file), assign the `process` activation policy to the interface that contains the operation that invokes the `TP::deactivateEnable()` operation.
3. Build and deploy your application as described in the sections "Step 5: Compile and Link the Server Application" on page 2-16 and "Step 6: Deploy the Server Application" on page 2-16.

Reading and Writing an Object's Data

Many of the CORBA objects managed by the server application may have data that is in external storage. This externally stored data may be regarded as the *persistent* or *durable* state of the object. You must address durable state handling at appropriate points in the object implementation for object state management to work correctly.

Because of the wide variety of requirements you may have for your client/server application with regards to reading and writing an object's durable state, the TP Framework cannot automatically handle durable object state on disk. In general, if an object's durable state is modified as a result of one or more client invocations, you must make sure that durable state is saved before the object is deactivated, and you should plan carefully how the object's data is stored or initialized while the object is active.

The sections that follow describe the mechanisms available to you to handle an object's durable state, and give some general advice how to read and write object state under specific circumstances. The specific topics presented include:

- The available mechanisms for reading and writing an object's durable state
- Reading state at object activation
- Reading state within individual operations on an object
- Stateless objects and durable state
- Stateful objects and durable state

- Your responsibilities for object deactivation
- Avoiding unnecessary I/O

How you choose to read and write durable state invariably depends on the specific requirements of your client/server application, especially with regard to how the data is structured. In general, your priority should be to minimize the number of disk operations, especially where a database controlled by an XA resource manager is involved.

Available Mechanisms for Reading and Writing an Object's Durable State

Table 1-1 and Table 1-2 describe the available mechanisms for reading and writing an object's durable state.

Table 1-1 Available Mechanisms for Reading an Object's Durable State

Mechanism	Description
`Tobj_ServantBase::activate_object()`	After the TP Framework creates the servant for an object, the TP Framework invokes the `Tobj_ServantBase::activate_object()` operation on that servant. As mentioned in the section "How You Instantiate a CORBA Object at Run Time" on page 1-6, this operation is defined on the `Tobj_ServantBase` base class, from which all the CORBA objects you define for your client/server application inherit. You may choose not to define and implement the `Tobj_ServantBase::activate_object()` operation on your object, in which case nothing happens regarding specific object state handling when the TP Framework activates your object. However, if you define and implement this operation, you can choose to include code in this operation that reads some or all of an object's durable state into memory. Therefore, the `Tobj_ServantBase::activate_object()` operation provides your server application with its first opportunity to read an object's durable state into memory. Note that if an object's OID contains a database key, the `Tobj_ServantBase::activate_object()` operation provides the only means the object has to extract that key from the OID. For more information about implementing the `Tobj_ServantBase::activate_object()` operation, see "Step 2: Write the Methods That Implement Each Interface's Operations" on page 2-5. For an example of implementing the `Tobj_ServantBase::activate_object()` operation, see Chapter 3, "Designing and Implementing a Basic CORBA Server Application."
Operations on the object	You can include inside the individual operations that you define on the object the code that reads an object's durable state.

Table 1-2 Available Mechanisms for Writing an Object's Durable State

Mechanism	Description
`Tobj_ServantBase::deactivate_object()`	When an object is being deactivated by the TP Framework, the TP Framework invokes this operation on the object as the final step of object deactivation. As with the `Tobj_ServantBase::activate_object()` operation, the `Tobj_ServantBase::deactivate_object()` operation is defined on the `Tobj_ServantBase` class. You implement the `deactivate_object()` operation on your object optionally if you have specific object state that you want flushed from memory or written to a database. The `Tobj_ServantBase::deactivate_object()` operation provides the final opportunity your server application has to write durable state to disk before the object is deactivated. If your object keeps any data in memory, or allocates memory for any purpose, you implement the `Tobj_ServantBase::deactivate_object()` operation so your object has a final opportunity to flush that data from memory. Flushing any state from memory before an object is deactivated is critical in avoiding memory leaks.
Operations on the object	As you may have individual operations on the objects that read durable state from disk, you may also have individual operations on the object that write durable state back to disk. For method-bound and process-bound objects in general, you typically perform database write operations within these operations and not in the `Tobj_ServantBase::deactivate_object()` operation. For transaction-bound objects, however, writing durable state in the `Tobj_ServantBase::deactivate_object()` operation provides a number of object management efficiencies that may make sense for your transactional server applications.

Note: If you use the `Tobj_ServantBase::deactivate_object()` operation to write any durable state to disk, any errors that occur while writing to disk are not reported to the client application. Therefore, the only circumstances under which you should write data

to disk in this operation is when: the object is transaction-bound (that is, it has the `transaction` activation policy assigned to it), or you scope the disk write operations within a transaction by invoking the TransactionCurrent object. Any errors encountered while writing to disk during a transaction can be reported back to the client application. For more information about using the `Tobj_ServantBase::deactivate_object()` operation to write object state to disk, see the section "Caveat for State Handling in Tobj_ServantBase::deactivate_object()" on page 2-24.

Reading State at Object Activation

Using the `Tobj_ServantBase::activate_object()` operation on an object to read durable state may be appropriate when either of the following conditions exist:

- Object data is always used or updated in all the object's operations.
- All the object's data is capable of being read in one operation.

The advantages of using the `Tobj_ServantBase::activate_object()` operation to read durable state include:

- You write code to read data only once, instead of duplicating the code in each of the operations that use that data.
- None of the operations that use an object's data need to perform any reading of that data. In this sense, you can write the operations in a way that is independent of state initialization.

Reading State Within Individual Operations on an Object

With all objects, regardless of activation policy, you can read durable state in each operation that needs that data. That is, you handle the reading of durable state outside the `Tobj_ServantBase::activate_object()`operation. Cases where this approach may be appropriate include the following:

- Object state is made up of discrete data elements that require multiple operations to read or write.
- Objects do not always use or update state data at object activation.

For example, consider an object that represents a customer's investment portfolio. The object contains several discrete records for each investment. If a given operation affects only one investment in the portfolio, it may be more efficient to allow that operation to read the one record than to have a general-purpose `Tobj_ServantBase::activate_object()` operation that automatically reads in the entire investment portfolio each time the object is invoked.

Stateless Objects and Durable State

In the case of stateless objects—that is, objects defined with the `method` activation policy—you must ensure the following:

- That any durable state needed by the request is brought into memory by the time the operation's business logic starts executing.
- That any changes to the durable state are written out by the end of the invocation.

The TP Framework invokes the `Tobj_ServantBase::activate_object()` operation on an object at activation. If an object has an OID that contains a key to the object's durable state on disk, the `Tobj_ServantBase::activate_object()` operation provides the only opportunity the object has to retrieve that key from the OID.

If you have a stateless object that you want to be able to participate in a transaction, we generally recommend that if the object writes any durable state to disk that it be done within individual methods on the object. However, if you have a stateless object that is always transactional—that is, a transaction is always scoped when this object is invoked—you have the option to handle the database write operations in the `Tobj_ServantBase::deactivate_object()` operation, because you have a reliable mechanism in the XA resource manager to commit or roll back database write operations accurately.

Note: Even if your object is method-bound, you may have to take into account the possibility that two server processes are accessing the same disk data at the same time. In this case, you may want to consider a concurrency management technique, the easiest of which is transactions. For more information about transactions and transactional objects, see Chapter 6, "Integrating Transactions into a CORBA Server Application."

Servant Pooling and Stateless Objects

Servant pooling is a particularly useful feature for stateless objects. When your CORBA server application pools servants, you can significantly reduce the costs of instantiating an object each time a client invokes it. As mentioned in the section "Servant Pooling" on page -7, a pooled servant remains in memory after a client invocation on it is complete. If you have an application in which a given object is likely to be invoked repeatedly, pooling the servant means that only the object's data, and not its methods, needs to be read into and out of memory for each client invocation. If the cost associated with reading an object's methods into memory is high, servant pooling can reduce that cost.

For information about how to implement servant pooling, see the section "Servant Pooling" on page 2-25.

Stateful Objects and Durable State

For stateful objects, you should read and write durable state only at the point where it is needed. This may introduce the following optimizations:

- In the case of process-bound objects, you avoid the situation in which an object allocates a large amount of memory over a long period.
- In the case of transaction-bound objects, you can postpone writing durable state until the `Tobj_ServantBase::deactivate_object()` operation is invoked, when the transaction outcome is known.

In general, transaction-bound objects must depend on the XA resource manager to handle all database write or rollback operations automatically.

Note: For objects that are involved in a transaction, we do not support having those objects write data to external storage that is not managed by an XA resource manager.

For more information about objects and transactions, see Chapter 6, "Integrating Transactions into a CORBA Server Application."

Servant Pooling and Stateful Objects

Servant pooling does not make sense in the case of process-bound objects; however, depending on your application design, servant pooling may provide a performance improvement for transaction-bound objects.

Your Responsibilities for Object Deactivation

As mentioned in the preceding sections, you implement the `Tobj_ServantBase::deactivate_object()` operation as a means to write an object's durable state to disk. You should also implement this operation on an object as a means to flush any remaining object data from memory so that the object's servant can be used to activate another instance of that object. You should not assume that an invocation to an object's `Tobj_ServantBase::deactivate_object()` operation also results in an invocation of that object's destructor.

Avoiding Unnecessary I/O

Be careful not to introduce inefficiencies into the application by doing unnecessary I/O in objects. Situations to be aware of include the following:

- If many operations in an object do not use or affect object state, it may be inefficient to read and write state each time these operations are invoked. Design these objects so that

they handle state only in the operations that need it; in such cases, you may not want to have all of the object's durable state read in at object activation.

- If object state is made up of data that is read in multiple operations, try to do only the necessary operations at object activation by doing one of the following:
  - Reading only the state that is common to all the operations in the `Tobj_ServantBase::activate_object()` operation. Defer the reading of additional state to only the operations that require it.
  - Writing out only the state that has changed. You can do this by managing flags that indicate the data that was changed during an activation, or by comparing before and after data images.

    A general optimization is to initialize a `dirtyState` flag on activation and to write data in the `Tobj_ServantBase::deactivate_object()` operation only if the flag has been changed while the object was active. (Note that this works only if you can be assured that the object is always activated in the same server process.)

Sample Activation Walkthrough

For examples of the sequence of activity that takes place when an object is activated, see *Getting Started with BEA Tuxedo CORBA Applications*.

Using Design Patterns

It is important to structure the business logic of your application around a well-formed design. The BEA Tuxedo software provides a set of design patterns to address this need. A design pattern is simply a structured solution to a specific design problem. The value of a design pattern lies in its ability to be expressed in a form you can reuse and apply to other design problems.

The BEA Tuxedo design patterns are structured solutions to enterprise-class application design problems. You can use them to design successful large-scale client/server applications.

The design patterns summarized here are a guide to using good design practices in CORBA client and server applications. They are an important and integral part of designing CORBA client and server applications, and the chapters in this book show examples of using these design patterns to implement the University sample applications.

Process-Entity Design Pattern

The Process-Entity design pattern applies to a large segment of enterprise-class client/server applications. This design pattern is referred to as the flyweight pattern in *Object-Oriented Design Patterns*, Gamma et al., and as the Model-View-Controller in other publications.

In this pattern, the client application creates a long-lived process object that the client application interacts with to make requests. For example, in the University sample applications, this object might be the registrar that handles course browsing operations on behalf of the client application. The courses themselves are database entities and are not made visible to the client application.

The advantages of the Process-Entity design pattern include:

- You can achieve the advantages of a fine-grained object model without implementing fine-grained objects. Instead, you use CORBA `struct` datatypes to simulate objects.
- Machine resource usage is optimized because there is only a single object mapped into memory: the process object. By contrast, if each database entity were activated into memory as a separate object instance, the number of objects that would need to be handled could overwhelm the machine's resources quickly in a large-scale deployment.
- Because they are not exposed to the client application, database entities need not be implemented as CORBA objects. Instead, entities can be implemented as local language objects in the server process. This is a fundamental principle of three-tier designs, but it also accurately models the way in which many businesses operate (for example, a registrar at a real university). The individual who serves as the registrar at a university can handle a large course database for multiple students; you do not need an individual registrar for each individual student. Thus, the process object state is distinct from the entity object state.

An example of applying the Process-Entity design pattern is described in Chapter 3, "Designing and Implementing a Basic CORBA Server Application." For complete details on the Process-Entity design pattern, see *Technical Articles*.

List-Enumerator Design Pattern

The List-Enumerator design pattern also applies to a large segment of enterprise-class client/server applications. The List-Enumerator design pattern leverages a key BEA Tuxedo feature, application-controlled object deactivation, to handle a cache of data that is stored and tracked in memory during several client invocations, and then to flush the data from memory when the data is no longer needed.

An example of applying the List-Enumerator design pattern is described in Chapter 3, "Designing and Implementing a Basic CORBA Server Application."

Object preactivation, which is an especially useful tool for implementing the List-Enumerator design, is described in the section "Preactivating an Object with State" on page 3-15.

CHAPTER 2

Steps for Creating a BEA Tuxedo CORBA Server Application

This chapter describes the basic steps involved in creating a CORBA server application. The steps shown in this chapter are not definitive; there may be other steps you may need to take for your particular server application, and you may want to change the order in which you follow some of these steps. However, the development process for every CORBA server application has each of these steps in common.

This topic includes the following sections:

- Summary of the CORBA Server Application Development Process
- Development and Debugging Tips
- Servant Pooling
- Delegation-based Interface Implementation

This chapter begins with a summary of the steps, and also lists the development tools and commands used throughout this book. Your particular deployment environment might use additional software development tools, so the tools and commands listed and described in this chapter are also not definitive.

The chapter uses examples from the Basic University sample application, which is provided with the BEA Tuxedo software. For complete details about the Basic University sample application, see the *Guide to the CORBA University Sample Applications*. For complete information about the tools and commands used throughout this book, see the *CORBA Programming Reference*.

For information about creating multithreaded CORBA server applications, see Chapter 4, "Creating Multithreaded CORBA Server Applications."

Summary of the CORBA Server Application Development Process

The basic steps to create a server application are:

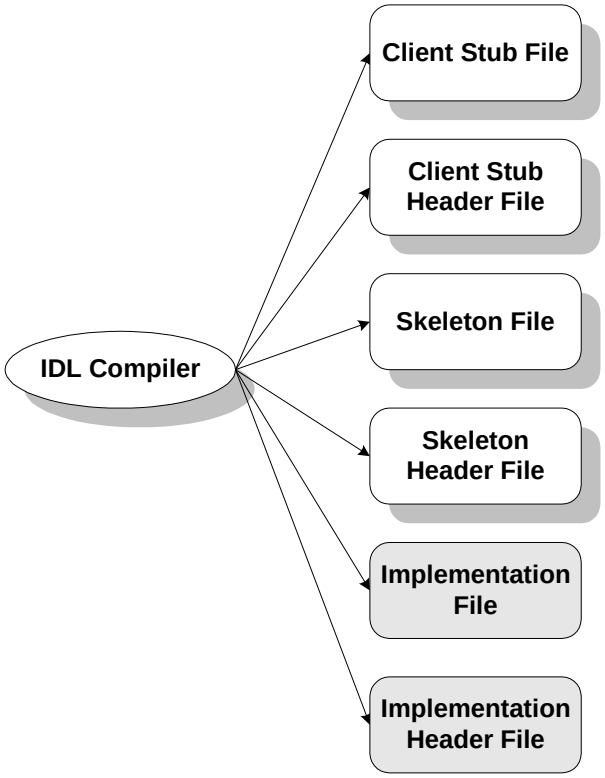
Step 1: Compile the OMG IDL File for the Server Application
Step 2: Write the Methods That Implement Each Interface's Operations
Step 3: Create the Server Object
Step 4: Define the In-memory Behavior of Objects
Step 5: Compile and Link the Server Application
Step 6: Deploy the Server Application

The BEA Tuxedo software also provides the following development tools and commands:

Tool	Description
IDL compiler	Compiles your application's OMG IDL file.
`genicf`	Generates an Implementation Configuration File (ICF file), which you can revise to specify nondefault object activation and transaction policies.
`buildobjserver`	Creates the executable image of your CORBA server application.
`tmloadcf`	Creates the `TUXCONFIG` file, a binary file for the CORBA domain that specifies the configuration of your server application.
`tmadmin`	Among other things, creates a log of transactional activities, which is used in some of the sample applications.

Step 1: Compile the OMG IDL File for the Server Application

The basic structure of the client and server portions of the application that runs in the BEA Tuxedo domain are determined by statements in the application's OMG IDL file. When you compile your application's OMG IDL file, the IDL compiler generates some or all of the files shown in the following diagram, depending upon which options you specify in the `idl` command. The shaded components are the generated files that you modify to create a server application.



The files produced by the IDL compiler are described in Table 2-1

Table 2-1 Files Produced by the IDL Compiler

File	Default Name	Description
Client stub file	*application*_c.cpp	Contains generated code for sending a request.
Client stub header file	*application*_c.h	Contains class definitions for each interface and type specified in the OMG IDL file.
Skeleton file	*application*_s.cpp	Contains skeletons for each interface specified in the OMG IDL file. The skeleton maps client requests to the appropriate operation in the server application during run time.
Skeleton header file	*application*_s.h	Contains the skeleton class definitions.

Table 2-1 Files Produced by the IDL Compiler

File	Default Name	Description
Implementation file	*application*_i.cpp	Contains signatures for the methods that implement the operations on the interfaces specified in the OMG IDL file.
Implementation header file	*application*_i.h	Contains the initial class definitions for each interface specified in the OMG IDL file.

Using the IDL Compiler

To generate the files listed in Table 2-1, enter the following command:

```
idl [options] idl-filename [icf-filename]
```

In the `idl` command syntax:

- `options` represents one or more command-line options to the IDL compiler. The command-line options are described in the *CORBA Programming Reference*. If you want to generate implementation files, you need to specify the `-i` option.
- *`idl-filename`* represents the name of your application's OMG IDL file.
- *`icf-filename`* is an optional parameter that represents the name of your application's Implementation Configuration File (ICF file), which you use to specify object activation policies or to limit the number of interfaces for which you want skeleton and implementation files generated. Using the ICF file is described in the section "Step 4: Define the In-memory Behavior of Objects" on page 2-14.

Note: The C++ IDL compiler implementation of pragmas changed in WebLogic Enterprise 5.1 to support CORBA 2.3 functionality and may affect your IDL files. The CORBA 2.3 functionality changes the scope that the pragma prefix definitions can affect. Pragmas do not affect definitions contained within included IDL files, nor do pragma prefix definitions made within included IDL files affect objects outside the included file.

The C++ IDL compiler has been modified to correct the handling of pragma prefixes. This change can effect the repository ID of objects, resulting in failures for some operations, such as a `_narrow`.

To prevent such failures:

- If you reload your IDL into the repository, you must also regenerate the client stubs and server skeletons of the application.

- If you regenerate any client stub or server skeleton, you must regenerate all stubs and skeletons of the application, and you must reload the IDL into the Interface Repository.

For more information about the IDL compiler, including details on the `idl` command, see the *CORBA Programming Reference*.

Generating the Skeleton and Implementation Files

The following command line generates client stub, skeleton, and initial implementation files, along with skeleton and implementation header files, for the OMG IDL file `univb.idl`:

```
idl -i univb.idl
```

For more information about the `idl` command, see the *CORBA Programming Reference*. For more information about generating these files for the BEA Tuxedo University sample applications, see the *Guide to the CORBA University Sample Applications*.

Note: If you plan to specify nondefault object activation or transaction policies, or if you plan to limit the number of interfaces for which you want skeleton and implementation files generated, you need to generate and modify an Implementation Configuration File (ICF) and pass the ICF file to the IDL compiler. For more information, see "Specifying Object Activation and Transaction Policies in the ICF File" on page 2-14.

Generating Tie Classes

The IDL compiler also provides the `-T` command-line option, which you can use for generating tie class templates for your interfaces. For more information about implementing tie classes in a CORBA application, see the section "Delegation-based Interface Implementation" on page 2-29.

Step 2: Write the Methods That Implement Each Interface's Operations

As the server application programmer, your task is to write the methods that implement the operations for each interface you have defined in your application's OMG IDL file.

The implementation file contains:

- Method declarations for each operation specified in the OMG IDL file.
- Your application's business logic, include files, and other data you want the application to use.

- Constructors and destructors for each interface implementation (implementing these is optional).
- Optionally, the `Tobj_ServantBase::activate_object()` and `Tobj_ServantBase::deactivate_object()` operations.

  Within the `Tobj_ServantBase::activate_object()` and `Tobj_ServantBase::deactivate_object()`operations, you write code that performs any particular steps related to activating or deactivating an object. This includes reading and writing the object's durable state from and to disk, respectively. If you implement these operations in your object, you must also edit the implementation header file and add the definitions for these operations in each implementation that uses them.

The Implementation File Generated by the IDL Compiler

Although you can create your server application's implementation file entirely by hand, the IDL compiler generates an implementation file that you can use as a starting place for writing your implementation file. The implementation file generated by the IDL compiler contains signatures for the methods that implement each of the operations defined for your application's interfaces.

You typically generate this implementation file only once, using the `-i` option with the command that invokes the IDL compiler. As you iteratively refine your application's interfaces, and modify the operations for those interfaces, including operation signatures, you add all the required changes to the implementation file to reflect those changes.

Implementing a Factory

As mentioned in the section "How Client Applications Access and Manipulate Your Application's CORBA Objects" on page 1-4, you need to create factories so that client applications can easily locate the objects managed by your server application. A factory is like any other CORBA object that you implement, with the exception that you register it with the FactoryFinder object. Registering a factory is described in the section "Writing the Code That Creates and Registers a Factory" on page 2-10.

The primary function of a factory is to create object references, which it does by invoking the `TP::create_object_reference()` operation. The `TP::create_object_reference()` operation requires the following input parameters:

- The Interface Repository ID of the object's OMG IDL interface
- The object ID (OID) in string format

- Optionally, routing criteria

For example, in the Basic University sample application, the `RegistrarFactory` interface specifies only one operation, as follows:

```
University::Registrar_ptr RegistrarFactory_i::find_registrar()
```

The `find_registrar()` operation on the `RegistrarFactory` object contains the following invocation to the `TP::create_object_reference()` operation to create a reference to a `Registrar` object:

```
CORBA::Object_var v_reg_oref =
            TP::create_object_reference(
                University::_tc_Registrar->id(),
                object_id,
                CORBA::NVlist::_nil()
            );
```

In the previous code example, notice the following:

- The following parameter specifies the `Registrar` object's Interface Repository ID by extracting it from its typecode:

  ```
  University::_tc_Registrar->id()
  ```

- The following parameter specifies that no routing criteria are used, with the result that an object reference created for the `Registrar` object is routed to the same group as the `RegistrarFactory` object that created the object reference:

  ```
  CORBA::NVlist::_nil()
  ```

  For information about specifying routing criteria that affect the group to which object references are routed, see Chapter 8, "Scaling a BEA Tuxedo CORBA Server Application."

Step 3: Create the Server Object

Implementing the Server object is not like implementing other language objects. The header class for the Server object has already been created, and the Server object class has already been instantiated for you. Creating the Server object involves implementing a specific set of methods in the prepackaged Server object class. The methods you implement are described in this section.

To create the Server object, create a new file using a common text editor and implement the following operations:

Operation	Description
`Server::initialize();`	After the server application is booted, the TP Framework invokes this operation as the last step in the server application initialization process. Within this operation, you perform a number of initialization tasks needed for your particular server application. What you provide within this operation is described in the section "Initializing the Server Application" on page 2-9.
`Server::create_servant();`	When a client request arrives that cannot be serviced by an existing servant, the TP Framework invokes this operation, passing the Interface Repository ID of the OMG IDL interface for the CORBA object to be activated. What you provide within this operation is described in the section "Creating Servants" on page 2-11.
`Server::release();`	The TP Framework invokes this operation when the server application is being shut down. This operation includes code to unregister any object factories managed by the server application and to perform other shutdown tasks. What you provide within this operation is described in the section "Releasing the Server Application" on page 2-13.

There is only one instance of the Server object in any server application. If your server application is managing multiple CORBA object implementations, the `Server::initialize()`, `Server::create_servant()`, and `Server::release()` operations you write must include code that applies to all those implementations.

The code that you write for most of these tasks involves interaction with the TP Framework. The sections that follow explain the code required for each of these Server object operations and shows sample code from the Basic University sample application.

Initializing the Server Application

The first operation that you implement in your Server object is the operation that initializes the server application. This operation is invoked when the BEA Tuxedo system starts the server application. The TP Framework invokes the following operation in the Server object during the startup sequence of the server application:

```
CORBA::Boolean Server::initialize(int argc, char** argv)
```

Any command-line options specified in the `CLOPT` parameter for your specific server application in the `SERVERS` section of the BEA Tuxedo domain's `UBBCONFIG` file are passed to the

`Server::initialize()` operation as `argc` and `argv`. For more information about passing arguments to the server application, see *Administering a BEA Tuxedo Application at Run Time*. For examples of passing arguments to the server application, see the *Guide to the CORBA University Sample Applications*.

Within the `Server::initialize()` operation, you include code that does the following, if applicable:

- Creates and registers factories
- Allocates any machine resources
- Initializes any global variables needed by the server application
- Opens the databases used by the server application
- Opens the XA resource manager

Writing the Code That Creates and Registers a Factory

If your server application manages a factory that you want client applications to be able to locate easily, you need to write the code that registers that factory with the FactoryFinder object, which is invoked typically as the final step of the server application initialization process.

To write the code that registers a factory managed by your server application, you do the following:

1. Create an object reference to the factory.

   This step involves creating an object reference as described in the section "Implementing a Factory" on page 2-7. In this step, you include an invocation to the `TP::create_object_reference()` operation, specifying the Interface Repository ID of the factory's OMG IDL interface.The following example creates an object reference, represented by the variable `s_v_fact_ref`, to the `RegistrarFactory` factory:

   ```
   University::RegistrarFactory s_v_fact_ref =
        TP::create_object_reference(
            University::_tc_RegistrarFactory->id(),
            object_id,
            CORBA::NVList::_nil()
        );
   ```

2. Register the factory with the BEA Tuxedo domain.

   This step involves invoking the following operation for each of the factories managed by the server application:

```
TP::register_factory (CORBA::Object_ptr factory_or,
                      const char* factory_id);
```

The `TP::register_factory()` operation registers the server application's factories with the FactoryFinder object. This operation requires the following input parameters:

- The object reference for the factory, created in step 1 above.
- A string identifier, based on the factory object's interface typecode, used to identify the Interface Repository ID of the factory's OMG IDL interface.

The following example registers the `RegistrarFactory` factory with the BEA Tuxedo domain:

```
TP::register_factory(s_v_fact_ref.in(),
                     University::_tc_RegistrarFactory->id());
```

Notice the parameter `University::_tc_RegistrarFactory->id()`. This is the same parameter specified in the `TP::create_object_reference()` operation. This parameter extracts the Interface Repository ID of the object's OMG IDL interface from its typecode.

Creating Servants

After the server application initialization process is complete, the server application is ready to begin processing client requests. If a request arrives for an operation on a CORBA object for which there is no servant available in memory, the TP Framework invokes the following operation in the Server object:

```
Tobj_Servant Server::create_servant(const char* interfaceName)
```

The `Server::create_servant()` operation contains code that instantiates a servant for the object required by the client request. For example, in C++, this code includes a `new` statement on the interface class for the object.

The `Server::create_servant()` operation does not associate the servant with an OID. The association of a servant with an OID takes place when the TP Framework invokes the `Tobj_ServantBase::activate_object()` operation on the servant, which completes the instantiation of the object. (You cannot associate an OID with an object in the object's constructor.) Likewise, the disassociation of a servant with an OID takes place when the TP Framework invokes the `deactivate_object()` operation on the servant.

This behavior of a servant in the BEA Tuxedo system makes it possible, after an object has been deactivated, for the TP Framework to make a servant available for another object instantiation. Therefore, do not assume that an invocation of an object's `Tobj_ServantBase::deactivate_object()` operation results in an invocation of that

object's destructor. If you use the servant pooling feature in your server application, you can implement the `TP::application_responsibility()` operation in an object's `Tobj_ServantBase::deactivate_object()` operation to pass a pointer to the servant to a servant pool for later reuse. Servant pooling is discussed in the section "Servant Pooling" on page 2-28.

The `Server::create_servant()` operation requires a single input argument. The argument specifies a character string containing the Interface Repository ID of the OMG IDL interface of the object for which you are creating a servant.

In the code you write for this operation, you specify the Interface Repository IDs of the OMG IDL interfaces for the objects managed by your server application. During run time, the `Server::create_servant()`operation returns the servant needed for the object specified by the request.

The following code implements the `Server::create_servant()` operation in the University server application from the Basic University sample application:

```
Tobj_Servant Server::create_servant(const char* intf_repos_id)
{
  if (!strcmp(intf_repos_id, University::_tc_RegistrarFactory->id())) {
      return new RegistrarFactory_i();
  }
  if (!strcmp(intf_repos_id, University::_tc_Registrar->id())) {
      return new Registrar_i();
  }
  if (!strcmp(intf_repos_id, University::_tc_CourseSynopsisEnumerator->id())) {
      return new CourseSynopsisEnumerator_i();
  }
  return 0; // unknown interface
}
```

Releasing the Server Application

When the BEA Tuxedo system administrator enters the `tmshutdown` command, the TP Framework invokes the following operation in the Server object of each running server application in the BEA Tuxedo domain:

```
void Server::release()
```

Within the `Server::release()` operation, you may perform any application-specific cleanup tasks that are specific to the server application, such as:

- Unregistering object factories managed by the server application

- Deallocating resources
- Closing any databases
- Closing an XA resource manager

Once a server application receives a request to shut down, the server application can no longer receive requests from other remote objects. This has implications on the order in which server applications should be shut down, which is an administrative task. For example, do not shut down one server process if a second server process contains an invocation in its `Server::release()` operation to the first server process.

During server shutdown, you may want to include the following invocation to unregister each of the server application's factories:

```
TP::unregister_factory (CORBA::Object_ptr factory_or,
                        const char* factory_id)
```

The invocation of the `TP::unregister_factory()` operation should be one of the first actions in the `Server::release()` implementation. The TP::`unregister_factory()` operation unregisters the server application's factories. This operation requires the following input arguments:

- The object reference for the factory.
- A string identifier, based on the factory object's interface typecode, used to identify Interface Repository ID of the object's OMG IDL interface.

The following example unregisters the `RegistrarFactory` factory used in the Basic sample application:

```
TP::unregister_factory(s_v_fact_ref.in(), UnivB::_tc_RegistrarFactory->id());
```

In the preceding code example, notice the use of the global variable `s_v_fact_ref`. This variable was set in the `Server::initialize()` operation that registered the `RegistrarFactory` object, which is used again here.

Notice also the parameter `UnivB::_tc_RegistrarFactory->id()`. This is also the same as the interface name used to register the factory.

Step 4: Define the In-memory Behavior of Objects

As stated in the section "Managing Object State" on page 1-11, you determine what events cause an object to be deactivated by assigning object activation policies, transaction policies, and, optionally, using the application-controlled deactivation feature.

You specify object activation and transaction policies in the ICF file, and you implement application-controlled deactivation via the `TP::deactivateEnable()` operation. This section explains how you implement both mechanisms, using the Basic University sample application as an example.

The sections that follow describe the following:

- How to specify object activation and transaction policies in the ICF file
- How to implement application-controlled deactivation

Specifying Object Activation and Transaction Policies in the ICF File

The BEA Tuxedo software supports the following activation policies, described in "Object Activation Policies" on page 1-11:

Activation Policy	Description
`method`	Causes the object to be active only for the duration of the invocation on one of the object's operations.
`transaction`	Causes the object to be activated when an operation is invoked on it. If the object is activated within the scope of a transaction, the object remains active until the transaction is either committed or rolled back.
`process`	Causes the object to be activated when an operation is invoked on it, and to be deactivated only when one of the following occurs: • The process in which the server application exists is shut down. • The object has invoked the `TP::deactivateEnable()` operation on itself.

The BEA Tuxedo software also supports the following transaction policies, described in Chapter 6, "Integrating Transactions into a CORBA Server Application":

Transaction Policy	Description
`always`	When an operation on this object is invoked, this policy causes the TP Framework to begin a transaction for this object, if there is not already an active transaction. If the TP Framework starts the transaction, the TP Framework commits the transaction if the operation completes successfully, or rolls back the transaction if the operation raises an exception.
`optional`	When an operation on this object is invoked, this policy causes the TP Framework to include this object in a transaction if a transaction is active. If no transaction is active, the invocation on this object proceeds according to the activation policy defined for this object. This is the default transaction policy.
`never`	Causes the TP Framework to generate an error condition if this object is invoked during a transaction.
`ignore`	If a transaction is currently active when an operation on this object is invoked, the transaction is suspended until the operation invocation is complete. This transaction policy prevents any transaction from being propagated to the object to which this transaction policy has been assigned.

To assign these policies to the objects in your application:

1. Generate the ICF file by entering the `genicf` command, specifying your application's OMG IDL file as input, as in the following example:

   ```
   # genicf university.idl
   ```

   The preceding command generates the file `university.icf`.

2. Edit the ICF file and specify the activation policies for each of your application's interfaces. The following example shows the ICF file generated for the Basic University sample application. Notice that the default object activation policy is `method`, and that the default transaction activation policy is `optional`.

```
module POA_UniversityB
    {
    implementation CourseSynopsisEnumerator_i
        {
```

```
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::CourseSynopsisEnumerator );
        };
    };
module POA_UniversityB
    {
    implementation Registrar_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::Registrar );
        };
    };
module POA_UniversityB
    {
    implementation RegistrarFactory_i
        {
        activation_policy ( method );
        transaction_policy ( optional );
        implements ( UniversityB::RegistrarFactory );
        };
    };
```

3. If you want to limit the number of interfaces for which you want skeleton and implementation files generated, you can remove from the ICF file the implementation blocks that implement those interfaces. Using the preceding ICF code as an example, to prevent skeleton and implementation files from being generated for the `RegistrarFactory` interface, remove the following lines:

   ```
   implementation RegistrarFactory_i
           {
           activation_policy ( method );
           transaction_policy ( optional );
           implements ( UniversityB::RegistrarFactory );
           };
   ```

4. Pass the ICF file to the IDL compiler to generate the skeleton and implementation files that correspond to the specified policies. For more information, see the section “Generating the Skeleton and Implementation Files” on page 2-5.

Step 5: Compile and Link the Server Application

After you have finished writing the code for the Server object and the object implementations, you compile and link the server application.

You use the `buildobjserver` command to compile and link CORBA server applications. The `buildobjserver` command has the following format:

```
buildobjserver [-o servername] [options]
```

In the `buildobjserver` command syntax:

- `-o` *`servername`* represents the name of the server application to be generated by this command.
- `options` represents the command-line options to the `buildobjserver` command.

For complete information about compiling and linking the University sample applications, see the *Guide to the CORBA University Sample Applications*. For complete details about the `buildobjserver` command, see the *BEA Tuxedo Command Reference.*

There are special considerations for designing and building multithreaded CORBA server applications. See “Using the buildobjserver Command” on page 4-13.

Note: If you are running the BEA Tuxedo software on IBM AIX 4.3.3 systems, you need to recompile your CORBA applications using the `-brtl` compiler option.

Step 6: Deploy the Server Application

You or the system administrator deploy the CORBA server application by using the procedure summarized in this section. For complete details on building and deploying the University sample applications, see the *Guide to the CORBA University Sample Applications*.

To deploy the server application:

1. Place the server application executable file in an appropriate directory on a machine that is part of the intended BEA Tuxedo domain.
2. Create the application’s configuration file, also known as the `UBBCONFIG` file, in a common text editor.
3. Set the following environment variables on the machine from which you are booting the CORBA server application:

  - TUXCONFIG, which needs to match exactly the TUXCONFIG entry in the UBBCONFIG file. This variable represents the location or path of the application's UBBCONFIG file.
  - APPDIR, which represents the directory in which the application's executable file exists.

4. Set the TUXDIR environment variable on all machines that are running in the BEA Tuxedo domain or that are connected to the BEA Tuxedo domain. This environment variable points to the location where the BEA Tuxedo software is installed.
5. Enter the following command to create the TUXCONFIG file:

   ```
   tmloadcf -y application-ubbconfig-file
   ```

   The command-line argument *application*-ubbconfig-*file* represents the name of your application's UBBCONFIG file. Note that you may need to remove any old TUXCONFIG files to execute this command.
6. Enter the following command to start the CORBA server application:

   ```
   tmboot -y
   ```

   You can reboot a server application without reloading the UBBCONFIG file.

For complete details about configuring the University sample applications, see the *Guide to the CORBA University Sample Applications*. For complete details on creating the UBBCONFIG file for CORBA applications, see *Setting Up a BEA Tuxedo Application*.

Development and Debugging Tips

This topic includes the following sections:

- Use of CORBA exceptions and the user log
- Detecting error conditions in the callback methods
- Common pitfalls of OMG IDL interface versioning and modification
- Caveat for state handling in the Tobj_ServantBase::deactivate_object() operation

Use of CORBA Exceptions and the User Log

This topic includes the following sections:

- The client application view of exceptions
- The server application view of exceptions

Client Application View of Exceptions

When a client application invokes an operation on a CORBA object, an exception may be returned as a result of the invocation. The only valid exceptions that can be returned to a client application are the following:

- Standard CORBA-defined exceptions that are known to every CORBA-compliant ORB
- Exceptions that are defined in OMG IDL and known to the client application via either its stub or the Interface Repository

The BEA Tuxedo system works to ensure that these CORBA-defined restrictions are not violated, which is described in the section "Server Application View of Exceptions" on page 2-20.

Because the set of exceptions exposed to the client application is limited, client applications may occasionally catch exceptions for which the cause is ambiguous. Whenever possible, the BEA Tuxedo system supplements such exceptions with descriptive messages in the user log, which serves as an aid in detecting and debugging error conditions. These cases are described in the following section.

Server Application View of Exceptions

This topic includes the following sections:

- Exceptions raised by the BEA Tuxedo system that can be caught by application code
- The BEA Tuxedo system's handling of exceptions raised by application code during the invocation of operations on CORBA objects

Exceptions Raised by the BEA Tuxedo System That Can Be Caught by Application Code

The BEA Tuxedo system may return the following types of exceptions to an application when operations on the TP object are invoked:

- CORBA-defined system exceptions
- CORBA `UserExceptions` defined in the file `TobjS_c.h`. The OMG IDL for the exceptions defined in this file is the following:

```
interface TobjS {
          exception AlreadyRegistered { };
          exception ActivateObjectFailed { string reason; };
          exception ApplicationProblem { };
          exception CannotProceed { };
          exception CreateServantFailed { string reason; };
          exception DeactivateObjectFailed { string reason; };
```

```
            exception IllegalInterface { };
            exception IllegalOperation { };
            exception InitializeFailed { string reason; };
            exception InvalidDomain { };
            exception InvalidInterface { };
            exception InvalidName { };
            exception InvalidObject { };
            exception InvalidObjectId { };
            exception InvalidServant { };
            exception NilObject { string reason; };
            exception NoSuchElement { };
            exception NotFound { };
            exception OrbProblem { };
            exception OutOfMemory { };
            exception OverFlow { };
            exception RegistrarNotAvailable { };
            exception ReleaseFailed { string reason; };
            exception TpfProblem { };
            exception UnknownInterface { };
}
```

The BEA Tuxedo System's Handling of Exceptions Raised by Application Code During the Invocation of Operations on CORBA Objects

A server application can raise exceptions in the following places in the course of servicing a client invocation:

- In the `Server::create_servant`, `Tobj_ServantBase::activate_object()`, and `Tobj_ServantBase::deactivate_object()` callback methods.
- In the implementation code for the invoked operation.

It is possible for the server application to raise any of the following types of exceptions:

- A CORBA-defined system exception.
- A CORBA user-defined exception defined in OMG IDL
- A CORBA user-defined exception defined in the file `TobjS_c.h`. The following exceptions are intended to be used in server applications to help the BEA Tuxedo system send messages to the user log, which can help with troubleshooting:

  ```
  interface TobjS {
            exception ActivateObjectFailed { string reason; };
            exception CreateServantFailed { string reason; };
            exception DeactivateObjectFailed { string reason; };
            exception InitializeFailed { string reason; };
  ```

```
            exception ReleaseFailed { string reason; };
};
```

- Any other C++ exception type

All exceptions raised by server application code that are not caught by the server application are caught by the BEA Tuxedo system. When these exceptions are caught, one of the following occurs:

- The exception is returned to the client application without alteration.
- The exception is converted to a standard CORBA exception, which is then returned to the client application.
- The exception is converted to a standard CORBA exception, and the following actions occur:
  - The exception is returned to the client application
  - One or more messages containing descriptive information about the error are sent to the user log. The descriptive information may originate from either the server application code or from the BEA Tuxedo system.

The following sections show how the BEA Tuxedo system handles exceptions raised by the server application during the course of a client invocation on a CORBA object.

Exceptions Raised in the `Server::create_servant()` Operation

If any exception is raised in the `Server::create_servant()` operation, then:

- The `CORBA::OBJECT_NOT_EXIST` exception is returned to the client application.
- If the exception raised is `TobjS::CreateServantFailed`, then a message is sent to the user log. If a reason string is supplied in the constructor for the exception, then the reason string is also written as part of the message.
- Neither the `Tobj_ServantBase::activate_object()` or `Tobj_ServantBase::deactivate_object()` operations are invoked. The operation requested by the client is not invoked.

Exceptions Raised in the `Tobj_ServantBase::activate_object()` Operation

If any exception is raised in the `Tobj_ServantBase::activate_object()` operation, then:

- The `CORBA::OBJECT_NOT_EXIST` exception is returned to the client application.

- If the exception raised is `TobjS::ActivateObjectFailed`, a message is sent to the user log. If a reason string is supplied in the constructor for the exception, the reason string is also written as part of the message.
- Neither the operation requested by the client nor the `Tobj_ServantBase::deactivate_object()` operation is invoked.

Exceptions Raised in Operation Implementations

The BEA Tuxedo system requires operation implementations to throw either CORBA system exceptions, or user-defined exceptions defined in OMG IDL that are known to the client application. If these types of exceptions are thrown by operation implementations, then the BEA Tuxedo system returns them to the client application, unless one of the following conditions exists:

- The object has the `always` transaction policy, and the BEA Tuxedo system automatically started a transaction when the object was invoked. In this case, the transaction is automatically rolled back by the BEA Tuxedo system. Because the client application is unaware of the transaction, the BEA Tuxedo system then raises the `CORBA::OBJ_ADAPTER` CORBA system exception, and not the `CORBA:: TRANSACTION_ROLLEDBACK` exception, which would have been the case had the client initiated the transaction.
- The exception is defined in the file `TobjS_c.h`. In this case, the exception is converted to the `CORBA::BAD_OPERATION` exception and is returned to the client application. In addition, the following message is sent to the user log:

  ```
  "WARN: Application didn't catch TobjS exception. TP Framework throwing
  CORBA::BAD_OPERATION."
  ```

  If the exception is `TobjS::IllegalOperation`, the following supplementary message is written to warn the developer of a possible coding error in the application:

  ```
  "WARN: Application called TP::deactivateEnable() illegally and didn't
  catch TobjS exception."
  ```

  This can occur if the `TP::deactivateEnable()` operation is invoked inside an object that has the `transaction` activation policy. (Application-controlled deactivation is not supported for transaction-bound objects.)
- The BEA Tuxedo system raised an internal system exception following the client invocation. In this case, the `CORBA::INTERNAL` exception is returned to the client. This usually indicates serious system problems with the process in which the object is active.

As defined by the CORBA standard, a reply sent back to the client can either contain result values from the operation implementation, or an exception thrown in the operation implementation, but not both. In the first case—that is, if the reply status value is `NO_EXCEPTION`–the reply contains

the operation's return value and any `inout` or `out` argument values. Otherwise—that is, if the reply status value is `USER_EXCEPTION` or `SYSTEM_EXCEPTION`—all the reply contains is the encoding of the exception.

Exceptions Raised in the `Tobj_ServantBase::deactivate_object()` Operation

If any exception is raised in the `Tobj_ServantBase::deactivate_object()` operation, the following occurs:

- The exception is not returned to the client application.
- If the exception raised is `TobjS::DectivateObjectFailed`, a message is sent to the user log. If a reason string is supplied in the constructor for the exception, the reason string is also written as part of the message.
- A message is sent to the user log for exceptions other than the `TobjS::DeactivateObjectFailed` exception, indicating the type of exception caught by the BEA Tuxedo system.

CORBA Marshal Exception Raised When Passing Object Instances

The ORB cannot marshal an object instance as an object reference. For example, passing a factory reference in the following code fragment generates a CORBA marshal exception in the BEA Tuxedo system:

```
connection::setFactory(this);
```

To pass an object instance, you should create a proxy object reference and pass the proxy instead, as in the following example:

```
CORBA::Object myRef = TP::get_object_reference();
ResultSetFactory factoryRef = ResultSetFactoryHelper::_narrow(myRef);
connection::setFactoryRef(factoryRef);
```

Detecting Error Conditions in the Callback Methods

The BEA Tuxedo system provides a set of predefined exceptions that allow you to specify message strings that the TP Framework writes to the user log if application code gets an error in any of the following callback methods:

- `Tobj_ServantBase::activate_object()`
- `Tobj_ServantBase::deactivate_object()`
- `Server::create_servant()`
- `Server::initialize()`

- `Server::release()`

You can use these exceptions as a useful debugging aid that allows you to send unambiguous information about why an exception is being raised. Note that the TP Framework writes these messages to the user log only. They are not returned to the client application.

You specify these messages with the following exceptions, which have an optional reason string:

Exception	Callback Methods That Can Raise This Exception
`ActivateObjectFailed`	`Tobj_ServantBase::activate_object()`
`DeactivateObjectFailed`	`Tobj_ServantBase::deactivate_object()`
`CreateServantFailed`	`Server::create_servant()`
`InitializeFailed`	`Server::initialize()`
`ReleaseFailed`	`Server::release()`

To send a message string to the user log, specify the string in the exception, as in the following example:

```
throw CreateServantFailed("Unknown interface");
```

Note that when you throw these exceptions, the reason string parameter is required. If you do not want to specify a string with one of these exceptions, you must use the double quote characters, as in the following example:

```
throw ActivateObjectFailed("");
```

Common Pitfalls of OMG IDL Interface Versioning and Modification

The Server object's implementation of the `Server::create_servant()` operation instantiates an object based on its interface ID. It is crucial that this interface ID is the same as the one supplied in the factory when the factory invokes the `TP::create_object_reference()` operation. If the interface IDs do not match, the `Server::create_servant()` operation usually raises an exception or returns a NULL servant. The BEA Tuxedo system then returns a `CORBA::OBJECT_NOT_EXIST` exception to the client application. The BEA Tuxedo system does not perform any validation of interface IDs in the `TP::create_object_reference()` operation.

It is possible for this condition to arise if, during the course of development, different versions of the interface are being developed or many modifications are being made to IDL file. Even if you typically specify string constants for interface IDs in OMG IDL and use these in the factory and the `Server::create_servant()` operation, it is possible for a mismatch to occur if the object implementation and factory are in different executables. This potential problem may be difficult to diagnose.

You may want to consider the following defensive programming strategies during development to avoid this potential problem. This code should be included only in debugging versions of your application, because it introduces performance inefficiencies that may be unacceptable in the production versions of your software.

- Immediately before factory invokes the `TP::create_object_reference()` operation, include code that checks the Interface Repository to see if the required interface exists. Make sure that all the application OMG IDL is up-to-date and loaded into the Interface Repository. Should this check fail to find the interface ID, you can assume that there is a mismatch.
- Following the invocation of the `TP::create_object_reference()` operation in your factories, include code that "pings" the object. That is, the code invokes any operation on the object (typically an operation that does not do anything). If this invocation raises the `CORBA::OBJECT_NOT_EXIST` exception, an interface ID mismatch exists. Note that "pinging" an object causes the object to be activated, with the overhead associated with the activation.

Caveat for State Handling in Tobj_ServantBase::deactivate_object()

The `Tobj_ServantBase::deactivate_object()` operation is invoked when the activation boundary for an object is reached. You may, optionally, write durable state to disk in the implementation of this operation. It is important to understand that exceptions raised in this operation are not returned to the client application. The client application will be unaware of any error conditions raised in this operation unless the object is participating in a transaction. Therefore, in cases where it is important that the client application know whether the writing of state via this operation is successful, we recommend that transactions be used.

If you decide to use the `Tobj_ServantBase::deactivate_object()` operation for writing state, and the client application needs to know the outcome of the write operations, we recommend that you do the following:

- Ensure that each operation that affects object state is invoked within a transaction, and that deactivation occurs within the transaction boundaries. This can be done by using either the `method` or `transaction` activation policies, and is possible with the `process` activation policy if the `TP::deactivateEnable()` operation is invoked within the transaction boundary.
- If an error occurs during the writing of object state, invoke the `COSTransactions::Current::rollback_only()` operation to ensure that the transaction is rolled back. This ensures that the client application receives one of the following exceptions:
  - If the client application initiated the transaction, the client application receives the `CORBA::TRANSACTION_ROLLEDBACK` exception.
  - If the BEA Tuxedo system initiated the transaction, the client application receives the `CORBA::OBJ_ADAPTER` exception.

If transactions are not used, we recommend that you write object state within the scope of individual operations on the object, rather than via the `Tobj_ServantBase::deactivate_object()` operation. This way, if an error occurs, the operation can raise an exception that is returned to the client application.

Servant Pooling

As mentioned in the section "Servant Pooling and Stateless Objects" on page 1-21, servant pooling provides a means to reduce the cost of object instantiation for method-bound or transaction-bound objects.

How Servant Pooling Works

Normally, during object deactivation (that is, when the TP Framework invokes the `Tobj_ServantBase::deactivate_object()` operation), the TP Framework deletes the object's servant; however, when servant pooling is used, the TP Framework does *not* delete the servant at object deactivation. Instead, the server application maintains a pointer to the servant in a pool. When a subsequent client request arrives that can be satisfied by a servant in that pool, the server application reuses the servant and assigns a new object ID. When a servant is reused from a pool, the TP Framework does not create a new servant.

How You Implement Servant Pooling

You implement servant pooling by doing the following:

1. In the `Server::initialize()` operation on the Server object, write the code that sets up the servant pool. The pool consists of a set of pointers to one or more servants, and the code for the pool specifies how many servants for a given class are to be maintained in the pool.

2. In the pooled servant's `Tobj_ServantBase::deactive_object()` operation, you implement the `TP::application_responsibility()` operation. In the implementation of the `TP::application_responsibility()` operation, you provide code that places a pointer to the servant into the servant pool at the time that the TP Framework invokes the `Tobj_ServantBase::deactivate_object()` operation.

3. In the Server object's implementation of the `Server::create_servant()` operation, write code that does the following when a client request arrives:

   a. Checks the pool to see if there is a servant that can satisfy the request.

   b. If a servant does not exist, create a servant and invoke the `Tobj_ServantBase::activate_object()` operation on it.

   c. If a servant exists, invoke the `Tobj_ServantBase::activate_object()` operation on it, assigning the object ID contained in the client request.

Note: Support for the `TP::application_responsibility()` operation has changed in this release. For complete information, see the *CORBA Programming Reference*.

Delegation-based Interface Implementation

There are two primary ways in which an object can be implemented in a BEA Tuxedo CORBA application: by inheritance, or by delegation. When an object inherits from the POA skeleton class, and is thus a CORBA object, that object is said to be implemented by *inheritance*.

However, there may be instances in which you want to use a C++ object in a CORBA application in which inheriting from the POA skeleton class is difficult or impractical. For example, you might have a C++ object that would require a major rewrite to inherit from the POA skeleton class. You can bring this non-CORBA object into a CORBA application by creating a *tie class* for the object. The tie class inherits from the POA skeleton class, and the tie class contains one or more operations that delegate to the legacy class for the implementation of those operations. The legacy class is thereby implemented in the CORBA application by *delegation*.

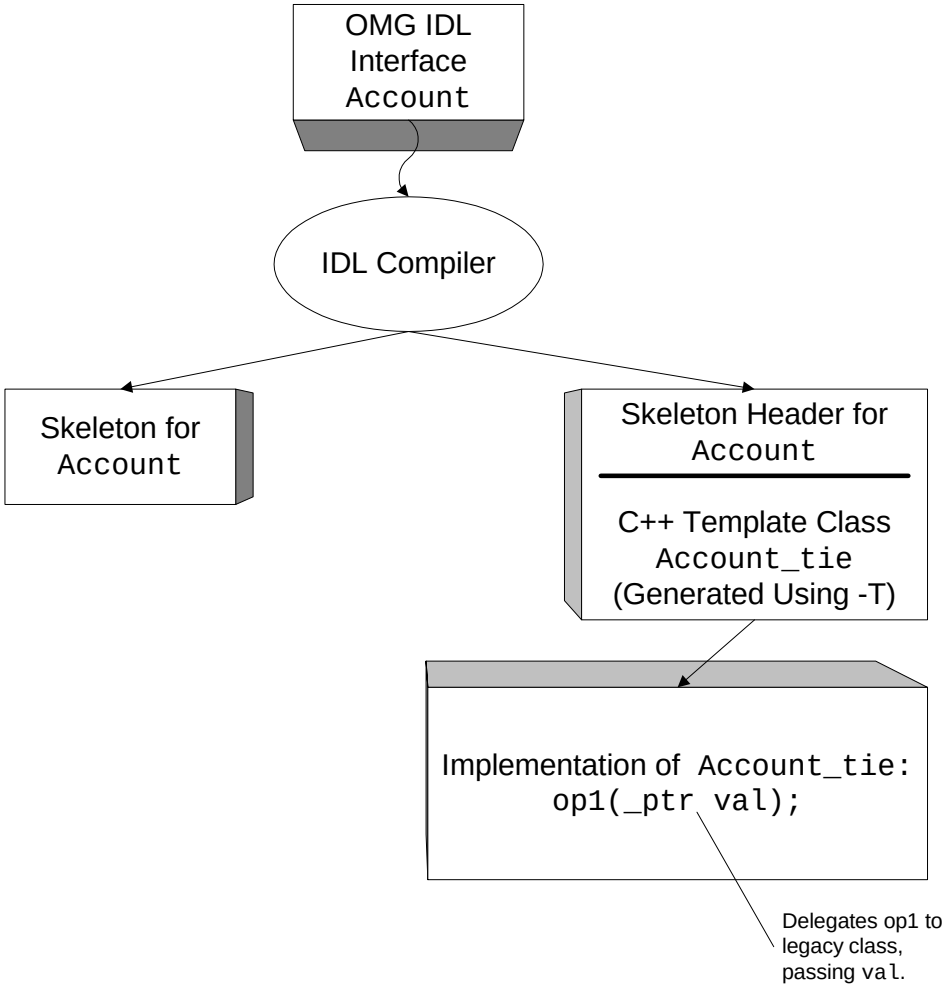
About Tie Classes in the BEA Tuxedo System

To create a delegation-based interface implementation, use the `-T` command-line option of the IDL compiler to generate tie class templates for each interface defined in the OMG IDL file.

Using tie classes in a CORBA application also affects how you implement the `Server::create_servant()` operation in the Server object. The following sections explain the use of tie classes in the BEA Tuxedo product in more detail, and also explains how to implement the `Server::create_servant()` operation to instantiate those classes.

In BEA Tuxedo CORBA, the tie class is the servant, and, therefore, serves basically as a wrapper object for the legacy class.

The following figure shows the inheritance characteristics of the interface `Account`, which serves as a wrapper for a legacy object. The legacy object contains the implementation of the operation `op1`. The tie class delegates `op1` to the legacy class.



Tie classes are transparent to the client application. To the client application, the tie class appears to be a complete implementation of the object that the client application invokes. The tie class delegates all operations to the legacy class, which you provide. In addition, the tie class contains the following:

- Constructor and destructor code, which handles startup and shutdown procedures for the tie class and the legacy class

- Housekeeping code, which implements operations such as accessors

When to Use Tie Classes

Tie classes are not unique to BEA Tuxedo CORBA, and they are not the only way to implement delegation in a CORBA application. However, the BEA Tuxedo CORBA convenience features for tie classes can greatly reduce the amount of coding you need to do for the basic constructor, destructor, and housekeeping operations for those tie classes.

Using tie classes might be recommended in one of the following situations:

- You want to implement an object in a CORBA application in which inheriting from the POA skeleton class is difficult or impractical.
- All the invocations on a legacy class instance can be accomplished from a single servant.
- You are using a legacy class in your CORBA application, and you want to tie the lifetime of an instance of that legacy class to a servant class.
- Delegation is the only purpose of a particular servant; therefore, nearly all the code in that servant is dedicated to legacy object startup, shutdown, access, and delegation.

Tie classes are *not* recommended when:

- The operations on an object instance delegate to more than one legacy object instance.
- Delegation is only a part of the purpose of an object.

How to Create Tie Classes in a CORBA Application

To create tie classes in an application in a BEA Tuxedo domain:

1. Create the interface definition for the tie class in an OMG IDL file, as you would for any object in your application.
2. Compile the OMG IDL file using the `-T` option.

   The IDL compiler generates a C++ template class, which takes the name of the skeleton, with the string `_tie` appended to it. The IDL compiler adds this template class to the skeleton header file.

   Note that the IDL compiler does *not* generate the implementation file for the tie class; you need to create this file by hand, as described in the next step.

3. Create an implementation file for the tie class. The implementation file contains the code that delegates its operations to the legacy class.

4. In the Server object's `Server::create_servant()` operation, write the code that instantiates the legacy object.

   In the following example, the servant for tie class `POA_Account_tie` is created, and the legacy class `LegacyAccount` is instantiated.

```
Account * Account_ptr = new LegacyAccount();
AccountFactoryServant = new POA_Account_tie<LegacyAccount> (Account_ptr)
```

Note: When compiling tie classes with the Compaq C++ Tru64 compiler for UNIX, you must include the `-noimplicit_include` option in the definition of the `CFLAGS` or `CPPFLAGS` environment variables used by the `buildobjserver` command. This option prevents the C++ compiler from automatically including the server skeleton definition file (`_s.cpp`) everywhere the server skeleton header file (`_s.h`) is included, which is necessary to avoid multiply-defined symbol errors. See Compaq publications for additional information about using class templates, such as the tie classes, with Tru64 C++.

CHAPTER 3

Designing and Implementing a Basic CORBA Server Application

This chapter describes how to design and implement a CORBA server application, using the Basic University sample application as an example. The content of this chapter assumes that the design of the application to be implemented is complete and is expressed in OMG IDL. This chapter focuses on design and implementation choices that are oriented to the server application.

This topic includes the following sections:

- How the Basic University Sample Application Works, which helps provide context to the design and implementation considerations
- Design Considerations for the University Server Application, which includes comprehensive discussions about the following topics:
  - Design Considerations for Generating Object References
  - Design Considerations for Managing Object State
  - Design Considerations for Handling Durable State Information
  - How the Basic Sample Application Applies Design Patterns
  - Additional Performance Efficiencies Built into the BEA Tuxedo System
  - Preactivating an Object with State

How the Basic University Sample Application Works

The Basic University sample application provides the student with the ability to browse course information from a central University database. Using the Basic sample application, the student can do the following:

- Browse course synopses from the database by specifying a search string. The server application then returns synopses for all courses that have a title, professor, or description containing the search string. (A course synopsis returned to the client application includes only the course number and title.)
- View detailed information about specific courses. The detailed information available for a specified course includes the following, in addition to synopsis information:
  - Cost
  - Number of credits
  - Class schedule
  - Number of seats
  - Number of registered students
  - Professor
  - Description

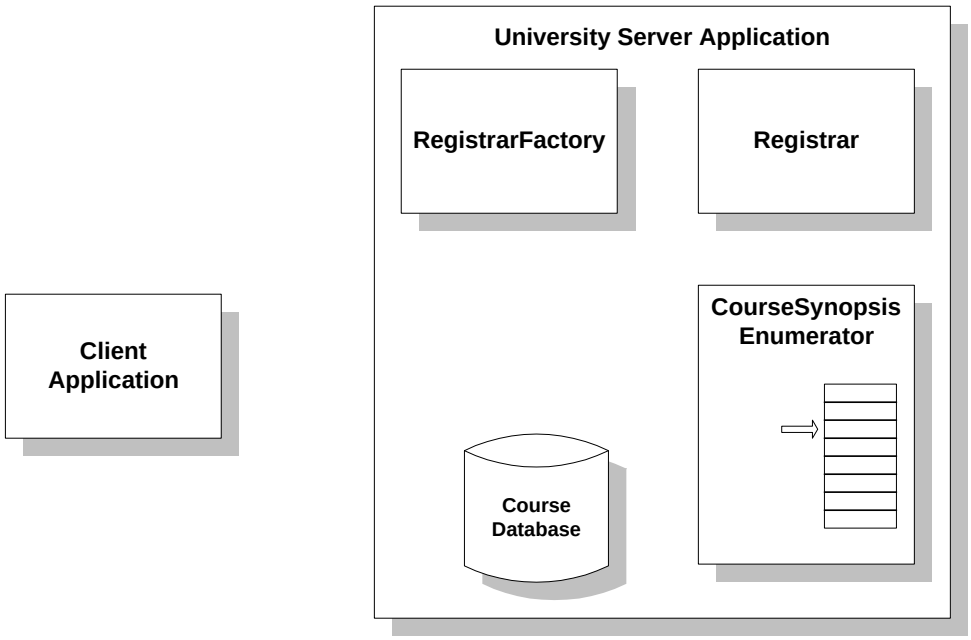
The Basic University Sample Application OMG IDL

In its OMG IDL file, the Basic University sample application defines the following interfaces:

Interface	Description	Operations
`RegistrarFactory`	Creates object references to the `Registrar` object	`find_registrar()`
`Registrar`	Obtains course information from the database	`get_courses_synopsis()` `get_courses_details()`
`CourseSynopsisEnumerator`	Fetches synopses of courses that match the search criteria from the database and reads them into memory	`get_next_n()` `destroy()`

The Basic University sample application is shown in Figure 3-1.

Figure 3-1 Basic University Sample Application



For the purposes of explaining what happens when the Basic University sample application runs, the following separate groups of events are described:

- Application startup—when the server application is booted and the client application gets an object reference to the `Registrar` object
- Browsing course synopses—when the client application sends a request to view course synopses
- Browsing course details—when the client application sends a request to view details on a specific list of courses

Application Startup

The following sequence shows a typical set of events that take place when the Basic client and server applications are started and the client application obtains an object reference to the `Registrar` object:

1. The Basic client and server applications are started, and the client application obtains a reference to the `RegistrarFactory` object from the FactoryFinder.
2. Using the reference to the `RegistrarFactory` object, the client application invokes the `find_registrar()` operation on the `RegistrarFactory` object.
3. The `RegistrarFactory` object is not in memory (because no previous request for that object has arrived in the server process), so the TP Framework invokes the `Server::create_servant()` operation in the Server object to instantiate it.
4. Once instantiated, the `RegistrarFactory` object's `find_registrar()` operation is invoked. The `RegistrarFactory` object creates the `Registrar` object reference and returns it to the client application.

Browsing Course Synopses

The following sequence traces the events that may occur when the student browses a list of course synopses:

1. Using the object reference to the `Registrar` object, the client application invokes the `get_courses_synopsis()` operation, specifying:
   - A search string to be used for retrieving course synopses from the database.
   - An integer, represented by the variable `number_to_get`, which specifies the size of the synopsis list to be returned.
2. The `Registrar` object is not in memory (because no previous request for that object has arrived in the server process), so the TP Framework invokes the `Server::create_servant()` operation, which is implemented in the Server object. This causes the `Registrar` object to be instantiated in the server machine's memory.
3. The `Registrar` object receives the client request and creates an object reference to the `CourseSynopsisEnumerator` object. The `CourseSynopsisEnumerator` object is invoked by the `Registrar` object to fetch the course synopses from the database.

   To create the object reference `CourseSynopsisEnumerator` object, the `Registrar` object does the following:

   a. Generates a unique ID for the `CourseSynopsisEnumerator` object.

   b. Generates an object ID for the `CourseSynopsisEnumerator` object that is a concatenation of the unique ID generated in the preceding step and the search string specified by the client.

c. Gets the `CourseSynopsisEnumerator` object's Interface Repository ID from the interface typecode.

d. Invokes the `TP::create_object_reference()` operation. This operation creates an object reference to the `CourseSynopsisEnumerator` object needed for the initial client request.

4. Using the object reference created in the preceding step, the `Registrar` object invokes the `get_next_n()` operation on the `CourseSynopsisEnumerator` object, passing the list size. The list size is represented by the parameter `number_to_get`, described in step 1.

5. The TP Framework invokes the `Server::create_servant()` operation on the Server object to instantiate the `CourseSynopsisEnumerator` object.

6. The TP Framework invokes the `activate_object()` operation on the `CourseSynopsisEnumerator` object. This operation does the following two things:

   – Extracts the search criteria from its OID.

   – Using the search criteria, fetches matching course synopses from the database and reads them into the server machine's memory.

7. The `CourseSynopsisEnumerator` object returns the following information to the `Registrar` object:

   – A course synopsis list, specified in the return value `CourseSynopsisList`, which is a `sequence` containing the first list of course synopses.

   – The number of matching course synopses that have not yet been returned, specified by the parameter `number_remaining`.

8. The `Registrar` object returns the `CourseSynopsisEnumerator` object reference to the client application, and also returns the following information obtained from that object:

   – The initial course synopsis list

   – The `number_remaining` variable

   (If the `number_remaining` variable is 0, the `Registrar` object invokes the `destroy()` operation on the `CourseSynopsisEnumerator` object and returns a nil reference to the client application.)

9. The client application sends directly to the `CourseSynopsisEnumerator` object its next request to get the next batch of matching synopses.

10. The `CourseSynopsisEnumerator` object satisfies the client request, also returning the updated `number_remaining` variable.

11. When the client application is done with the `CourseSynopsisEnumerator` object, the client application invokes the `destroy()` operation on the `CourseSynopsisEnumerator` object. This causes the `CourseSynopsisEnumerator` object to invoke the `TP::deactivateEnable()` operation.

12. The TP Framework invokes the `deactivate_object()` operation on the `CourseSynopsisEnumerator` object. This causes the list of course synopses maintained by the `CourseSynopsisEnumerator` object to be erased from the server computer's memory so that the `CourseSynopsisEnumerator` object's servant can be reused for another client request.

Browsing Course Details

The following sequence shows a typical set of events that take place when the client application browses course details:

1. The student enters the course numbers for the courses about which he or she is interested in viewing details.

2. The client application invokes the `get_course_details()` operation on the `Registrar` object, passing the list of course numbers.

3. The `Registrar` object searches the database for matches on the course numbers, and then returns a list containing full details for each of the specified courses. The list is contained in the `CourseDetailsList` variable, which is a sequence of `structs` containing full course details.

Design Considerations for the University Server Application

The Basic University sample application contains the University server application, which deals with several fundamental CORBA server application design issues. This section addresses the following topics:

- Design Considerations for Generating Object References
- Design Considerations for Managing Object State
- Design Considerations for Handling Durable State Information
- How the Basic Sample Application Applies Design Patterns

This section also addresses the following two topics:

- Additional Performance Efficiencies Built into the BEA Tuxedo System
- Preactivating an Object with State

Design Considerations for Generating Object References

The Basic client application needs references to the following objects, which are managed by the University server application:

- The `RegistrarFactory` object
- The `Registrar` object
- The `CourseSynopsisEnumerator` object

The following table shows how these references are generated and returned.

Object	How the Object Reference Is Generated and Returned
`RegistrarFactory`	The object reference for the `RegistrarFactory` object is generated in the Server object, which registers the `RegistrarFactory` object with the FactoryFinder. The client application then obtains a reference to the `RegistrarFactory` object from the FactoryFinder. There is only one `RegistrarFactory` object in the Basic University server application process.

Object	How the Object Reference Is Generated and Returned
`Registrar`	The object reference for the `Registrar` object is generated by the `RegistrarFactory` object and is returned when the client application invokes the `find_registrar()` operation. The object reference created for the `Registrar` object is always the same; this object reference does not contain a unique OID. There is only one `Registrar` object in the Basic University server application process.
`CourseSynopsisEnumerator`	The object reference for the `CourseSynopsisEnumerator` object is generated by the `Registrar` object when the client application invokes the `get_courses_synopsis()` operation. In this way, the `Registrar` object is the factory for the `CourseSynopsisEnumerator` object. The design and use of the `CourseSynopsisEnumerator` object is described later in this chapter. There can be any number of `CourseSynopsisEnumerator` objects in the Basic University server application process.

Note the following about how the University server application generates object references:

- The Server object registers the `RegistrarFactory` object with the FactoryFinder. This is the recommended way to ensure that client applications can locate the factories they need to obtain references to the basic objects in the application.
- The object reference to the `Registrar` object is created by the `RegistrarFactory` object. This shows a very common and basic way to return object references to the client application; namely, that there is a factory dedicated to creating and returning references to the primary object that is required by the client application to execute business logic.
- The object reference to the `CourseSynopsisEnumerator` object is created outside a registered factory. In the University sample applications, this is a good design because of the way the `CourseSynopsisEnumerator` object is meant to be used; namely, its existence is specific to a particular client application operation. The

`CourseSynopsisEnumerator` object returns a specific list and results that are not related to the results from other queries.

- Because the `Registrar` object creates, in one of its operations, an object reference to another object, the `Registrar` object is a factory. However, the `Registrar` object is not registered as a factory with the FactoryFinder; therefore, client applications cannot get a reference to the `Registrar` object from the FactoryFinder.

Design Considerations for Managing Object State

Each of the three objects in the Basic sample application has its own state management requirements. This section discusses the object state management requirements for each.

The RegistrarFactory Object

The `RegistrarFactory` object does not need to be unique for any particular client request. It makes sense to keep this object in memory and avoid the expense of activating and deactivating this object for each client invocation on it. Therefore, the `RegistrarFactory` object has the `process` activation policy.

The Registrar Object

The Basic sample application is meant to be deployed in a small-scale environment. The `Registrar` object has many qualities similar to the `RegistrarFactory` object; namely, this object does not need to be unique for any particular client request. Also, it makes sense to avoid the expense of continually activating and deactivating this object for each invocation on it. Therefore, in the Basic sample application, the `Registrar` object has the `process` activation policy.

The CourseSynopsisEnumerator Object

The fundamental design problem for the University server application is how to handle a list of course synopses that is potentially too big to be returned to the client application in a single response. Therefore, the solution centers on the following:

- To begin a conversation between the client application and an object that can fetch the course synopses from the University database.
- To have the object return an initial batch of synopses to the client application.
- To keep the remainder of the course synopses in memory so that the client application can retrieve them one batch at a time.

- To have the client application terminate the conversation when finished, thus freeing machine resources.

The University server application has the `CourseSynopsisEnumerator` object, which implements this solution. Although this object returns an initial batch of synopses when it is first invoked, this object retains an in-memory context so that the client application can get the remainder of the synopses in subsequent requests. To retain an in-memory context, the CourseSynopsisEnumerator object must be stateful; that is, this object stays in memory between client invocations on it.

When the client is finished with the `CourseSynopsisEnumerator` object, this object needs a way to be flushed from memory. Therefore, the appropriate state management decision for the `CourseSynopsisEnumerator` object is to assign it the `process` activation policy and to implement the CORBA application-controlled deactivation feature.

Application-controlled deactivation is implemented in the `destroy()` operation on that object.

The following code example shows the `destroy()` operation on the `CourseSynopsisEnumerator` object:

```
void CourseSynopsisEnumerator_i::destroy()
{
        // When the client calls "destroy" on the enumerator,
        // then this object needs to be "destructed".
        // Do this by telling the TP framework that we're
        // done with this object.

        TP::deactivateEnable();
}
```

Basic University Sample Application ICF File

The following code example shows the ICF file for the Basic sample application:

```
module POA_UniversityB
{
    implementation CourseSynopsisEnumerator_i
    {
        activation_policy  ( process                                );
        transaction_policy ( optional                               );
        implements         ( UniversityB::CourseSynopsisEnumerator );
    };
    implementation Registrar_i
    {
```

```
        activation_policy  ( process                              );
        transaction_policy ( optional                             );
        implements         ( UniversityB::Registrar               );
    };
    implementation RegistrarFactory_i
    {
        activation_policy  ( process                              );
        transaction_policy ( optional                             );
        implements         ( UniversityB::RegistrarFactory        );
    };
};
```

Design Considerations for Handling Durable State Information

Handling durable state information refers specifically to reading durable state information from disk at some point during or after the object activation, and writing it, if necessary, at some point before or during deactivation. The following two objects in the Basic sample application handle durable state information:

- The `Registrar` object
- The `CourseSynopsisEnumerator` object

The following two sections describe the design considerations for how these two objects handle durable state information.

The Registrar Object

One of the operations on the `Registrar` object returns detailed course information to the client application. In a typical scenario, a student who has browsed dozens of course synopses may be interested in viewing detailed information on perhaps as few as two or three courses at one time.

To implement this usage scenario efficiently, the `Registrar` object is defined to have the `get_course_details()` operation. This operation accepts an input parameter that specifies a list of course numbers. This operation then retrieves full course details from the database and returns the details to the client application. Because the object in which this operation is implemented is process-bound, this operation should avoid keeping any state data in memory after an invocation on that operation is complete.

The `Registrar` object does not keep any durable state in memory. When the client application invokes the `get_course_details()` operation, this object simply fetches the relevant course information from the University database and sends it to the client. This object does not keep any

course data in memory. No durable state handling is done via the `activate_object()` or `deactivate_object()` operations on this object.

The CourseSynopsisEnumerator Object

The `CourseSynopsisEnumerator` object handles course synopses, which this object retrieves from the University database. The design considerations, with regard to handling state, involve how to read state from disk. This object does not write any state to disk.

There are three important aspects of how the `CourseSynopsisEnumerator` object works that influence the design choices for how this object reads its durable state:

- The OID for this object contains the search criteria provided in the initial client request for synopses. The search criteria work as a key to the database: this object extracts information from the database based on search criteria stored in the OID.
- All the operations on this object use the course synopses that this object reads into memory.
- This object must flush course synopses from memory when it is deactivated.

Given these three aspects, it makes sense for this object to:

- Read its durable state information when activated; namely, via the `activate_object()` operation on this object.
- Flush the course synopses from memory when deactivated; namely, via the `deactivate_object()` operation.

Therefore, when the `CourseSynopsisEnumerator` object is activated, the `activate_object()` operation on this object does the following:

1. Extracts the search criteria from its OID.
2. Retrieves from the database course synopses that match the search criteria.

Note: If you implement the `Tobj_ServantBase::activate_object()` or `Tobj_ServantBase::deactivate_object()`operations on an object, remember to edit the implementation header file (that is, the *`application`*`_i.h` file) and add the definitions for those operations to the class definition template for the object's interface.

Using the University Database

Note the following about the way in which the University sample applications use the University database:

- All of the University sample applications access the University database to manipulate course and student information. Typically this is a large part of the code you write in the implementation files. To make the University sample implementation files simpler, and to help you focus on CORBA features instead of database code, the samples have wrapped all the code that reads and writes to the database within a set of classes. The file `samplesdb.h` in the `utils` directory contains the definitions of these classes. These classes make all the necessary SQL calls to read and write the course and student records in the University database.

Note: The BEA Tuxedo Teller Application in the Wrapper and Production sample applications accesses the account information in the University database directly and does not use the `samplesdb.h` file.

For more information on the files you build into the Basic server application, see the *Guide to the CORBA University Sample Applications*.

- The `CourseSynopsisEnumerator` object 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. For more information about how the University sample applications use transactions, see Chapter 6, "Integrating Transactions into a CORBA Server Application."

How the Basic Sample Application Applies Design Patterns

The Basic sample application uses the following design patterns:

- Process-Entity
- List-Enumerator

This section describes why these two patterns are appropriate for the Basic sample application and how this application implements them.

Process-Entity Design Pattern

As mentioned in the section "Process-Entity Design Pattern" on page 1-22, this design pattern is appropriate in situations where you can have one process object that handles data entities needed by the client application. The data entities are encapsulated as CORBA `structs` that are manipulated by the process object and not by the client application.

Adapting the Process-Entity design pattern to the Basic sample application allows the application to avoid implementing fine-grained objects. For example, the `Registrar` object is an efficient

alternative to a similarly numerous set of course objects. The processing burden of managing a single, coarse-grained `Registrar` object is small relative to the potential overhead of managing hundreds or thousands of fine-grained course objects.

For complete details about the Process-Entity design pattern, see the Design Patterns technical article.

List-Enumerator Design Pattern

This design pattern is appropriate in situations where an object has generated an internal list of data that is potentially too large to return to the client application in a single response. Therefore, the object must return an initial batch of data to the client application in one response, and have the ability to return the remainder of the data in subsequent responses.

A list-enumerator object must also simultaneously keep track of how much of the data has already been returned so that the object can return the correct subsequent batch. List-enumerator objects are always stateful (that is, they remain active and in memory between client invocations on them) and the server application has the ability to deactivate them when they are no longer needed.

The list-enumerator design pattern is an excellent choice for the `CourseSynopsisEnumerator` object, and implementing this design pattern provides the following benefits:

- The University server application has a means to return potentially large lists of course synopses in a way that client applications can handle; namely, in manageable chunks.
- Each `CourseSynopsisEnumerator` object is unique, and its content is determined by the request that caused this object to be created. (In addition, each `CourseSynopsisEnumerator` object ID is also unique.) When the client invokes the `get_courses_synopsis()` operation on the `Registrar` object, the `Registrar` object returns the following:
  - An initial list of synopses.
  - An object reference to a `CourseSynopsisEnumerator` object that can return the remainder of the synopses.

  Therefore, all subsequent invocations go to the correct `CourseSynopsisEnumerator` object. This is critical in the situation where the server process has multiple active instances of the `CourseSynopsisEnumerator` class.

  Because the `get_courses_synopsis()` operation returns a unique `CourseSynopsisEnumerator` object reference, client requests never collide; that is, a client request never mistakenly goes to the wrong `CourseSynopsisEnumerator` object.

Although the `Registrar` object has the `get_courses_synopsis()` operation on it, the knowledge of the database query and the synopsis list is embedded entirely in the `CourseSynopsisEnumerator` object. In this situation, the `Registrar` object serves only as a means for the client to get the following:

- The initial list of synopses.
- A reference to a `CourseSynopsisEnumerator` object that can return the remainder of the synopses.

Additional Performance Efficiencies Built into the BEA Tuxedo System

The BEA Tuxedo system implements a performance efficiency in which data marshaling between two objects in the same server process is automatically disabled. This efficiency exists if the following circumstances exist:

- An object reference routes to the same group as the one containing the server process in which the object reference was created.
- An object in that server process invokes an operation using that object reference that causes an object to be instantiated in the same process.

An example of this is when the `Registrar` object creates an object reference to the `CourseSynopsisEnumerator` object and causes that object to be instantiated. No data marshaling takes place in the requests and responses between those two objects.

Preactivating an Object with State

The preactivate object with state feature allows you to preactivate an object before a client application invokes that object. This feature can be particularly useful for creating iterator objects, such as the `CourseSynopsisEnumerator` object in the University samples.

Preactivating an object with state centers around using the `TP::create_active_object_reference()` operation. Typically, objects are not created in a CORBA server application until a client issues an invocation on that object. However, by preactivating an object and using the `TP::create_active_object_reference()` operation to pass a reference to that object back to the client, your client application can invoke an object that is already active and populated with state.

Note: The preactivate object with state feature was first introduced in WebLogic Enterprise version 4.2.

How You Preactivate an Object with State

The process for using the preactivation feature is to write code in the server application that:

1. Includes an invocation of the C++ `new` statement to create an object.
2. Sets the object's state.
3. Invokes the `TP::create_active_object_reference()` operation to obtain a reference for the newly created object. This object reference can then be returned to the client application.

Thus, the preactivated object is created in such a way that the TP Framework invokes neither the `Server::create_servant()` nor the `Tobj_ServantBase::activate_object()` operations for that object.

Usage Notes for Preactivated Objects

Note the following when using the preactivation feature:

- Preactivated objects must have the `process` activation policy. Therefore, these objects can be deactivated only at the end of the process or by an invocation to the `TP::deactivateEnable()` operation on those objects.
- The object reference created by the `TP::create_active_object_reference()` operation is *transient*. This is because a preactivated object should exist only for the lifetime of the process in which it was created, and this object should not be reactivated again in another server process.

  If a client application invokes on a transient object reference after the process in which the object reference was created is shut down, the TP Framework returns the following exception:

  `CORBA::OBJECT_NOT_EXIST`
- For objects that are preactivated, the state usually cannot be recovered if a crash occurs. However, this is acceptable because such objects are typically meant to be used within the context of a specific series of operations, and then deleted. Its state has no meaning outside that specific series.

  To prevent the situation in which a server has crashed, and a client application subsequently attempts to invoke the now-deleted object, add the `TobjS::ActivateObjectFailed` exception to the implementation of the `Tobj_ServantBase::activate_object()` operation to the object meant for preactivation. Then, if a client attempts to invoke such an object after a server crash, in which case the TP Framework invokes the `Tobj_ServantBase::activate_object()`

operation on that object, the TP Framework returns the following exception to the client application:

```
CORBA::OBJECT_NOT_EXIST
```

- Use preactivation sparingly because, as with all process-bound objects, preactivation preallocates scarce resources.

CHAPTER 4

Creating Multithreaded CORBA Server Applications

This topic includes the following sections:

- Overview
- Developing and Building Multithreaded CORBA Server Applications
- Building and Running the Multithreaded Simpapp Sample Application
- Multithreaded CORBA Server Application Administration

Overview

This topic includes the following sections:

- Introduction
- Mechanisms for Supporting Multithreaded CORBA Servers
- Running Single-threaded Server Applications in a Multithreaded System

Introduction

Designing an application to use multiple, independent threads provides concurrency within an application and can improve overall throughput. Using multiple threads enables applications to be structured efficiently with threads servicing several independent tasks in parallel. Multithreading is particularly useful when:

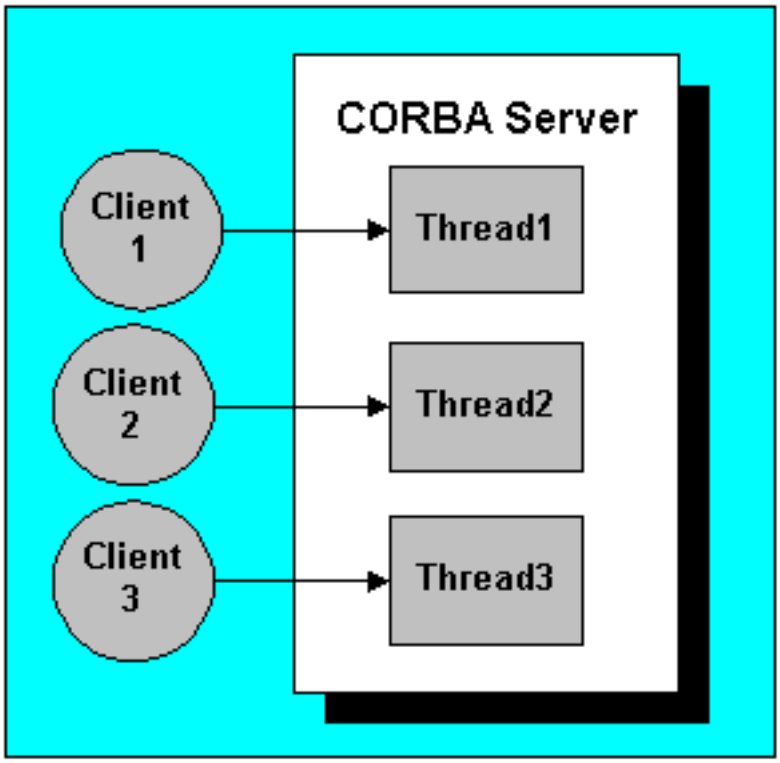
- There is a set of lengthy operations that do not necessarily depend on other processing.
- The amount of data to be shared is small and identifiable.
- You can break the task into various activities that can be executed in parallel.
- There are occasions where objects must be reentrant.

Historically, industry-wide, multithreaded applications have been complicated to design and implement. The support provided by BEA Tuxedo simplifies this complexity by managing threads within a CORBA server environment.

The BEA Tuxedo software supports server applications that have the following multithreading characteristics (see Figure 4-1):

- Instances of server objects can handle multiple client requests simultaneously.
- A server object can make recursive invocations on itself.
- Server objects can create and monitor their own threads to implement parallelism within a servant method.

Figure 4-1 Multithreaded CORBA Server Application



Generally, the BEA Tuxedo software creates and manages threads on behalf of a server application. Building multithreaded server applications affects how you use the TP Framework, implement servants, and design objects that create their own threads.

The BEA Tuxedo software allows you to implement either the thread-per-request model or a thread-per-object model. Each model is explained in "Threading Models" on page 4-5.

Requirements, Goals, and Concepts

Some computer operations take a substantial amount of time to complete. A multithreaded design can significantly reduce the wait time between the request and completion of operations. This is true in situations when operations perform a large number of I/O operations such as when accessing a database, invoking operations on remote objects, or are CPU-bound on a

multiprocessor machine. Implementing multithreading in a server process can increase the number of requests a server processes in a fixed amount of time.

The primary requirement for multithreaded server applications is the simultaneous handling of multiple client requests. The motivations for developing this type of server are to:

- **Simplify program design**

  This is achieved by allowing multiple server tasks to proceed independently using conventional programming abstractions.

- **Improve throughput**

  This is achieved by taking advantage of the parallel processing capabilities of multiprocessor hardware platforms and overlapping computation with communication.

- **Improve perceived response time**

  By associating separate threads with different server tasks, clients do not block each other for an extended period of time.

- **Simplify coding of remote procedure calls and conversations**

  Some applications are easier to code when you use separate threads to interact with different remote procedure calls (RPCs) and conversations.

- **Provide simultaneous access to multiple applications**

  When wrapping legacy applications or databases in a CORBA server, implementations can interact with more than one legacy application at a time.

- **Reduce the number of servers required**

  Because one server can dispatch multiple service threads, the number of servers your application requires can be reduced.

However, a multithreaded design is not without cost. In general, multithreaded server applications require more complicated synchronization strategies than single-threaded servers. An application developer must write thread-safe code. Additionally, the overhead of creating a thread to handle a request might be greater than the potential benefit of parallelism. The actual performance of a particular concurrency model depends on the following factors:

- **Characteristics of requests from the client**

  Are the requests of long or short duration?

- **How threads are implemented**

Are the threads managed in the operating system kernel, in a library in user space, or some combination of both?

- **Operating system and network overhead**

  How much additional overhead is introduced by repeatedly setting up and tearing down connections?

- **Higher-level system configuration factors**

  Do replication, dynamic load balancing, or other factors affect performance?

While threading libraries provide the mechanisms for creating concurrency models, developers are ultimately responsible for knowing how to use the mechanisms successfully. By studying design patterns, application developers can master the subtle differences and make better design choices for different situations.

Threading Models

There are a number of different models you can use for designing concurrency in servers. The following sections describe the thread-per-request model, the thread-per-object model, the thread pool, and how the BEA Tuxedo software implements each model. A specific server is designed for either the thread-per-request model or the thread-per-object model.

Thread-Per-Request Model

In this model, each request from a client is processed in a different thread of control. This model is useful when a server typically receives requests of long duration from multiple clients. It is less useful for requests of short duration due to the overhead of creating a new thread for each request. Each time a new request arrives, BEA Tuxedo associates that request with a thread and executes it. Because a multithreaded application server process can host more than one thread at a time, it can simultaneously execute more than one client request at a time. BEA Tuxedo controls the association of a request to a thread, therefore, applications do not need to explicitly create threads unless the applications require a greater degree of control than that provided by BEA Tuxedo.

The thread-per-request model requires that you design your application servers to be thread-safe, which means that you must implement concurrency mechanisms to control access to data that might be shared among multiple server objects. The need to use concurrency control mechanisms increases the complexity of the applications development process. Additionally, if many clients make requests simultaneously, this design can consume a large number of operating system resources.

Thread-Per-Object Model

The thread-per-object model associates each active object in the server process with a single thread at any one time. Each request for an object establishes an association between a dispatch thread and the object. Serial requests for the same object can be serviced by different threads. A specific thread can be shared by multiple objects.

The Thread Pool

Thread pools are a means to reduce the cost of managing threads. At startup and as needed, threads are created, assigned, and released to a pool of available threads where the thread waits until it is needed again to process future requests. Thread pools can be used to support any of the threading models previously described. For example, a thread may be allocated for a request in a thread-per-request model, used for the method execution, and released back to the pool.

Allocating and deallocating threads can be time-consuming and expensive, especially for short-lived requests and objects. Thread pools provide a means of reducing the cost of managing threads. During startup, or as needed, threads are created, assigned, and released by the BEA Tuxedo software to a pool of available threads. A thread exists in the pool and waits until it is needed to process future requests.

The initial and ultimate size of the BEA Tuxedo thread pool for an application server process is controlled through settings in the server configuration file. At startup, the minimum pool size is pre-allocated. As requests arrive to be serviced, the BEA Tuxedo software assigns a thread from the pool to handle the request. If the pool does not contain an available thread to process the request and the pool has not been filled, the BEA Tuxedo software creates a new thread to handle the request. If a request arrives when there are no threads available in the pool, and no new threads can be created, the request will be queued until a thread is available.

Thread pools are appropriate for situations in which you want to limit the amount of system resources that can be consumed for server threading. When a thread pool is used, client requests are executed concurrently until the number of simultaneous requests exceeds the number of threads in the pool.

The BEA Tuxedo thread pool has the following characteristics and behavior:

- You can set the maximum size of the pool as a BEA Tuxedo administration function. You can adjust the size of this pool without making changes to the application itself.
- The BEA Tuxedo software allocates threads from the pool as necessary. The threads are used during the processing of a request, and are then released back to the pool.
- Threads can be serially reused for servicing multiple requests and multiple objects.

Reentrant Servants

The BEA Tuxedo software provides the capability for an object to invoke operations on itself recursively. Using this capability requires a great deal of care in how you implement an object, because the application code must employ the operating system concurrency mechanisms needed to control access to shared state data. In some cases, such as with objects that implement the Process or Distribution Adapter design patterns, there is little or no shared state for an object, and it is relatively easy to support reentrancy.

BEA Tuxedo software also allows you to enable or prohibit reentrant method invocations on an active object. Reentrancy is disabled by default. If a request for an active object is received while the object is currently executing another request in a different thread, the following rules apply:

- If the `_is_reentrant` method returns `TRUE`, a new thread is allocated from the pool and the request is dispatched to the appropriate method using the same servant instance. It is the responsibility of the servant implementation code to ensure the integrity of the state of the object when multiple threads interact with it.
- If the `_is_reentrant` method returns `FALSE`, a new instance of the servant is created and the method is dispatched to the new instance. This instance is not automatically deleted. Future reentrant requests may be dispatched to either instance.

Note: The reentrant servant mechanism is available only when a server is started with the `PER_REQUEST` concurrency strategy specified.

For information about using this method, see the *CORBA Programming Reference*.

The Current Object

One of the most important attributes of a multithreaded CORBA server application environment is ensuring that the Current object is used and managed correctly. This ensures behavior such as the following:

- Individual threads function within the correct transaction and security contexts.
- The Current object behaves correctly when accessed from different threads.

The BEA Tuxedo product conforms to the multithreading model defined by the ORB Portability Specification, published by the OMG, which has been incorporated into the OMG CORBA specification. In the BEA Tuxedo product, operations on interfaces derived from `CORBA::Current` have access to the state associated with the thread in which operations are invoked, not to the state associated with the thread from which the Current object was obtained. The reason for this behavior is twofold:

- Prevents one thread from manipulating the state of another thread
- Avoids the need to obtain and narrow a new Current object in the thread context for each method

When used in a multithreaded environments, the behaviors of the following objects are consistent with the ORB Portability Specification:

- `CosTransactions::Current`
- `SecurityLevel1::Current`
- `SecurityLevel2::Current`
- `PortableServer::Current`

For example, when an application passes a transaction from one thread to another, the application should not use the `CosTransactions::Current` object. Instead, the application passes the `CosTransactions::Control` object to the other thread. To pass the `CosTransctions::Current` object would only allow the receiving thread to gain access to the transaction state associated with that thread.

Mechanisms for Supporting Multithreaded CORBA Servers

This section provides an overview of the following tools, APIs, and administrative capabilities in BEA Tuxedo CORBA that support multithreaded server applications:

- Context Services
- Classes and Methods in the TP Framework
- Capabilities in the Build Commands
- Tools for Administration

Context Services

You can choose to create and manage your own threads in your object implementations. Other threads are managed automatically by the BEA Tuxedo CORBA software. The BEA Tuxedo CORBA software maintains context information internally for each thread that it creates and maintains. This required context information is used during the processing of CORBA requests. Since BEA Tuxedo CORBA has no knowledge of when an application creates and deletes its own threads, the context services mechanism allows programmers to initialize their own threads correctly, prior to calling BEA Tuxedo services, and to release any context resources that are no longer needed when a thread is deleted.

The following set of ORB methods satisfies the thread management requirements. Together these are called **context services**:

- `ORB::get_ctx()`

  When an object creates a thread, the object invokes this operation on the ORB to obtain system context information that the object can pass onto the thread. This operation must be called from a thread that already has a context. For example, the thread in which a method was dispatched will already have a context. For information about using this operation, see `ORB::get_ctx()` in the *CORBA Programming Reference*.

- `ORB::set_ctx()`

  When an object spawns a thread, the spawned thread typically retrieves the context information from the thread that invoked the `get_ctx` method. The spawned thread then uses the retrieved context information when invoking `ORB::set_ctx` to set the system context in which the spawned thread should execute. For information about using this operation, see `ORB::set_ctx()` in the *CORBA Programming Reference*.

- `ORB::clear_ctx()`

  When a spawned thread has completed its work, the thread invokes this method to dissociate itself from the system context. For information about using this operation, see `ORB::clear_ctx()` in the *CORBA Programming Reference*.

- `ORB::inform_thread_exit()`

  When a thread has completed its work, the thread invokes this method to inform the BEA Tuxedo system that resources associated with an application-managed thread can be released. For information about using this operation, see `ORB::inform_thread_exit()` in the *CORBA Programming Reference*.

Classes and Methods in the TP Framework

These classes and methods in the BEA Tuxedo TP Framework support multithreaded server applications:

- `ServerBase` class

  To override the default implementations of the `ServerBase` class, an application developer can create a class that derives from `ServerBase`. In addition to `ServerBase` methods already supported, these methods are provided to support the implementation of multithreaded server applications:

  - `create_servant_with_id()`
  - `thread_initialize()`

- `thread_release()`

These methods allow you to obtain a high-degree of granularity of control over the multithreading characteristics of your application. For information on how to use these methods see `ServerBase Class` in the *CORBA Programming Reference*.

- `Tobj_ServantBase` class

  This class provides these methods to support multithreaded server applications:

  - `Tobj_ServantBase::_is_reentrant()`
  - `Tobj_ServantBase::_add_ref()`
  - `Tobj_ServantBase::_remove_ref()`

  For information about using these methods, see `Tobj_ServantBase Class` in the *CORBA Programming Reference*.

Capabilities in the Build Commands

The `buildobjserver` and `buildobjclient` commands include the following thread-management capabilities.

- The `buildobjserver` command includes platform-specific thread library support so that server applications are compatible with the multithreading support in the BEA Tuxedo software.

  The `buildobjserver` command includes command-line options for building multithreaded or single-threaded server applications.

- The `buildobjclient` command includes platform-specific thread library support so that client applications can be compatible with the multithreading support provided in the BEA Tuxedo software.

Tools for Administration

The BEA Tuxedo system employs configuration files to assemble and run applications. Typically, the application developer creates these files, and BEA Tuxedo system administrators modify the contents of the file as necessary to satisfy application and system requirements.

The control parameters associated with the support of threads specify the following:

- Whether a server should be single-threaded or multithreaded
- The size of the thread pool available for dispatching methods on objects

For more information about threads parameters in the UBBCONFIG file, see "Sample UBBCONFIG File" on page 4-36.

Running Single-threaded Server Applications in a Multithreaded System

The default behavior of the threading support provided in BEA Tuxedo CORBA is to emulate a single-threaded server support environment. To run a single-threaded CORBA application in a multithreaded environment, you do not need to change the server application code or the configuration files. However, before you run an existing single-threaded application, you must rebuild it using the `buildobjserver` and `buildobjclient` commands. If you do not specifically enable multithreading for a server application, the application runs as a single-threaded server.

Developing and Building Multithreaded CORBA Server Applications

This topic includes the following sections:

- Using the buildobjserver Command
- Using the buildobjclient Command
- Creating Non-reentrant Servants
- Creating Reentrant Servants
- Building and Running the Multithreaded Simpapp Sample Application

Using the buildobjserver Command

The `buildobjserver` command supports multithreaded CORBA server applications through the following capabilities:

- Platform-specific Thread Libraries
- Specifying Multithreaded Support
- Specifying an Alternate Server Class

Platform-specific Thread Libraries

Server applications generated by the `buildobjserver` command are compiled using the correct platform-specific compiler settings, and are linked using the correct platform-specific thread support libraries. This ensures compatibility with the shared libraries provided by the BEA Tuxedo software.

Specifying Multithreaded Support

When you create a CORBA server application to support multithreading, you must specify the `-t` option on the `buildobjserver` command when you build the application. At run time, the BEA Tuxedo system verifies compatibility between the executable program and the threading model selected in the CORBA server application configuration file `UBBCONFIG`. For information on how to set the threading model in the `UBBCONFIG` file, see "Sample UBBCONFIG File" on page 4-36.

Note: When you specify `-t` in your build of a CORBA server application, you should set the `MAXDISPATCHTHREADS` parameter in the `UBBCONFIG` file to a value greater than 1; otherwise, the CORBA server application will run as a single-threaded server.

Note: Multithreaded joint client/server implementations are not supported.

If you attempt to start a single-threaded executable with an incompatible threading model specification in the configuration file, these events occur:

- The BEA Tuxedo software records a warning in the log file.
- The server executable program is started as a single-threaded server.

Specifying an Alternate Server Class

If you implement your own `Server` class, inheriting from the `ServerBase` class, you must specify your alternate `Server` class in the `buildobjserver` command using the `-b` option. The `buildobjserver` command provides the following syntax to support the `-b` option:

```
buildobjserver [-v] [-o outfile] [-f {firstfiles|@def-file}]
[-l {lastfiles|@def-file}] [-r rmname] [-b bootserverclass] [-t]
```

In the preceding syntax, the value for *`bootserverclass`* specifies the C++ class to be used when the CORBA server application is booted. If you do not specify the `-b` option, the BEA Tuxedo system creates an instance of the class named `Server`.

When you specify the `-b` option, the Tuxedo system creates a main function for the alternate server class, and your project must supply a header file with the name you specified for *`bootserverclass`* on the `-b` option. The header file contains the definition of the alternate C++ class. This alternate `Server` class must inherit from the `ServerBase` class.

For example, if the command line specifies `-b AslanServer`, the application project must supply an `AslanServer.h file`. The `AslanServer.h` file is an example of a *`bootserverclass`*`.h` file. A `bootserverclass` file provides logic similar to this code sample:

Listing 4-1 Example of a bootserverclass.h File

```
// File name: AslanServer.h
#include <Server.h>
class AslanServer : public ServerBase {
    public:
     CORBA::Boolean initialize(int argc, char** argv);
     void release();
```

```
    Tobj_Servant create_servant(const char* interfaceName);
     Tobj_Servant create_servant_with_id(const char* interfaceName,
                                         const char* stroid);
    CORBA::Boolean thread_initialize(int argc, char** argv);
    void thread_release();
};
```

Using the buildobjclient Command

When you use the `buildobjclient` command to create a client application executable program, the application is compiled using the correct platform-specific compiler settings and linked using the correct thread support libraries for your operating system. This ensures that clients are compatible with the shared libraries provided by the BEA Tuxedo software.

Creating Non-reentrant Servants

Before you can run any CORBA server application in the BEA Tuxedo CORBA environment, you must build it using the `buildobjserver` command.

Use the `buildobjserver -t` option to inform the BEA Tuxedo system that the CORBA server application is thread safe. The `-t` option indicates that the application does not employ shared context data or other programming constructs that are not thread safe. If you run single-threaded applications that are not thread safe in a multithreaded environment, you risk data corruption.

If you update configuration files for an application to enable multithreading support, but the application code has not been updated to indicate that the servant implementation can support reentrancy, note the following:

- Methods are executed in arbitrary threads assigned by the BEA Tuxedo system.
- Servant implementation code does not necessarily protect an object from concurrent access to its state. However, active servants are limited to a single thread of execution at a time.
- You cannot assume that a method is executed in a specific thread. Do not use storage that depends on or is tied to a specific thread.
- Do not assume that the servant's `activate_object` or `deactivate_object` methods are executed in the same thread as the request in which they were originally invoked.

- Additional application-managed threads can be created within a servant method. Your object implementations must ensure that threads are created, handled, and destroyed properly.
- An application-managed thread can include invocations on other objects.
- Do not use signals for synchronization; the mixing of signals and threads is not supported.

Note: The `SIGKILL` signal to terminate a process is supported. The use of `SIGIO` is not supported in BEA Tuxedo CORBA for single or multithreaded applications.

- Request-level interceptors are invoked by BEA Tuxedo CORBA through the same thread used by the method.

Creating Reentrant Servants

To create a multithreaded reentrant servant:

- Build the CORBA server application using the `buildobjserver` command with the `-t` option, and modify the `UBBCONFIG` server configuration file for the application.
- Update the CORBA server application code to enable reentrancy using the `TobjServantBase::_is_reentrant` method.
- Start the server using the thread-per-request threading model, by specifying `CONCURR_STRATEGY = PER_REQUEST` in the `UBBCONFIG` file.

If you do create a multithreaded, reentrant servant, the implementation code for that object must protect the state of the object, in order to ensure its integrity while multiple threads interact with it.

Considerations for Client Applications

There are considerations for CORBA client applications running in the BEA Tuxedo environment:

- Multithreaded CORBA clients using IIOP are supported.
- Multithreaded native CORBA clients are not supported.
- A multithreaded CORBA client is limited to a single bootstrap object.
- A multithreaded CORBA client is limited to a single logon.
- CORBA clients using stub-based invocation are supported.
- CORBA clients using the Dynamic Invocation Interface (DII) are not supported.

Building and Running the Multithreaded Simpapp Sample Application

This topic includes the following sections:

- About the Simpapp Multithreaded Sample
- How the Sample Application Works
- How to Build and Run the Sample Application
- Shutting Down the Sample Application

About the Simpapp Multithreaded Sample

The BEA Tuxedo software provides a multithreaded CORBA sample application, consisting of a client program and a CORBA server program. The server receives an alphabetic string from the client and returns the string in uppercase and lowercase letters. The multithreading capability of simpapp_mt provides parallel processing. Through this parallelism, a single server process can handle concurrent requests from multiple clients for multiple objects or for a single object.

Note: The client application in the simpapp_mt sample is not a multithreaded client application.

How the Sample Application Works

The purpose of a multithreaded server is to handle multiple requests from one or more clients in a parallel manner. The simpapp_mt sample application is a CORBA application that demonstrates multithreading functionality, by using the `buildobjserver -t` command-line option and using the `UBBCONFIG` file to specify concurrency strategy.

The simpapp_mt sample first creates a server process named `SimplePerObject` and secondly a server process named `SimplePerRequest`. The client communicates first with the `SimplePerRequest` server and then with the `SimplePerObject` server.

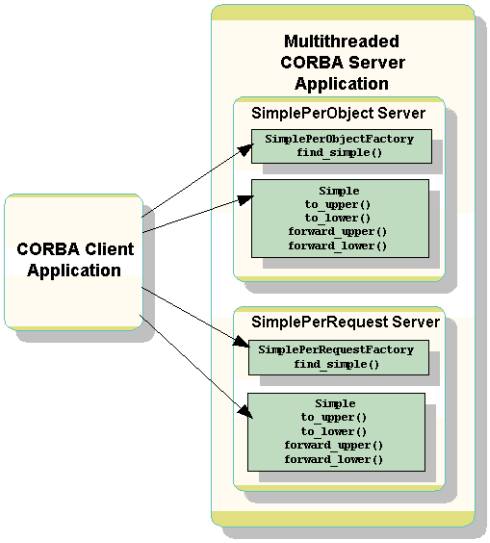
The thread-per-request server implementation for `SimplePerRequest` demonstrates the use of a user-defined server class that implements thread initialization methods. The `SimplePerRequest` server process handles each request from a client in a separate thread of control. Each time a new request arrives, a thread is allocated from the thread pool to handle the request. Once the request has been processed and the reply sent, the thread is released back to the pool. This model is useful for servers that handle long-duration requests from multiple clients.

The simpapp_mt sample application provides an implementation of a CORBA object that has the following methods:

- The `to_upper` method accepts a string from the client application and converts it to uppercase letters.
- The `to_lower` method accepts a string from the client application and converts it to lowercase letters.
- The `forward_upper` method creates an application-managed thread to another instance of the server and forwards the request received from the client to the new server instance to convert the string to uppercase letters.
- The `forward_lower` method creates another instance of the `Simple` object and forwards the request received from the client to the new instance to convert the string to lowercase letters.

Figure 4-2 shows the operation of the simpapp_mt sample application, employing both the thread-per-object and thread-per-request threading models.

Figure 4-2 simpapp_mt Sample Application



OMG IDL Code for the Simpapp Multithreaded Sample Application

The simpapp multithreaded sample application described in this chapter implements the CORBA interfaces listed in the following table.

Interface	Description	Action
`SimplePerRequestFactory`	Creates object references to the `Simple` object	`find_simple()`
`SimplePerObjectFactory`	Creates object references to the `Simple` object	`find_simple()`

`Simple`	Converts the case of a string	`to_upper()`
		`to_lower()`
		`forward_upper()`
		`forward_lower()`

Listing 4-2 shows the content of the `simple.idl` file, describing the CORBA interface in the simpapp_mt sample application.

Listing 4-2 OMG IDL Code for the simpapp_mt Sample Application

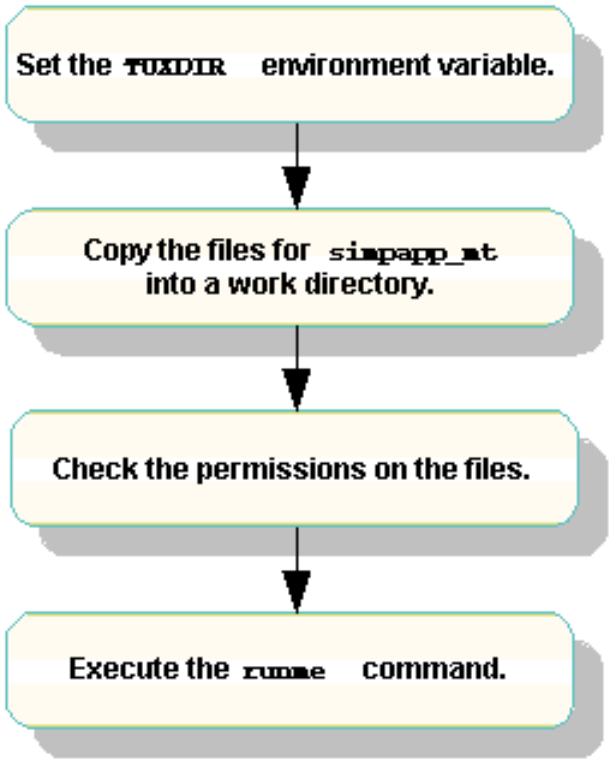
```
#pragma prefix "beasys.com"

interface Simple
{
     //Convert a string to lower case (return a new string)
     string to_lower(in string val);

     //Convert a string to upper case (in place)
     string to_upper(in string val);

     //Use other server to convert string to lower case
     string forward_lower(in string val);

     //Use other server to convert string to upper case
     string forward_upper(in string val);
};

interface SimplePerRequestFactory
{
     Simple find_simple();
```

```
};
interface SimplePerObjectFactory
{
     Simple find_simple();
};
```

How to Build and Run the Sample Application

This section leads you, step-by-step, through the process of building and running the simpapp_mt sample application. The flowchart summarizes the process and following sections explain how to perform the tasks.

Figure 4-3 Process for Building and Running simpapp_mt



Setting the TUXDIR Environment Variable

Before building and running the simpapp_mt sample application, ensure that the TUXDIR environment variable is set on your system. Typically, the environment variable is set during the installation process. You should confirm that the environment variable defines the correct directory location.

The TUXDIR environment variable must be set to the directory path where you installed the BEA Tuxedo software. For example:

Windows

```
TUXDIR=D:\TUXDIR
```

UNIX

```
TUXDIR=/usr/local/TUXDIR
```

Verifying the TUXDIR Environment Variable

Before you run the application, perform the following procedure to ensure that the environment variable contains the correct information.

Windows

Execute the echo command to show the setting of TUXDIR:

```
prompt> echo %TUXDIR%
```

UNIX

1. Execute the ksh command at the prompt to launch the Korn shell.
2. Execute the printenv command to show the setting of TUXDIR:

   ```
   ksh prompt> printenv TUXDIR
   ```

Changing the Setting of the Environment Variable

To change the value of the environment variable:

Windows

Execute the set command to set a new value for TUXDIR:

```
prompt> set TUXDIR=directorypath
```

UNIX

1. At the system prompt, execute the `ksh` command to launch the Korn shell.
2. At the `ksh` prompt, enter the `export` command to set the value for the `TUXDIR` environment variable:

   ```
   ksh prompt> export TUXDIR=directorypath
   ```

Creating a Working Directory for the Sample Application

Note: The technique of using a work directory is recommended so that you can see what additional files are created when you run the simpapp multithreaded sample. After you execute the `runme` command, compare the set of files in the installation directory to the set of files in your work directory.

The files required for the simpapp multithreaded sample application are in the following directories:

Windows

```
%TUXDIR%\samples\corba\simpapp_mt
```

UNIX

```
$TUXDIR/samples/corba/simpapp_mt
```

Create a working directory containing all of the simpapp multithreaded files.

Windows

You can use Windows Explorer to create a copy of the simpapp_mt directory, or you can use the command prompt as follows:

1. Create a target working directory for a copy of the simpapp_mt files.

   ```
   > mkdir work_directory
   ```

2. Copy the `simpapp_mt` files to the working directory.

   ```
   > copy %TUXDIR%\samples\corba\simpapp_mt\* work_directory
   ```

3. Change to the working directory.

   ```
   cd work_directory
   ```

4. List all the files in the working directory.

```
prompt> dir
```

```
makefile.mk             simple_per_object_i.h
makefile.nt             simple_per_object_server.cpp
Readme.txt              simple_per_request_i.cpp
runme.cmd               simple_per_request_i.h
runme.ksh               simple_per_request_server.cpp
simple.idl              simple_per_request_server.h
simple_client.cpp       thread_macros.cpp
simple_per_object_i.cpp thread_macros.h
```

UNIX

You can use your user interface tool to create a copy of the simpapp_mt directory, or you can use the command prompt as follows:

1. Create a target working directory for a copy of the `simpapp_mt` files.

```
> mkdir work_directory
```

2. Copy all `simpapp_mt` files to the working directory.

```
> cp $TUXDIR/samples/corba/simpapp_mt/* work_directory
```

3. Change to the working directory.

```
cd work_directory
```

4. List all the files in the working directory.

```
$ ls
```

```
makefile.mk             simple_per_object_i.h
makefile.nt             simple_per_object_server.cpp
Readme.txt              simple_per_request_i.cpp
runme.cmd               simple_per_request_i.h
runme.ksh               simple_per_request_server.cpp
simple.idl              simple_per_request_server.h
simple_client.cpp       thread_macros.cpp
simple_per_object_i.cpp thread_macros.h
```

Table 4-1 lists and describes the simpapp_mt files used to build and run the application.

Table 4-1 simpapp_mt Files

File	Description
`makefile.mk`	(UNIX) Makefile for the simpapp_mt sample application. Use this file to build the application.
`makefile.nt`	(Windows) Makefile for the simpapp_mt sample application. Use this file to build the application.
`Readme.txt`	`Readme` file that provides information about building and running the simpapp_mt sample application.
`runme.cmd`	(Windows) Command file for building and running the simpapp_mt sample application.
`runme.ksh`	(UNIX) Korn shell script for building and running the simpapp_mt sample application.
`simple.idl`	Object Management Group (OMG) Interface Definition Language (IDL) code that declares the `SimplePerRequestFactory`, `SimplePerObjectFactory`, and `Simple` interfaces.
`simple_client.cpp`	CORBA client program source code for the simpapp_mt sample application.
`simple_per_object_i.cpp`	Source code that includes implementations for `Simple` and `SimplePerObjectFactory` servants that are to be included in a server. The CORBA server is started using a thread-per-object concurrency strategy.
`simple_per_object_i.h`	Source code file for declaring `Simple` and `SimplePerObjectFactory` servants to be included in a server.

Table 4-1 simpapp_mt Files (Continued)

File	Description
simple_per_object_server.cpp	CORBA server program source code for the simpapp_mt sample application, thread-per-object concurrency strategy. Set CONCURR_STRATEGY = PER_OBJECT in the UBBCONFIG file.
simple_per_request_i.cpp	Source code that includes implementations for Simple and SimplePerRequestFactory servants that are to be included in a reentrant server. The reentrant CORBA server is started using a thread-per-request concurrency strategy.
simple_per_request_i.h	Source code file for declaring Simple and SimplePerRequestFactory servants to be included in a reentrant server.
simple_per_request_server.cpp	CORBA server program source code for the simpapp_mt sample application, thread-per-request concurrency strategy. Set CONCURR_STRATEGY = PER_REQUEST in the UBBCONFIG file.
simple_per_request_server.h	An example of a bootserverclass.h file, containing the declarations required for the user-defined Server class in the simpapp_mt sample application.
thread_macros.cpp	Platform-independent thread convenience macros that support the simpapp_mt sample application.
thread_macros.h	Source code file for declaring all the classes and variables for thread convenience macros.

Checking Permissions on All the Files

To build and run the simpapp_mt sample application, you must have user and read permissions on all the files you copied into your working directory. Check the permissions, and change the permissions if required.

Note: Ensure that the `make` utility is in your path.

Windows

```
> attrib -R /S *.*
```

UNIX

```
> /bin/ksh
> chmod u+r work_directory/*.*
```

Executing the runme Command

This section describes the steps required to execute the application end-to-end. Enter the `runme` command as follows:

Windows

```
> cd work_directory
> ./runme
```

UNIX

```
> /bin/ksh
> cd work_directory
> ./runme.ksh
```

The `runme` command automates the following steps:

1. Checks the `TUXDIR` environment variable.
2. Sets the environment variables that are used by this application.
3. Ensures that the proper `bin` directories are in the `PATH`.
4. If this is not the first time this script has been run, removes unneeded files from the directory.
5. Creates a directory to capture the results from running this script.
6. Creates a `setenv.ksh` file (UNIX) or `setenv.bat` file (Windows) so that you can build and run this sample step-by-step.
7. Creates the `ubb` configuration file for this sample.
8. Creates a file containing the user input for the client.

9. Creates a file with the expected output from the client.
10. Builds the sample.
11. Loads the configuration file.
12. Starts the thread-per-object server.
13. Starts the thread-per-request server.
14. Runs the client and captures the output.
15. Compares the output with the expected output.
16. Shuts down the server application.
17. Captures logs that are generated when you run the sample.
18. Saves the results.
19. Informs the user whether the sample ran successfully.

The simpapp_mt sample application prints the following messages while executing the `runme` command:

```
Testing simpapp_mt
    cleaned up
    prepared
    built
    loaded ubb
    booted
    ran
    shutdown
    saved results
  PASSED
```

The entire run-time output for the simpapp_mt sample application is stored in the results directory in your working directory. To see the output created at run time, examine the following files:

- `log`—compile, server boot, or server shutdown errors
- `output`—client application output and exceptions
- `ULOG.`*`date`*—server application errors and exceptions

Table 4-2 and Table 4-3 identify and describe the files created by executing the `runme` command.

Table 4-2 Files Created in the Working Directory

File	Description
`simple_c.cpp`	Created by the `idl` command for the `simple.idl` file. This module contains the client stub function for the `Simple` and `SimplePerRequestFactory` interface.
`simple_c.h`	Created by the `idl` command for the `simple.idl` file. This module contains definitions and prototypes for the `Simple` and `SimplePerRequestFactory` interfaces.
`simple_s.cpp`	Created by the `idl` command for the `simple.idl` file. This module contains the skeleton functions for the `Simple_i` and `SimplePerRequestFactory_i` implementations.
`simple_s.h`	Created by the `idl` command for the `simple.idl` file. This module contains definitions and prototypes for the skeleton classes for the `Simple_i` and `SimplePerRequestFactory_i` interfaces.
`simple_client`	Created by the `buildobjclient` command for the `simple_c.cpp` and `simple_client.cpp` files.
`simple_per_object_server`	Created by the `buildobjserver` command for the `simple_c.cpp`, `simple_s.cpp`, `simple_per_object_i.cpp`, `simple_per_object_server.cpp`, and `thread_macros.cpp` files.
`simple_per_request_server`	Created by the `buildobjserver` command for the `simple_c.cpp`, `simple_s.cpp`, `simple_per_request_i.cpp`, `simple_per_request_server.cpp`, and `thread_macros.cpp` files.

Table 4-2 Files Created in the Working Directory (Continued)

File	Description
`results` directory	Created by the `runme` command to capture the results from running this script.
`adm` directory	Created by the `runme` command to contain the security encryption key database file.

Table 4-3 Files Created in the Results Directory

Files	Description
`input`	Created by the `runme` command to store the input that the `runme` command provides to the C++ client application.
`output`	Created by the `runme` command to contain the output when the `runme` command executes the C++ client application.
`expected_output`	Created by the `runme` command to contain the expected output when the `runme` command is executed. The output file is compared to determine whether the test passed or failed.
`log`	Created by the `runme` command to contain the output generated by the `runme` command. If the command fails, check this file and the `ULOG` file for errors.
`setenv.cmd`	(Windows) Command file to set up environment variables required to build and run the simpapp_mt sample application step-by-step.
`setenv.ksh`	(UNIX) Command file to set up environment variables required to build and run the simpapp_mt sample application step-by-step.
`stderr`	Contains messages generated by `tmboot.` If the `-noredirect` server option is specified in the `UBBCONFIG` file, the `fprintf` method sends the output to this file.

Table 4-3 Files Created in the Results Directory (Continued)

Files	Description
stdout	Contains messages generated by tmboot. If the -noredirect server option is specified in the UBBCONFIG file, the fprintf method sends the output to this file.
tmsysevt.dat	Generated by the tmboot command in the runme command. It contains filtering and notification rules used by the TMSYSEVT process.
tuxconfig	A binary version of the configuration file.
ubb	UBBCONFIG file for the simpapp_mt sample application.
ULOG.*date*	ULOG file for storing run-time errors.

Running the Sample Application Step-by-Step

This section explains how to run the simpapp_mt sample application in step-by-step mode. You must execute the runme command before running simpapp_mt in step-by-step mode.

Follow the numbered steps to run the simpapp_mt application:

1. Set the environment variables.

   Windows

   ```
   > ..\results\setenv
   ```

   UNIX

   ```
   > ../results/setenv.ksh
   ```

2. Execute tmboot -y to launch the application. Information similar to the following is displayed:

```
>tmboot -y
Booting all admin and server processes in /work_directory/results/tuxconfig

Booting admin processes ...

exec BBL -A : process id=212 ... Started.

Booting server processes ...
```

```
exec TMSYSEVT -A : process id=289 ... Started.
exec TMFFNAME -A -- -N -M : process id=297 ... Started.
exec TMFFNAME -A -- -N : process id=233 ... Started.
exec TMFFNAME -A -- -F : process id=265 ... Started.
exec simple_per_object_server -A : process id=116 ... Started.
exec simple_per_request_server -A : process id=127 ... Started.
exec ISL -A -- -n //MrBeaver:2468 : process id=270 ... Started.
7 processes started.
>
```

Table 4-4 describes the server processes started by `tmboot`.

Table 4-4 Server Processes Started by tmboot

Process	Description
`TMSYSEVT`	System EventBroker.
`TMFFNAME`	`TMFFNAME` server processes: • `Master NameManager`—`TMFFNAME` server process started when you specify both the `-N` option and the `-M` option. • `SLAVE NameManager`—`TMFFNAME` server process started when you specify only the `-N` option. • `FactoryFinder` object—a `TMFFNAME` server process started with the `-F` option contains this object.
`simple_per_object_server`	Started as a thread-per-object server.
`simple_per_request_server`	Started as a reentrant thread-per-request server.
`ISL`	IIOP listener process.

3. Execute the client application.

   Windows

   ```
   > .\simple_client
   ```

   UNIX

   ```
   > ./simple_client
   ```

When you execute the client application, messages similar to the following are displayed:

Listing 4-3 Messages Displayed When simpapp_mt Client Is Executed

```
Number of simultaneous requests to post (1-50)?
String to convert using thread-per-request server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #2 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #3 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
String to convert using thread-per-object server?
Sending 4 deferred forward_lower requests
forward_lower request #0
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #1
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #2
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
forward_lower request #3
returned:aabbccddeeffgghhiijjkkllmmnnooppqqrrssttuuvvwwxxyyzz
Sending 4 deferred forward_upper requests
forward_upper request #0 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #1 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
forward_upper request #2 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ

forward_upper request #3 returned:
AABBCCDDEEFFGGHHIIJJKKLLMMNNOOPPQQRRSSTTUUVVWWXXYYZZ
```

Shutting Down the Sample Application

Before running another sample application, you should shut down the simpapp_mt sample application and eliminate all unwanted files from the working directory.

1. To end the application, run the `tmshutdown -y` command. Information similar to the following is displayed:

```
>tmshutdown -y
Shutting down all admin and server processes in
/work_directory/results/tuxconfig

Shutting down server processes ...

Server Id = 5 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 2 Group Id = APP_GRP2 Machine = SITE1:    shutdown succeeded.
Server Id = 4 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 3 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 2 Group Id = SYS_GRP Machine = SITE1:     shutdown succeeded.
Server Id = 1 Group Id = SYS_GRP Machine = SITE1:       shutdown succeeded.

Shutting down admin processes ...

Server Id = 0 Group Id = SITE1 Machine = SITE1: shutdown succeeded.
7 processes stopped.
```

2. Restore the working directory to its original state.

   Windows

   ```
   > ..\results\setenv
   > make -f clean
   ```

   UNIX

   ```
   > ../results/setenv.ksh
   > make -f makefile.mk clean
   ```

Multithreaded CORBA Server Application Administration

This topic includes the following sections:

- Specifying Thread Pool Size
- Specifying a Threading Model
- Specifying the Number of Active Objects
- Sample UBBCONFIG File

Specifying Thread Pool Size

The `MAXDISPATCHTHREADS` and `MINDISPATCHTHREADS` parameters for specifying the maximum and minimum sizes of the thread pool are in the `SERVERS` section of the `UBBCONFIG` file. For examples of how to specify these parameters, see Listing 4-4. A multithreaded CORBA application uses these values to create and manage the thread pool.

MAXDISPATCHTHREADS

The `MAXDISPATCHTHREADS` parameter determines the maximum number of concurrently dispatched threads that each server process can spawn. When specifying this parameter, consider the following:

- The value for `MAXDISPATCHTHREADS` determines the maximum size the thread pool can grow to be, as it increases in size to accommodate incoming requests.
- The default value for `MAXDISPATCHTHREADS` is 1. If you specify a value greater than 1, the system creates and uses a special dispatcher thread. This dispatcher thread is not included in the number of threads determining the maximum size of the thread pool.

  Note: If you specify a value greater than 1 for `MAXDISPATCHTHREADS` and do not supply a value for the `CONCURR_STRATEGY` threading model parameter, the threading model for the application defaults to thread-per-object. For a discussion of the `CONCURR_STRATEGY` threading model parameter, see "Specifying a Threading Model" on page 4-35.

- Specifying a value of 1 for the `MAXDISPATCHTHREADS` parameter indicates that the CORBA server application should be configured as a single-threaded server.

  Note: When you build a multithreaded CORBA server application specifying `buildobjserver -t`, that server is capable of running in multithreaded mode. To run as a multithreaded CORBA server application, the `MAXDISPATCHTHREADS` parameter

in the UBBCONFIG file must be set to a value greater than 1; if it is not, the server application will run in single-threaded mode.

- The value you specify for the MAXDISPATCHTHREADS parameter must not be less than the value you specify for the MINDISPATCHTHREADS parameter.
- The operating system resources limit the maximum number of threads that can be created in a process. MAXDISPATCHTHREADS should be less than that limit, minus the number of application managed threads that your application requires.

The value of the MAXDISPATCHTHREADS parameter affects other parameters. For example, the MAXACCESSORS parameter controls the number of simultaneous accesses to the BEA Tuxedo system, and each thread counts as one accessor. For a multithreaded server application, you must account for the number of system-managed threads that each server is configured to run. A system-managed thread is a thread that is started and managed by the BEA Tuxedo software, as opposed to threads started and managed by an application. Internally, BEA Tuxedo manages a pool of available system-managed threads. When a client request is received, an available system-managed thread from the thread pool is scheduled to execute the request. When the request is completed, the system-managed thread is returned to the pool of available threads.

For example, if that you have 4 multithreaded servers in your system and each server is configured to run 50 system-managed threads, the accessor requirement for these servers is the sum total of the accessors, calculated as follows:

```
50 + 50 + 50 + 50 = 200 accessors
```

MINDISPATCHTHREADS

Use the MINDISPATCHTHREADS parameter to specify the number of server dispatch threads that are started when the server is initially booted. When you specify this parameter, consider the following:

- The value for MINDISPATCHTHREADS determines the initial allocation of threads in the thread pool.
- The separate dispatcher thread that is created when MAXDISPATCHTHREADS is greater than 1 is not counted as part of the MINDISPATCHTHREADS limit.
- The value you specify for MINDISPATCHTHREADS must not be greater than the value you specify for MAXDISPATCHTHREADS.
- The default value for MINDISPATCHTHREADS is 0.

Specifying a Threading Model

To specify a threading model, you set the CONCURR_STRATEGY parameter which is defined in the SERVERS section of the UBBCONFIG file.

Use the CONCURR_STRATEGY parameter to specify the threading model a multithreaded CORBA server application is to use. The CONCURR_STRATEGY parameter accepts either of these values:

- CONCURR_STRATEGY = PER_REQUEST
- CONCURR_STRATEGY = PER_OBJECT

When you specify CONCURR_STRATEGY = PER_REQUEST to employ the thread-per-request model, each invocation on the CORBA server application is assigned to an arbitrary thread from the threads pool.

When you specify CONCURR_STRATEGY = PER_OBJECT to employ the thread-per-object model, each active object is associated with a single thread at any one time. Each request for an object establishes an association between a dispatch thread and the object.

If the value for MAXDISPATCHTHREADS is greater than one and you do not specify a value for CONCURR_STRATEGY, the threading model is set to PER_OBJECT.

For more information on the characteristics of threading models, see "Threading Models" on page 4-5.

Specifying the Number of Active Objects

Use the MAXOBJECTS parameter to specify the maximum number of objects per machine to be accommodated in the Active Object Map tables in the bulletin board. You can set this value in either the RESOURCES section or the MACHINES section of the configuration file. The MAXOBJECTS number in the RESOURCES section is a system-wide setting. Use the MAXOBJECTS number in the MACHINES section to override the system-wide setting on a per-machine basis.

For a system-wide setting, specify:

```
*RESOURCES
        MAXOBJECTS number
```

To override a system-wide setting for a specific machine, specify:

```
*MACHINES
        MAXOBJECTS = number
```

The value for *number* is limited only by the resources of your operating system.

Sample UBBCONFIG File

Listing 4-4 shows a the UBBCONFIG file for the BEA Tuxedo Threads sample application. The threads-related parameters are presented in **boldface** text.

Note: The value for the MAXOBJECTS parameter affects the operation of a multithreaded server. However, this parameter is not specific to multithreaded servers, since it also affects the operation of single-threaded servers. Increasing the value for MAXOBJECTS results in the consumption of additional system resources for any server.

Listing 4-4 Threads Sample Application UBBCONFIG File

```
*RESOURCES
     IPCKEY     55432
     DOMAINID   simpapp
     MAXOBJECTS 100
     MASTER     SITE1
     MODEL      SHM
     LDBAL      N

*MACHINES
     "sunstar"
     LMID         = SITE1
     APPDIR       = "/rusers1/lyon/samples/corba/simpapp_mt"
     TUXCONFIG    = "/rusers1/lyon/samples/corba/simpapp_mt/results/tuxconfig"
     TUXDIR       = "/usr/local/TUXDIR"
     MAXWSCLIENTS = 10
     MAXACCESSERS = 200

*GROUPS
     SYS_GRP
        LMID    = SITE1
        GRPNO   = 1
     APP_GRP1
        LMID    = SITE1
        GRPNO   = 2
     APP_GRP2
        LMID    = SITE1
        GRPNO   = 3

*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"
   simple_per_object_server
      SRVGRP  = APP_GRP1
      SRVID   = 1
      MINDISPATCHTHREADS = 10
      MAXDISPATCHTHREADS = 100
      CONCURR_STRATEGY = PER_OBJECT
      RESTART = N
   simple_per_request_server
      SRVGRP  = APP_GRP2
      SRVID   = 2
      MINDISPATCHTHREADS = 10
      MAXDISPATCHTHREADS = 100
      CONCURR_STRATEGY = PER_REQUEST
      RESTART = N
   ISL
      SRVGRP  = SYS_GRP
      SRVID   = 5
   CLOPT   = "-A -- -n //sunbstar:2468 -d /dev/tcp"

*SERVICES
```

CHAPTER 5

Security and CORBA Server Applications

This chapter discusses security and CORBA server applications, using the Security University sample application as an example. The Security sample application implements a security model that requires student users of the University sample application to be authenticated as part of the application login process.

This topic includes the following sections:

- Overview of Security and CORBA Server Applications
- Design Considerations for the University Server Application

Overview of Security and CORBA Server Applications

Generally, CORBA server applications have little to do with security. Security in the BEA Tuxedo domain is specified by the system administrator in the `UBBCONFIG` file, and client applications are responsible for logging on, or authenticating, to the domain. None of the security models supported in the BEA Tuxedo system make any requirements on server applications running in the BEA Tuxedo domain.

However, there may be occasions when implementing or enhancing a security model in your CORBA application involves adding objects, or adding operations to existing objects, that are managed by the server application.

This chapter shows how the University server application is enhanced to add the notion of a student, which is incorporated into the client application as a means to identify, and log in, users of the client application.

For information about how client applications are authenticated into the BEA Tuxedo domain, see *Creating CORBA Client Applications*. For information about implementing a security model in the BEA Tuxedo domain, see *Setting Up a BEA Tuxedo Application*.

Design Considerations for the University Server Application

The design rationale for the Security University sample application is to require users of the client application to log on before they can do anything. The Security sample application, therefore, needs to define the notion of a user.

To log on to the application, the client application needs to provide the following to the security service in the BEA Tuxedo domain (note that the student user of the application provides only the username and application password):

- Client name
- Username
- An application password

The Security sample application adds an operation, `get_student_details()`, to the `Registrar` object. This operation enables the client application to obtain information about each student user from the University database after the client application is logged on to the BEA Tuxedo domain.

Note: The `get_student_details()` operation has nothing to do with implementing a security model in the BEA Tuxedo domain. The addition of this operation is only a supplemental feature added to the Security sample application. For details about the security model added to the Security sample application, and how client applications log on to the Security server application, see *Creating CORBA Client Applications*.

The sections that follow explain:

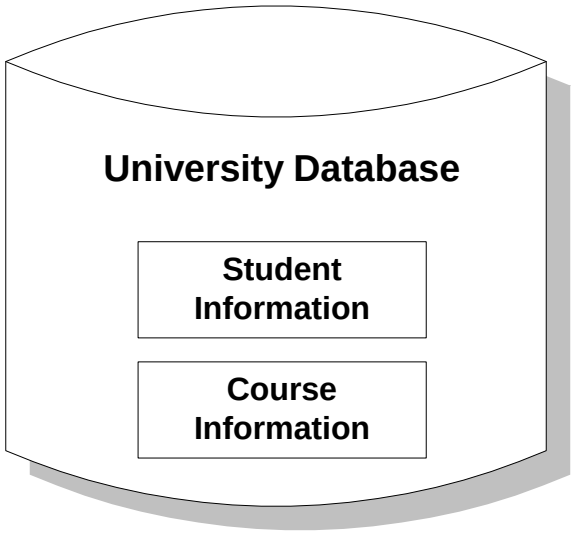
- How the Security University sample application works
- Design considerations for returning student details to the client application

How the Security University Sample Application Works

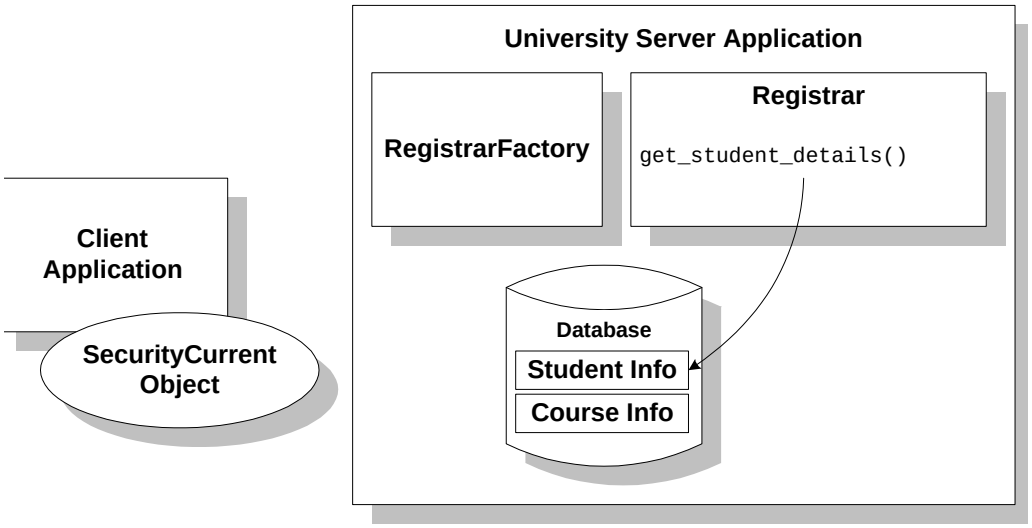
To implement the Security sample application, the client application adds a logon dialog with the student end user. This dialog uses the local SecurityCurrent object on the client machine to invoke operations on the PrincipalAuthenticator object, which is part of logging on to access the BEA

Tuxedo domain. After the user authentication process, the client application invokes the `get_student_details()` operation on the `Registrar` object to obtain information about each student user.

The University database used in the Security sample application is updated to contain student information in addition to course information, and is shown in the following figure:



The `get_student_details()` operation accesses the student information portion of the database to obtain student information needed by the client logon operation. The following figure shows the primary objects involved in the Security sample application:



A typical usage scenario of the Security sample application may include the following sequence of events:

1. The client application obtains a reference to the SecurityCurrent object from the Bootstrap object.
2. The client application invokes the SecurityCurrent object to determine the level of security that is required by the BEA Tuxedo domain.
3. The client application queries the student user for a student ID and the required passwords.
4. The client application authenticates the student by obtaining information about the student from the Authentication Service.
5. If the authentication process is successful, the client application logs on to the BEA Tuxedo domain.
6. The client application invokes the `get_student_details()` operation on the `Registrar` object, passing a student ID, to obtain information about the student.
7. The `Registrar` object scans the database for student information that matches the student ID in the client request.

8. If there is a match between the student ID provided in the client application request and the student information in the database, the `Registrar` object returns the `struct StudentDetails` to the client application. (If the student enters an ID that does not match the information in the database, the `Registrar` object returns a CORBA exception to the client application.)
9. If the `Registrar` object returns `StudentDetails` to the client application, the client application displays a personalized welcome message to the student user.

Design Considerations for Returning Student Details to the Client Application

The client application needs to provide a means by which to log a user on to the BEA Tuxedo system so that the user can continue to use the University application. To do this, the client application needs an identity for the user. In the Security sample application, this identity is the student ID.

All that is required of the University server application is to return data about a student, based on the student ID, so that the client application can complete the user authentication process. Therefore, the OMG IDL for the Security sample application adds the definition of the `get_student_details()` operation to the `Registrar` object. The primary design consideration for the University server application is based on the operational scenario described earlier; namely, that one student interacts with one client application at one time, so there is no need for the server application to deal with a sizable batch of data to implement the `get_student_details()` operation.

The `get_student_details()` operation has the following OMG IDL definition:

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

CHAPTER 6

Integrating Transactions into a CORBA Server Application

This chapter describes how to integrate transactions into a CORBA server application, using the Transactions University sample application as an example. The Transactions sample application encapsulates the process of a student registering for a set of courses. The Transactions sample application does not show all the possible ways to integrate transactions into a CORBA server application, but it does show two models of transactional behavior, showing the impact of transactional behavior on the application in general and on the durable state of objects in particular.

This topic includes the following sections:

- Overview of Transactions in the BEA Tuxedo System
- Designing and Implementing Transactions in a CORBA Server Application
- Integrating Transactions in a CORBA Client and Server Application. This section describes:
  - Making an Object Automatically Transactional
  - Enabling an Object to Participate in a Transaction
  - Preventing an Object from Being Invoked While a Transaction Is Scoped
  - Excluding an Object from an Ongoing Transaction
  - Assigning Policies
  - Opening an XA Resource Manager
  - Closing an XA Resource Manager

- Transactions and Object State Management
- Notes on Using Transactions in the BEA Tuxedo System
- User-defined Exceptions

This chapter also presents a section on user-defined exceptions. The Transactions sample application introduces a user-defined exception, which can be returned to the client application and that potentially causes a client-initiated transaction to be rolled back.

Overview of Transactions in the BEA Tuxedo System

The BEA Tuxedo system provides transactions as 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. That is, you have a requirement to perform multiple write operations on durable storage, and you must be guaranteed that the operations succeed; if any one of the operations fails, the entire set of operations is rolled back.

Transactions typically are appropriate in the situations described in the following list. Each situation encapsulates a transactional model supported by the BEA Tuxedo system.

- The client application needs to make invocations on several different 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 so far. For example, if the client application cannot book a flight from Los Angeles to Honolulu on a given date, the client application needs to cancel the flight reservations made up to that point.

- The client needs a conversation with an object managed by the server application, and the client 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 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 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 application. The user of the client application browses through an online catalog and makes multiple purchase selections. When the user is done choosing all the items he or she wants to buy, the user clicks on a button to make the purchase, where the user may enter credit card information. If the credit card check fails (for example, the user cannot provide valid credit card information), the shopping application needs a way to cancel all the pending purchase selections 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. (And in this situation, the individual database edits are not necessarily CORBA invocations.)

  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.

Designing and Implementing Transactions in a CORBA Server Application

This section explains how to design and implement transactions in a CORBA server application using the Transactions University sample application as an example. This section also describes how the Transactions sample application works, and discusses the design considerations for implementing transactions in it. For additional general information about transactions, see the section “Integrating Transactions in a CORBA Client and Server Application” on page -9.

The Transactions sample application uses transactions to encapsulate the task of a student registering for a set of courses. The transactional model used in this application is a combination

of the conversational model and the model in which a single invocation makes multiple individual operations on a database, as described in the preceding section.

The Transactions sample application builds on the Security sample application by adding the following capabilities:

- Students can submit a list of courses for which they want to register. (Each course is represented by a number.)
- For each course in the list, the University server application checks the following:
  - Whether the course is in the University database
  - Whether the student is already registered for the course
  - Whether the student exceeds the maximum number of credits he or she can take
- If the course passes the checks in the preceding list, the University server application registers the student for the course.
- If the server application cannot register the student for a course because the course does not exist in the database or because the student is already registered for the course, the server application returns to the client application a list of courses for which the registration process failed. The client application can then choose whether to commit the transaction to register the student for the courses for which the registration process succeeds, or to roll back the entire transaction.
- If a course registration fails because the student exceeds the maximum number of credits he or she can take, the server application returns a CORBA exception to the client application that provides a brief message explaining why the registration for the course was not successful. (The server application does not mark the transaction for rollback only.)

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.

The sections that follow explain:

- How the Transactions University Sample Application Works
- Transactional Model Used by the Transactions University Sample Application
- Object State Considerations for the University Server Application
- Configuration Requirements for the Transactions Sample Application

How the Transactions University Sample Application Works

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 the section "Overview of Transactions in the BEA Tuxedo System" on page 6-2. 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; that is, 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 (for example, Oracle) 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 an Oracle database, you must include specific Oracle-provided files in the server application build process.

For details about building, configuring, and running the Transactions sample application, see the *Guide to the CORBA University Sample Applications*. This online document also contains the `UBBCONFIG` files for each sample application and explains the entries in that file.

Integrating Transactions in a CORBA Client and Server Application

The BEA Tuxedo system 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 information about the TransactionCurrent object, see *Creating CORBA Client Applications* and *Using CORBA Transactions*.
- 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 the `rollback_only()` operation 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 the `rollback_only()` operation 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 vetos the current transaction, the transaction is committed.

The following sections explain how you can use object activation policies and transaction policies to get 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 6-13.

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.

Assigning the `always` transactional policy to an object's interface is appropriate 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 (ICF file):

Activation Policy	Transaction Policy
`process`, `method`, or `transaction`	`always`

Note: Database cursors cannot span transactions. The `CourseSynopsisEnumerator` object in the CORBA 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 specified in the ICF file for that object's interface, to make an object optionally transactional:

Activation Policy	Transaction Policy
`process`, `method`, or `transaction`	`optional`

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: Some XA resource managers used in the BEA Tuxedo system require that any object participating in a transaction scope their database read operations, in addition to write operations, within a transaction. (However, you can still scope your own transactions.) For example, using an Oracle TMS with the BEA Tuxedo system has this requirement. 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.

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. The BEA Tuxedo system 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; for example, 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 ICF file:

Activation Policy	Transaction Policy
`process` or `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. The BEA Tuxedo system 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 ICF file:

Activation Policy	Transaction Policy
`process` or `method`	`ignore`

Assigning Policies

For information about how to create an ICF file and specify policies on objects, see the section "Step 4: Define the In-memory Behavior of Objects" on page 2-12.

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

If you need transactions in your CORBA client and server application, you can integrate transactions with object state management in a few different ways. In general, the BEA Tuxedo system 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.

The following sections address some key points regarding transactions an object state management.

Delegating Object State Management to an XA Resource Manager

Using an XA resource manager, such as Oracle which is used in the CORBA University sample applications, generally simplifies the design problems associated with handling object state data in the event of a rollback. Transactional objects can always delegate the commit and rollback responsibilities to the XA resource manager, which greatly eases 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, the BEA Tuxedo system 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_ABORT`. If the variable is `DR_TRANS_COMMITTING`, the object can invoke its database write operations. If the variable is `DR_TRANS_ABORT`, the object skips its write operations.

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

- You want the object to write its durable 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 the BEA Tuxedo system passes the reason `DR_TRANS_COMMITTING`, the object can, if necessary, invoke the `rollback_only()` operation on the TransactionCurrent object. Note that if you do make an invocation to the `rollback_only()` operation from within the `Tobj_ServantBase::deactivate_object()` operation, the `Tobj_ServantBase::deactivate_object()` operation is not invoked again.

- 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.

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 ICF 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 the `Tobj_ServantBase::deactivate_object()` operation 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, respectively, 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 the sections "Opening an XA Resource Manager" on page 6-13 and "Closing an XA Resource Manager" on page 6-13.

Notes on Using Transactions in the BEA Tuxedo System

Note the following about integrating transactions into your CORBA client/server applications:

- The following transactions are not permitted in the BEA Tuxedo system:
  - Nested transactions

    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.
  - Recursive transactions

    A transactional object cannot call a second object, which in turn calls the first object.
- 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. 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.

- Objects can be involved with only one transaction at one time. The BEA Tuxedo system does not support concurrent transactions.
- The BEA Tuxedo system 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:

  `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:

  `CORBA::INVALID_TRANSACTION`
- For transaction-bound objects, you might consider doing all state handling in the `Tobj_ServantBase::deactivate_object()` operation. This makes it easier for the object to handle its state properly, since the outcome of the transaction is known at the time that the `Tobj_ServantBase::deactivate_object()` operation is invoked.
- For method-bound objects that have several operations, but only a few that affect the object's durable state, you may want to consider 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 an exception is raised inside an operation on that object, the client application receives an `OBJ_ADAPTER` exception. In this situation, 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 and returns a CORBA exception, the client application receives only a transaction rollback exception but not the CORBA exception.

Note: In the WebLogic Enterprise version 4.2 software, no workaround exists for this situation. We recommend that applications perform as much data validation as possible before starting a transaction.

- Note the following restriction on a transactional object that has the `TP::deactivateEnable` method:

  If the `TP::deactivateEnable` method is invoked during a transaction, the object is deactivated *when the transaction ends*. However, if any methods are invoked on the object between the time that the `TP::deactivateEnable` method is called and the time that the transaction is committed, the object is never deactivated.

User-defined Exceptions

The Transactions sample 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 CORBA client/server application, using the `TooManyCredits` exception as an example.

Including a user-defined exception in a 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.

The sections that follow explain and give examples of the first two steps.

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 example.

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

CHAPTER 7

Wrapping a BEA Tuxedo Service in a CORBA Object

This chapter presents an overview of one way in which you can call a BEA Tuxedo service from within an object managed by a CORBA server application, using the Wrapper sample application as an example.

This topic includes the following sections:

- Overview of Wrapping a BEA Tuxedo Service

  This section describes:

  - Designing the Object That Wraps the BEA Tuxedo Service
  - Creating the Buffer in Which to Encapsulate BEA Tuxedo Service Calls
  - Implementing the Operations That Send Messages to and from the BEA Tuxedo Service

- Design Considerations for the Wrapper Sample Application

The Wrapper sample application delegates a set of billing operations to a BEA Tuxedo ATMI teller application, which contains a set of services that perform basic billing procedures. The approach in this chapter shows one technique for incorporating a BEA Tuxedo application into a BEA Tuxedo domain.

The examples shown in this chapter demonstrate a one-to-one relationship between operations on a CORBA object and calls to specific services within an application. In a sense, the calls to the BEA Tuxedo services are wrapped as operations on a CORBA object; thus, the object delegates its work to the BEA Tuxedo application. If you have a set of BEA Tuxedo services that you want to use in a CORBA server application, the technique shown in this chapter may work for you.

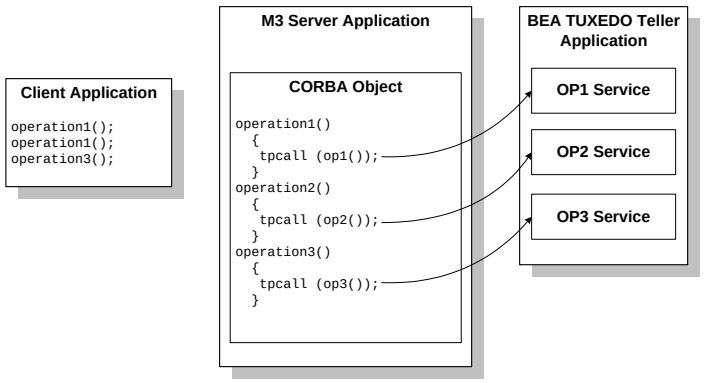
This chapter does not provide any details about BEA Tuxedo ATMI applications. For information about how to build and configure BEA Tuxedo ATMI applications, and for information about how they work, see the BEA Tuxedo ATMI information set, which is included in the BEA Tuxedo online documentation.

Overview of Wrapping a BEA Tuxedo Service

The process described in this chapter for wrapping a set of BEA Tuxedo services encompasses the following steps:

1. Designing the object that structures a set of tasks that are oriented to the BEA Tuxedo system as operations on that object.
2. Creating the message buffer used by the BEA Tuxedo services. You use this message buffer to send and receive messages to and from the BEA Tuxedo services. You can allocate the buffer in the object's constructor in the application's implementation file.
3. Implementing on the object the operations that send and receive messages to and from the BEA Tuxedo services. This step also includes choosing the implementation for how the BEA Tuxedo services are called.

The following figure shows a high-level view of the relationship among the client application, the CORBA object managed by the CORBA server application, and the BEA Tuxedo ATMI application that implements the services called from the CORBA object.



Designing the Object That Wraps the BEA Tuxedo Service

The first step described in this chapter is designing the object that wraps the calls to the BEA Tuxedo ATMI application. For example, the goal for the Wrapper sample application is to add billing capability to the student registration process, which can be done by delegating a set of billing operations to an existing BEA Tuxedo ATMI teller application.

The BEA Tuxedo ATMI teller application used by the Wrapper sample application contains the following services:

- `CURRBALANCE`—obtains the current balance of a given account
- `CREDIT`—credits an account by a given dollar amount
- `DEBIT`—debits an account by a given dollar amount

To wrap these services, the Wrapper sample application includes a separate OMG IDL file that defines a new interface, `Teller`, which has the following operations:

- `get_balance()`
- `credit()`
- `debit()`

Each of these operations on the `Teller` object maps one-to-one to calls on the services in the BEA Tuxedo ATMI teller application.

A typical usage scenario of the `Teller` object may be the following:

1. The client application invokes the `register_for_courses()` operation on the `Registrar` object, which requires a student ID.
2. As part of the registration process, the `Registrar` object invokes the `get_balance()` operation on the `Teller` object, passing an account number.
3. The `get_balance()` operation on the `Teller` object puts the account number into a message buffer and sends the buffer to the BEA Tuxedo ATMI teller application's `CURRBALANCE` service.
4. The BEA Tuxedo ATMI teller application receives the message buffer, extracts its contents, and makes the appropriate call to the `CURRBALANCE` service.
5. The `CURRBALANCE` service obtains from the University database the current balance of the account and gives it to the BEA Tuxedo ATMI teller application.
6. The BEA Tuxedo ATMI teller application inserts the current balance into a message buffer and returns it to the `Teller` object.

7. The `Teller` object extracts the current balance amount from the message buffer and returns the current balance to the `Registrar` object.

For more design information about the `Teller` object and the Wrapper sample application, see the section “Design Considerations for the Wrapper Sample Application” on page -7.

Creating the Buffer in Which to Encapsulate BEA Tuxedo Service Calls

The next step described in this chapter is creating the buffer within which messages are sent between the object and the BEA Tuxedo service. There are a number of buffer types that may be used by various BEA Tuxedo ATMI applications, and the examples used in this chapter are based on the FML buffer type. For more information about buffer types in the BEA Tuxedo system, see the BEA Tuxedo information set.

In your application implementation file, you need to allocate the chosen buffer type. You can allocate the buffer in the object’s constructor, because the buffer you allocate does not need to be unique to any particular `Teller` object instance. This allocation operation typically includes specifying the buffer type, passing any flags appropriate for the procedure call to the BEA Tuxedo service, and specifying a buffer size.

You also need to add to your implementation’s header file the definition of the variable that represents the buffer.

The following code example shows the constructor for the Wrapper application’s `Teller` object that allocates the BEA Tuxedo buffer, `m_tuxbuf`:

```
Teller_i::Teller_i() :
    m_tuxbuf((FBFR32*)tpalloc("FML32", "", 1000))
{
    if (m_tuxbuf == 0) {
        throw CORBA::INTERNAL();
    }
}
```

Note the following about the line that allocates the FML buffer:

Code	Description
`tpalloc`	Allocates the buffer.
`"FML32"`	Specifies the FML buffer type.
`""`	Typically enclose any flags passed to the BEA Tuxedo service. In this example, no flags are passed.
`1000`	Specifies the buffer size in bytes.

The object's implementation file should also deallocate the buffer in the destructor, as in the following statement from the Wrapper application implementation file:

```
tpfree((char*)m_tuxbuf);
```

Implementing the Operations That Send Messages to and from the BEA Tuxedo Service

The next step is implementing the operations on the object that wraps calls to the BEA Tuxedo ATMI application. In this step, you choose the implementation of how the BEA Tuxedo services are called from the object. The Wrapper sample application uses the `tpcall` implementation.

An operation on an object that wraps a BEA Tuxedo service typically includes statements that do the following:

- Fill the message buffer with the data that you want to send to the BEA Tuxedo service.
- Call the BEA Tuxedo service. The following arguments are included in the call:
  a. The BEA Tuxedo service that you want to invoke.
  b. The message buffer to be sent to the BEA Tuxedo service.
  c. The message buffer to be returned from the BEA Tuxedo service.
  d. The size of the buffer in which the BEA Tuxedo service response is to be placed.
- Extract the response from the BEA Tuxedo service.
- Return the results to the client application.

The following example shows the implementation of the `get_balance()` operation in the Wrapper application `Teller` object. This operation retrieves the balance of a specific account, and the BEA Tuxedo service being called is `CURRBALANCE`.

```
CORBA::Double Teller_i::get_balance(BillingW::AccountNumber account)
{
    // "marshal" the "in" parameters (account number)
    Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
    long size = Fsizeof32(tuxbuf);
    // Call the CURRBALANCE Tuxedo service
    if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
               (char**)&tuxbuf, &size, 0) ) {
        throw CORBA::PERSIST_STORE();
    }
    // "unmarshal" the "out" parameters (current balance)
    CORBA::Double currbal;
    Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
    return currbal;
}
```

The statement in the following code example fills the message buffer, `m_tuxbuf`, with the student account number. For information about FML, see the *BEA Tuxedo ATMI FML Function Reference*.

```
Fchg32(m_tuxbuf, ACCOUNT_NO, 0, (char*)&account, 0);
```

The following statement calls the `CURRBALANCE` BEA Tuxedo service, via the `tpcall` implementation, passing the message buffer. This statement also specifies where the BEA Tuxedo service response is to be placed, which in this example is also the same buffer as the one in which the request was sent.

```
if (tpcall("CURRBALANCE", (char*)tuxbuf, 0,
                (char**)&tuxbuf, &size, 0) ) {
        throw CORBA::PERSIST_STORE();
        }
```

The following statement extracts the balance from the returned BEA Tuxedo message buffer:

```
Fget32(m_tuxbuf, CURR_BALANCE, 0, (char*)&currbal, 0);
```

The last line in the `get_balance()` operation returns the results to the client application:

```
return currbal;
```

Restrictions

Note the following restrictions regarding how you can incorporate BEA Tuxedo services within a BEA Tuxedo domain:

- You may not combine object implementations and BEA Tuxedo services within the same server application. The BEA Tuxedo services may only exist within a separate BEA Tuxedo server application in the same domain as the CORBA server application.
- You may not include the `tpreturn()` or `tpforward()` BEA Tuxedo implementations within an object that calls a BEA Tuxedo service.

Design Considerations for the Wrapper Sample Application

The basic design considerations for the Wrapper sample application are based on the scenario that is described in this section. When a student registers for a course, the `Registrar` object performs, as part of its registration process, invocations to the `Teller` object, which charges the student's account for the course.

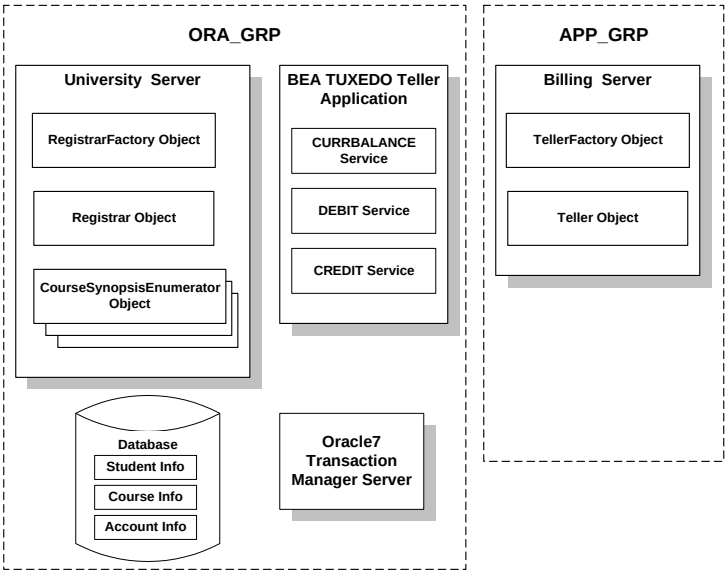
This section describes the design for the Wrapper sample application, which incorporates an additional server application, Billing, into the configuration. Therefore, the Wrapper sample application consists of the following four server applications:

- University, which has the `RegistrarFactory`, `Registrar`, and `CourseSynopsisEnumerator` objects
- Billing, which has the `TellerFactory` and `Teller` objects
- BEA Tuxedo ATMI Teller Application, which has the `CURRBALANCE`, `CREDIT`, and `DEBIT` services
- The Oracle Transaction Manager Server (TMS)

In addition, the `UBBCONFIG` file for the Wrapper sample application specifies the following groups:

- ORA_GRP, which contains the University server application, the BEA Tuxedo ATMI Teller application, and the Oracle TMS. Since these three processes are involved in transactions on the University database, they must all be in the same group, along with the database itself.
- APP_GRP, which contains the Billing server application. This application does not need to be in ORA_GRP, because this application does not interact with the University database.

The configuration of the BEA Tuxedo domain in the Wrapper sample application is shown in the following figure.



Incorporating a BEA Tuxedo ATMI application into the University sample applications makes sense from the standpoint of using the Process-Entity design pattern. BEA Tuxedo ATMI applications generally implement the Process-Entity design pattern, which are also used in the University sample applications.

The University database is updated to include a new table containing account information for each student. Therefore, when services in the BEA Tuxedo ATMI Teller Application process billing data, they perform transactions using the University database.

How the Wrapper University Sample Application Works

A typical usage scenario in the Wrapper sample application encompasses the following sequence of events:

1. After the student logon procedure, the client application invokes the `get_student_details()` operation on the `Registrar` object. Included in the implementation of the `get_student_details()` operation is code that retrieves:
   - The student's account number from the student table in the database

   – The student's balance from the account table in the database, which is obtained by invoking the `get_balance()` operation on the `Teller` object

2. The student browses courses, as with the Basic sample application, and identifies a list of courses for which he or she wants to register.

3. The client application sends a request to the `Registrar` object, as with the Transactions sample application scenario, to invoke the `register_for_courses()` operation. The request continues to include only a list of course numbers and a student ID.

4. While registering the student for the list of courses, the `register_for_courses()` operation invokes:

   – The `get_balance()` operation on the `Teller` object, to make sure that the student does not have a delinquent account

   – The `debit()` operation on the `Teller` object, which is managed by the Billing server application to bill for courses

5. The `get_balance()` and `debit()` operations on the `Teller` object each send a request to the BEA Tuxedo ATMI Teller application. Encapsulated in the request is an FML buffer containing the appropriate calls, including the account number calls to, respectively, the `CURRBALANCE` and `DEBIT` services in the BEA Tuxedo ATMI Teller application.

6. The `CURRBALANCE` and `DEBIT` services perform the appropriate database calls to, respectively, obtain the current balance and debit the student's account to reflect the charges for the courses for which he or she has registered.

   If the student has a delinquent account, the `Registrar` object returns the `DelinquentAccount` exception to the client application. The client application then rolls back the transaction.

   If the `debit()` operation fails, the `Teller` object invokes the `rollback_only()` operation on the TransactionCurrent object. Because the `Teller` and `Registrar` objects are scoped within the same transaction, this rollback affects the entire registration process and thus prevents the situation where there is an inconsistent database (showing, for example, that the student is registered for the course, but the student's account balance has not been debited for the course).

7. If no exceptions have been raised, the `Registrar` object registers the student for the desired courses.

Interface Definitions for the Billing Server Application

The following interface definitions are defined for the Billing server application:

- The `TellerFactory` object, whose only operation is `find_teller()`. The `find_teller()` operation works exactly the same as the `find_registrar()` operation in the University server `RegistrarFactory` object.
- The `Teller` object, which, as mentioned earlier, implements the following operations:
  - `debit()`
  - `credit()`
  - `current_balance()`

  Like the `Registrar` object, the `Teller` object has no state data and does not have a unique object ID (OID).

Additional Design Considerations for the Wrapper Sample Application

The following additional considerations influence the design of the Wrapper sample application:

- The `Registrar` object needs a way to send requests to the `Teller` object to handle billing operations.
- The University server application and the BEA Tuxedo ATMI teller application need access to the same database. Therefore, for course registration transactions to work properly, both server applications need to be in the same server group as the Oracle TMS and the University database.

Both of these considerations have implications on the `UBBCONFIG` file for the Wrapper sample application. The following sections discuss these and other additional design considerations in detail.

Sending Requests to the Teller Object

Up until now, all the objects in the University server application have been defined in the same server process. Therefore, for one object to send a request to another object is fairly straightforward, and is summarized in the following steps, using the `Registrar` and `CourseSynopsisEnumerator` objects as an example:

1. The `Registrar` object creates an object reference to the `CourseSynopsisEnumerator` object.
2. Using the newly created object reference, the `Registrar` object sends the request to the `CourseSynopsisEnumerator` object.

3. If the `CourseSynopsisEnumerator` object is not in memory, the TP Framework invokes the `Server::create_servant()` operation on the Server object to instantiate the `CourseSynopsisEnumerator` object.

However, now that there are two server processes running, and an object in one process needs to send a request to an object managed by the second process, the procedure is not quite so straightforward. For example, the notion of getting an object reference to an object in another server process has important implications. For one, the second server process has to be running when the request is made. Also, the factory for the object in the other server process must be available.

The Wrapper sample application addresses this by incorporating the following configuration and design elements:

- The University server application gets the object reference to the `TellerFactory` object in the University Server object's `Server::initialize()` operation. The University server application then caches the `TellerFactory` object reference. This introduces a performance optimization because, otherwise, the `Registrar` object would need to do the following each time it needs a `TellerFactory` object:
  - Invoke the `resolve_initial_references()` operation on the Bootstrap object to get the FactoryFinder object.
  - Invoke the `find_one_factory_by_id()` operation on the FactoryFinder object to obtain a reference to a `TellerFactory` object.
- The Billing server process is started before the University server process is started. When the `Registrar` object subsequently invokes the `TellerFactory` object, the `Registrar` object uses the object reference acquired by the `Server::initialize()` operation (described in the preceding list item). You specify in the `UBBCONFIG` file the order in which server processes are started.
- To handle billing during the course registration process, the `register_for_courses()` and `get_student_details()` operations on the `Registrar` object are modified to include code that invokes operations on the `Teller` object.

Exception Handling

The Wrapper sample application is designed to handle the situation in which the amount owed by the student exceeds the maximum allowed. If the student tries to register for a course when he or she owes more than is permitted by University, the `Registrar` object generates a user-defined `DelinquentAccount` exception. When this exception is returned to the client application, the

client application rolls back the transaction. For information about how to implement user-defined exceptions, see the section “User-defined Exceptions” on page -17.

Setting Transaction Policies on the Interfaces in the Wrapper Sample Application

Another consideration that affects the performance of the Wrapper sample application is setting the appropriate transaction policies for the interfaces of the objects in that application. The `Registrar`, `CourseSynopsisEnumerator`, and `Teller` objects are configured with the `always` transaction policy. The `RegistrarFactory` and `TellerFactory` objects are configured with the `ignore` transaction policy, which prevents the transactional context from being propagated to these objects, which do not need to be included in transactions.

Configuring the University and Billing Server Applications

As mentioned earlier, the Billing server application is configured in a group separate from the group containing the University database and the University application, BEA Tuxedo ATMI Teller application, and Oracle Transaction Manager Server (TMS) application.

However, since the Billing server application participates in the transactions that register students for courses, the Billing server application must include invocations to the `TP::open_xa_rm()` and `TP::close_xa_rm()` operations in the Server object. This is a requirement for any server application that manages an object that is included in any transaction. If that object does not perform any read or write operations on a database, you can specify the NULL resource manager in the following locations:

- In the appropriate group definition in the `UBBCONFIG` file
- In an argument to the `buildobjserver` command when you build the server application

For information about building, configuring, and running the Wrapper sample application, see the *Guide to the CORBA University Sample Applications*.

CHAPTER 8

Scaling a BEA Tuxedo CORBA Server Application

This chapter shows how you can take advantage of several key scalability features of the BEA Tuxedo system to make a CORBA server application highly scalable, using the Production University sample application as an example. The Production sample application uses these scalability features to achieve the following goals:

- To add a parallel processing capability, enabling the BEA Tuxedo domain to process multiple client requests simultaneously
- To spread the processing load on the server applications in the Production sample application across multiple machines

This topic includes the following sections:

- Overview of the Scalability Features Available in the BEA Tuxedo System
- Scaling a BEA Tuxedo Server Application. This section describes:
  - Replicating Server Processes and Server Groups
  - Scaling the Application Via Object State Management
  - Factory-based Routing
- How the Production Server Application Can Be Scaled Further
- Choosing Between Stateless and Stateful Objects

Overview of the Scalability Features Available in the BEA Tuxedo System

Supporting highly scalable applications is one of the strengths of the BEA Tuxedo system. Many applications may perform well in an environment characterized by 1 to 10 server processes, and 10 to 100 client applications. However, in an enterprise environment, applications need to support:

- Hundreds of server processes
- Tens of thousands of client applications
- Millions of objects

Deploying an application with such demands quickly reveals the resource shortcomings and performance bottlenecks in your application. The BEA Tuxedo system supports such large-scale deployments in several ways, three of which are described in this chapter as follows:

- Replicated server processes and server groups
- Object state management
- Factory-based routing

Other features provided in the BEA Tuxedo system to make an application highly scalable include the IIOP Listener/Handler, which is summarized in *Getting Started with BEA Tuxedo CORBA Applications* and fully described in *Setting Up a BEA Tuxedo Application.* See also *Scaling, Distributing, and Tuning CORBA Applications*.

Scaling a BEA Tuxedo Server Application

This section explains how to scale an application to meet a significantly greater processing capability, using the Production sample application as an example. The basic design goal for the Production sample application is to greatly scale up the number of client applications it can accommodate by doing the following:

- Processing in parallel and on one machine client requests on multiple objects that implement the same interface.
- Directing requests on behalf of some students to one machine, and other students to other machines.
- Adding more machines across which to spread the processing load.

To accommodate these design goals, the Production sample application does the following:

- Replicates the University, Billing, and BEA Tuxedo Teller Application server processes within the groups in which they are configured.
- Replicates the groups described above on an additional machine.
- Implements a stateless object model to scale up the number of client requests the server process can manage simultaneously.
- Assigns unique object IDs (OIDs) to the following objects so that they can be instantiated multiple times simultaneously in their respective groups. This makes these objects available on a per-client-application (and not per-process) basis, thereby accommodating a parallel-processing capability:
  - `RegistrarFactory`
  - `Registrar`
  - `TellerFactory`
  - `Teller`
- Implements factory-based routing to direct client requests on behalf of some students to one machine, and other students to another machine.

Note: To make the Production sample application easy for you to use, this application is configured on the BEA Tuxedo software kit to run on one machine, using one database. The examples shown in this chapter, however, show running this application on two machines using two databases.

The design of the Production sample application is set up so that it can be configured to run on several machines and to use multiple databases. Changing the configuration to multiple machines and databases involves modifying the `UBBCONFIG` file and partitioning the databases, and is described in "How the Production Server Application Can Be Scaled Further" on page -20.

The sections that follow describe how the Production sample application uses replicated server processes and server groups, object state management, and factory-based routing to meets its scalability goals. The first section that follows provides a description of the OMG IDL changes implemented in the Production sample application.

OMG IDL Changes for the Production Sample Application

The only OMG IDL changes for the Production sample application are limited to the `find_registrar()` and `find_teller()` operations on, respectively, the `RegistrarFactory`

and `TellerFactory` objects. These two operations are modified to require, respectively, a student ID and account number, which is needed to implement factory-based routing. See the section “Factory-based Routing” on page -11 to read about how the Production sample application implements and uses factory-based routing.

Replicating Server Processes and Server Groups

The BEA Tuxedo system offers a wide variety of choices for how you may configure your server applications, such as:

- One machine with one server process that implements one interface
- One machine with multiple server processes implementing one interface
- One machine with multiple server processes implementing multiple interfaces, with or without factory-based routing
- Multiple machines with multiple server processes and multiple interfaces, with or without factory-based routing

In summary:

- To add more parallel processing capability to your client/server application, replicate your server processes.
- To add more machines to your deployment environment, add more groups and implement factory-based routing.

The following sections describe replicated server processes and groups, and also explain how you can configure them in the BEA Tuxedo system.

Replicated Server Processes

When you replicate the server processes in your application:

- You obtain a means to balance the load of incoming requests on that server application. As requests arrive in the BEA Tuxedo domain for the server group, the BEA Tuxedo system routes the request to the least busy server process within that group.
- You can improve the server application’s performance. Instead of having one server process that can process one client request at one time, you can have multiple server processes available that can process multiple client requests simultaneously. (Note that to make this work, you need to make each object unique, which you can do by having your server application’s factory assign unique OIDs.)

- You obtain a useful failover protection in the event that one of the server images stops.

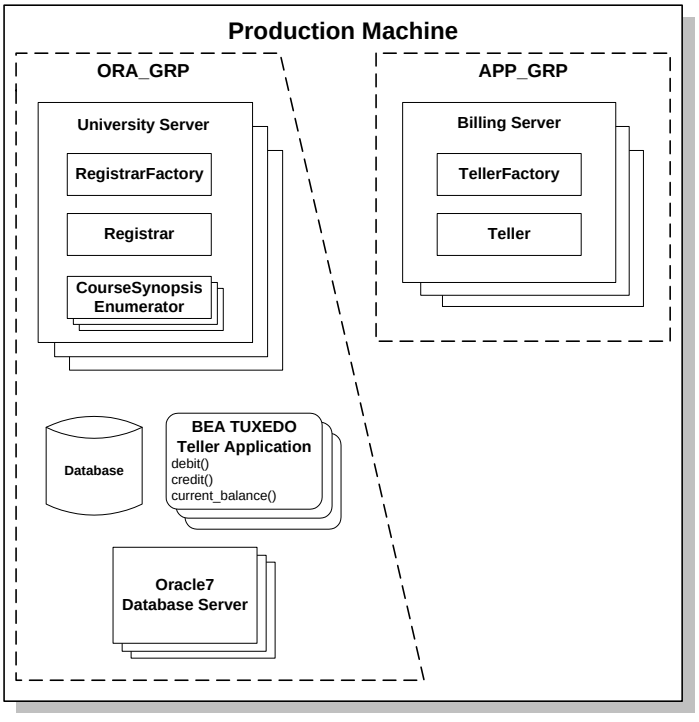
To achieve the full benefit of replicated server processes, make sure that the objects instantiated by your server application generally have unique IDs. This way, a client invocation on an object can cause the object to be instantiated on demand, within the bounds of the number of server processes that are available, and not queued up for an already active object.

Figure 8-1 shows the following:

- The University server application, BEA Tuxedo Teller Application, and Oracle TMS server processes are replicated within the ORA_GRP group.
- The Billing server process is replicated within the APP_GRP group.

Both groups are shown in this figure as running on a single machine.

Figure 8-1 Replicated Server Groups in the Production Sample



When a request arrives for either of these groups, the BEA Tuxedo domain has several server processes available that can process the request, and the BEA Tuxedo domain can choose the server process that is least busy.

In Figure 8-1, note the following:

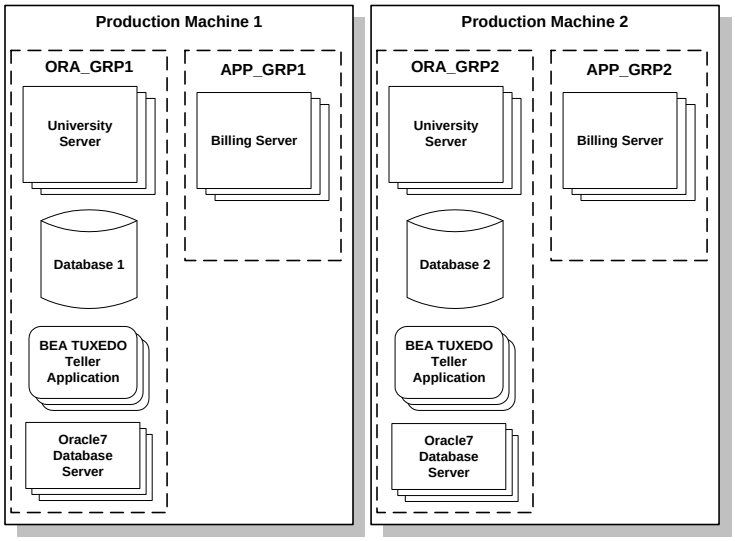
- At any time, there may be no more than one instance of the `RegistrarFactory`, `Registrar`, `TellerFactory`, or `Teller` objects within a given server process.
- There may be any number of `CourseSynopsisEnumerator` objects in any University server process.

Replicated Server Groups

The notion of server groups is specific to the BEA Tuxedo system and adds value to a CORBA implementation; server groups are an important part of the scalability features of the BEA Tuxedo system. Basically, to add more machines to a deployment, you need to add more groups.

Figure 8-2 shows the Production sample application groups replicated on another machine, as specified in the application's `UBBCONFIG` file, as ORA_GRP2 and APP_GRP2.

Figure 8-2 Replicating Server Groups Across Machines



In Figure 8-2, the only difference between the content of the groups on Production Machines 1 and 2 is the database. The database for Production Machine 1 contains student and account information for a subset of students. The database for Production Machine 2 contains student and account information for a different subset of students. (The course information table in both databases is identical.) Note that the student information in a given database may be completely unrelated to the account information in the same database.

The way in which server groups are configured, where they run, and the ways in which they are replicated is specified in the `UBBCONFIG` file. When you replicate a server group, you can do the following:

- Have a means to spread processing load for a given application or set of applications across additional machines.
- Use factory-based routing to send one set of requests on a given interface to one machine, and another set of requests on the same interface to another machine.

The effect of having multiple server groups includes the following:

- When a client request arrives in the BEA Tuxedo domain, the BEA Tuxedo system checks the group ID specified in the object reference.
- The BEA Tuxedo domain sends the request to the least busy server process within the group to which the request is routed that can process the request.

The section “Factory-based Routing” on page -11 shows how the Production sample application uses factory-based routing to spread the application’s processing load across multiple machines.

Configuring Replicated Server Processes and Groups

To configure replicated server processes and groups in your BEA Tuxedo domain:

1. Bring your application’s `UBBCONFIG` file into a text editor, such as WordPad.
2. In the `GROUPS` section, specify the names of the groups you want to configure.
3. In the `SERVERS` section, enter the following information for the server process you want to replicate:
   - A server application name.
   - The `GROUP` parameter, which specifies the name of the group to which the server process belongs. If you are replicating a server process across multiple groups, specify the server process once for each group.

– The SRVID parameter, which specifies a numeric identifier, giving the server process a unique identity.

– The MIN parameter, which specifies the number of instances of the server process to start when the application is booted.

– The MAX parameter, which specifies the maximum number of server processes that can be running at any one time.

Thus the MIN and MAX parameters determine the degree to which a given server application can process requests on a given object in parallel. During run time, the system administrator can examine resource bottlenecks and start additional server processes, if necessary. In this sense, the application is designed so that the system administrator can scale it.

The following example shows lines from the GROUPS and SERVERS sections of the UBBCONFIG file for the Production sample application.

```
*GROUPS
    APP_GRP1
        LMID      = SITE1
        GRPNO     = 2
        TMSNAME   = TMS
    APP_GRP2
        LMID      = SITE1
        GRPNO     = 3
        TMSNAME   = TMS
    ORA_GRP1
        LMID      = SITE1
        GRPNO     = 4
        OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"
    ORA_GRP2
        LMID      = SITE1
        GRPNO     = 5
        OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"
```

```
*SERVERS
    # By default, activate 2 instances of each server
    # and allow the administrator to activate up to 5
    # instances of each server
    DEFAULT:
        MIN     = 2
        MAX     = 5
    tellp_server
        SRVGRP  = ORA_GRP1
        SRVID   = 10
        RESTART = N
    tellp_server
        SRVGRP  = ORA_GRP2
        SRVID   = 10
        RESTART = N
    billp_server
        SRVGRP  = APP_GRP1
        SRVID   = 10
        RESTART = N
    billp_server
        SRVGRP  = APP_GRP2
        SRVID   = 10
        RESTART = N
    univp_server
        SRVGRP  = ORA_GRP1
        SRVID   = 20
        RESTART = N
univp_server
        SRVGRP  = ORA_GRP2
        SRVID   = 20
        RESTART = N
```

Scaling the Application Via Object State Management

As stated in Chapter 1, "CORBA Server Application Concepts," object state management is a fundamentally important concern of large-scale client/server systems because it is critically important that such systems achieve optimized throughput and response time. This section explains how you can use object state management to increase the scalability of the objects

managed by a BEA Tuxedo server application, using the `Registrar` and `Teller` objects in the Production sample applications as an example.

The following table summarizes how you can use the object state management models supported in the BEA Tuxedo system to achieve major gains in scalability in your BEA Tuxedo applications.

State Model	How You Can Use It to Achieve Scalability
Method-bound	Method-bound objects are brought into the machine's memory only for the duration of the client invocation on them. When the invocation is complete, the object is deactivated and any state data for that object is flushed from memory. You can use method-bound objects to create a stateless server model in your application, in which thousands of objects are managed by your application. From the client application view, all the objects are available to service requests. However, because the server application is mapping objects into memory only for the duration of client invocations on them, only comparatively few of the objects managed by the server application are in memory at any given moment. A method-bound object is said in this document to be a stateless object.
Process-bound	Process-bound objects remain in memory from the time they are first invoked until the server process in which they are running is shut down. If appropriate for your application, process-bound objects with a large amount of state data can remain in memory to service multiple client invocations, and the system's resources need not be tied up reading and writing the object's state data on each client invocation. A process-bound object is said in this document to be a stateful object. (Note that transaction-bound objects can also be considered stateful, since they can remain in memory between invocations on them within the scope of a transaction.)

To achieve scalability gains, the `Registrar` and `Teller` objects are configured in the Production server application to have the `method` activation policy. The `method` activation policy assigned to these two objects results in the following behavior changes:

- Whenever these objects are invoked, they are instantiated by the BEA Tuxedo domain in the appropriate server group.
- After the invocation is complete, the BEA Tuxedo domain deactivates these objects.

With the Basic through the Wrapper sample applications, the `Registrar` object was process-bound. All client requests on that object invariably went to the same object instance in the machine's memory. The Basic sample application design may be adequate for a small-scale deployment. However, as client application demands increase, client requests on the `Registrar` object eventually become queued, and response time drops.

However, when the `Registrar` and `Teller` objects are stateless, and the server processes that manage these objects are replicated, these objects can be made available to process client requests on them in parallel. The only constraint on the number of simultaneous client requests that these objects can handle is the number of server processes that are available that can instantiate these objects. These stateless objects, thereby, make for more efficient use of machine resources and reduced client response time.

Most importantly, so that the BEA Tuxedo system can instantiate copies of the `Registrar` and `Teller` objects in each of the replicated server processes, each copy of these objects must be unique. To make each instance of these objects unique, the factories for those objects must assign unique object IDs to them. This, and other design considerations on these two objects, are described in the section "Additional Design Considerations for the Registrar and Teller Objects" on page -17.

Factory-based Routing

Factory-based routing is a powerful feature that provides a means to send a client request to a specific server group. Using factory-based routing, you can spread that processing load for a given application across multiple machines, because you can determine the group, and thus the machine, in which a given object is instantiated.

You can use factory-based routing to expand upon the variety of load-balancing and scalability capabilities in the BEA Tuxedo system. In the case of the Production sample application, you can use factory-based routing to send requests to register one subset of students to one machine, and requests for another subset of students to another machine. As you add machines to ramp up your application's processing capability, the BEA Tuxedo system makes it easy to modify the factory-based routing in your application to add more machines.

The chief benefit of factory-based routing is that it provides a simple means to scale up an application, and invocations on a given interface in particular, across a growing deployment

environment. Spreading the deployment of an application across additional machines is strictly an administrative function that does not require any recoding or rebuilding of the application.

The chief design consideration regarding implementing factory-based routing in your client/server application is in choosing the value on which routing is based. The sections that follow describe how factory-based routing works, using the Production sample application, which uses factory-based routing in the following way:

- Client application requests to the `Registrar` object are routed based on the student ID. That is, requests on behalf of one subset of students go to one group; and requests on behalf of another subset of students go to another group.
- Requests to the `Teller` object are routed based on the account number. That is, billing requests on behalf of one subset of accounts go to one group; and requests on behalf of another subset of accounts go to another group.

How Factory-based Routing Works

Your factories implement factory-based routing by changing the way they create object references. All object references contain a group ID, and by default the group ID is the same as the factory that creates the object reference. However, using factory-based routing, the factory creates an object reference that includes routing criteria that determines the group ID. Then when client applications send an invocation using such an object reference, the BEA Tuxedo system routes the request to the group ID specified in the object reference. This section focuses on how the group ID is generated for an object reference.

To implement factory-based routing, you need to coordinate the following:

- Data in the `INTERFACES` and `ROUTING` sections of the `UBBCONFIG` file.
- Groups, machines, and databases configured in the `UBBCONFIG` file.
- How the factory specifies routing criteria. The interface definition for the factory needs to specify the parameter that represents the routing criteria used to determine the group ID.

To describe the data that needs to be coordinated, the following two sections discuss configuring for factory-based routing in the `UBBCONFIG` file, and implementing factory-based routing in the factory.

Configuring for Factory-based Routing in the UBBCONFIG file

For each interface for which requests are routed, you need to establish the following information in the `UBBCONFIG` file:

- Details about the data in the routing criteria
- For each kind of criteria, the values that route to specific server groups

To configure for factory-based routing, the UBBCONFIG file needs to specify the following data in the INTERFACES and ROUTING sections, and also in how groups and machines are identified:

1. The INTERFACES section lists the names of the interfaces for which you want to enable factory-based routing. For each interface, this section specifies what kinds of criteria the interface routes on. This section specifies the routing criteria via an identifier, FACTORYROUTING, as in the following example:

   ```
   INTERFACES
       "IDL:beasys.com/UniversityP/Registrar:1.0"
           FACTORYROUTING = STU_ID
       "IDL:beasys.com/BillingP/Teller:1.0"
           FACTORYROUTING = ACT_NUM
   ```

   The preceding example shows the fully qualified interface names for the two interfaces in the Production sample in which factory-based routing is used. The FACTORYROUTING identifier specifies the names of the routing values, which are STU_ID and ACT_NUM, respectively.

2. The ROUTING section specifies the following data for each routing value:
   - The TYPE parameter, which specifies the type of routing. In the Production sample, the type of routing is factory-based routing. Therefore, this parameter is defined to FACTORY.
   - The FIELD parameter, which specifies the name that the factory inserts in the routing value. In the Production sample, the field parameters are student_id and account_number, respectively.
   - The FIELDTYPE parameter, which specifies the data type of the routing value. In the Production sample, the field types for student_id and account_number are long.
   - The RANGES parameter, which specifies the values that are routed to each group.

   The following example shows the ROUTING section of the UBBCONFIG file used in the Production sample application:

```
ROUTING
    STU_ID
        FIELD     = "student_id"
        TYPE      = FACTORY
        FIELDTYPE = LONG
        RANGES    = "100001-100005:ORA_GRP1,100006-100010:ORA_GRP2"
    ACT_NUM
```

```
FIELD     = "account_number"
TYPE      = FACTORY
FIELDTYPE = LONG
RANGES    = "200010-200014:APP_GRP1,200015-200019:APP_GRP2"
```

The preceding example shows that `Registrar` object references for students with IDs in one range are routed to one server group, and `Registrar` object references for students with IDs in another range are routed to another group. Likewise, `Teller` object references for accounts in one range are routed to one server group, and `Teller` object references for accounts in another range are routed to another group.

3. The groups specified by the `RANGES` identifier in the `ROUTING` section of the `UBBCONFIG` file need to be identified and configured. For example, the Production sample specifies four groups: APP_GRP1, APP_GRP2, ORA_GRP1, and ORA_GRP2. These groups need to be configured, and the machines on which they run need to be identified.

   The following example shows the `GROUPS` section of the Production sample `UBBCONFIG` file, in which the ORA_GRP1 and ORA_GRP2 groups are configured. Notice how the names in the `GROUPS` section match the group names specified in the `ROUTING` section; this is critical for factory-based routing to work correctly. Furthermore, any change in the way groups are configured in an application must be reflected in the `ROUTING` section. (Note that the Production sample packaged with the BEA Tuxedo software is configured to run entirely on one machine. However, you can easily configure this application to run on multiple machines.)

```
*GROUPS
    APP_GRP1
        LMID      = SITE1
        GRPNO     = 2
        TMSNAME   = TMS
    APP_GRP2
        LMID      = SITE1
        GRPNO     = 3
        TMSNAME   = TMS
    ORA_GRP1
        LMID      = SITE1
        GRPNO     = 4
        OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
        CLOSEINFO = ""
        TMSNAME   = "TMS_ORA"
    ORA_GRP2
        LMID      = SITE1
```

```
GRPNO     = 5
OPENINFO  = "ORACLE_XA:Oracle_XA+Acc=P/scott/..."
CLOSEINFO = ""
TMSNAME   = "TMS_ORA"
```

Implementing Factory-based Routing in a Factory

Factories implement factory-based routing by the way the invocation to the `TP::create_object_reference()` operation is implemented. This operation has the following C++ binding:

```
CORBA::Object_ptr  TP::create_object_reference (
                       const char* interfaceName,
                       const PortableServer::oid &stroid,
                       CORBA::NVlist_ptr criteria);
```

The third parameter to this operation, `criteria`, specifies a list of named values to be used for factory-based routing. Therefore, the work of implementing factory-based routing in a factory is in building the `NVlist`.

As stated previously, the `RegistrarFactory` object in the Production sample application specifies the value `STU_ID`. This value must match exactly the following in the `UBBCONFIG` file:

- The routing name, type, and allowable values specified by the `FACTORYROUTING` identifier in the `INTERFACES` section.
- The routing criteria name, field, and field type specified in the `ROUTING` section.

The `RegistrarFactory` object inserts the student ID into the `NVlist` using the following code:

```
// put the student id (which is the routing criteria)
// into a CORBA NVList:
CORBA::NVList_var v_criteria;
TP::orb()->create_list(1, v_criteria.out());
CORBA::Any any;
any <<= (CORBA::Long)student;
v_criteria->add_value("student_id", any, 0);
```

The `RegistrarFactory` object has the following invocation to the `TP::create_object_reference()` operation, passing the `NVlist` created in the preceding code example:

```
// create the registrar object reference using
// the routing criteria :
```

```
CORBA::Object_var v_reg_oref =
    TP::create_object_reference(
        UniversityP::_tc_Registrar->id(),
        object_id,
        v_criteria.in()
    );
```

The Production sample application also uses factory-based routing in the `TellerFactory` object to determine the group in which `Teller` objects should be instantiated based on an account number.

Note: It is possible for an object with a given interface and OID to be simultaneously active in two different groups, if those two groups both contain the same object implementation. (However, if your factories generate unique OIDs, this situation is very unlikely.) If you need to guarantee that only one object instance of a given interface name and OID is available at any one time in your domain, either: use factory-based routing to ensure that objects with a particular OID are always routed to the same group, or configure your domain so that a given object implementation is in only one group. This assures that if multiple clients have an object reference containing a given interface name and OID, the reference is always routed to the same object instance.

To enable routing on an object's OID, specify the OID as the routing criterion in the `TP::create_object_reference()` operation, and set up the `UBBCONFIG` file appropriately.

What Happens at Run Time

When you implement factory-based routing in a factory, the BEA Tuxedo system generates an object reference. The following example shows how the client application gets an object reference to a `Registrar` object when factory-based routing is implemented:

1. The client application invokes the `RegistrarFactory` object, requesting a reference to a `Registrar` object. Included in the request is a student ID.
2. The `RegistrarFactory` inserts the student ID into an `NVlist`, which is used as the routing criteria.
3. The `RegistrarFactory` invokes the `TP::create_object_reference()` operation, passing the `Registrar` interface name, a unique OID, and the `NVlist`.
4. The BEA Tuxedo system compares the contents of the routing tables with the value in the `NVlist` to determine a group ID.

5. The BEA Tuxedo system inserts the group ID into the object reference.

When the client application subsequently does an invocation on an object using the object reference, the BEA Tuxedo system routes the request to the group specified in the object reference.

Note: Be careful how you implement factory-based routing if you use the Process-Entity design pattern. The object can service only those entities that are contained in the group's database.

Additional Design Considerations for the Registrar and Teller Objects

The principal considerations that influence the design of the `Registrar` and `Teller` objects include:

- How to ensure that the `Registrar` and `Teller` objects work properly for the Production deployment environment; namely, across multiple replicated server processes and multiple groups. Given that the University and Billing server processes are replicated, the design must consider how these two objects should be instantiated.
- How to ensure that client requests for registration and billing operations for a given student go to the correct server group, given that the two server groups in the Production BEA Tuxedo domain each deal with different databases.

The primary implications of these considerations are that these objects must:

- Have unique object IDs (OIDs)
- Be method-bound; that is, have the `method` activation policy assigned to them

The remainder of this section discusses these considerations and implications in detail.

Instantiating the Registrar and Teller Objects

In University server applications prior to the Production sample application, the run-time behavior of the `Registrar` and `Teller` objects was fairly simple:

- Each object was process-bound, meaning that each was activated the first time it was invoked, and it stayed in memory until the server process in which it ran was shut down.
- Since there was only one server group running in the BEA Tuxedo domain, and only one University and Billing server process in the group, all client requests were directed to the

same objects. As multiple client requests arrived in the BEA Tuxedo domain, these objects each processed one client request at one time.

- Because there was only one instance of each object in the server processes in which they ran, neither object needed a unique OID. The OID for each object specified only the Interface Repository ID.

However, since the University and Billing server processes are now replicated, the BEA Tuxedo domain must have a means to differentiate between multiple instances of the `Registrar` and `Teller` objects. That is, if there are two University server processes running in a group, the BEA Tuxedo domain must have a means to distinguish between, say, the `Registrar` object running in the first University server process and the `Registrar` object running in the second University server process.

The way to provide the BEA Tuxedo domain with the ability to distinguish among multiple instances of these objects is to make each object instance unique.

To make each `Registrar` and `Teller` object unique, the factories for those objects must change the way in which they make object references to them. For example, when the `RegistrarFactory` object in the Basic sample application created an object reference to the `Registrar` object, the `TP::create_object_reference()` operation specified an OID that consisted only of the string `registrar`. However, in the Production sample application, the same `TP::create_object_reference()` operation uses a generated unique OID instead.

A consequence of giving each `Registrar` and `Teller` object a unique OID is that there may be multiple instances of these objects running simultaneously in the BEA Tuxedo domain. This characteristic is typical of the stateless object model, and is an example of how the BEA Tuxedo domain can be highly scalable and at the same time offer high performance.

And last, since unique `Registrar` and `Teller` objects need to be brought into memory for each client request on them, it is critical that these objects be deactivated when the invocations on them are completed so that any object state associated with them does not remain idle in memory. The Production server application addresses this issue by assigning the `method` activation policy to these two objects in the ICF file.

Ensuring That Student Registration Occurs in the Correct Server Group

The chief scalability advantage of having replicated server groups is to be able to distribute processing across multiple machines. However, if your application interacts with a database, which is the case with the University sample applications, it is critical that you consider the impact of these multiple server groups on the database interactions.

In many cases, you may have one database associated with each machine in your deployment. If your server application is distributed across multiple machines, you must consider how you set up your databases.

The Production sample application, as described in this chapter, uses two databases. However, this application can easily be configured to accommodate more. The system administrator can decide how many.

In the Production sample application, the student and account information is partitioned across the two databases, but course information is identical. Having identical course information in both databases is not a problem because the course information is read-only for the purposes of course registration. However, the student and account information is read-write. If multiple databases were also to contain identical data for students and accounts (that is, the database is not partitioned), the application would need to deal with the overhead of synchronizing the updates to student and account information across all the databases each time any student or account information were to change.

The Production sample application uses factory-based routing to send one set of requests to one machine, and another set to the other machine. As mentioned earlier, factory-based routing is implemented in the `RegistrarFactory` object by the way in which references to `Registrar` objects are created.

For example, when the client application sends a request to the `RegistrarFactory` object to get an object reference to a `Registrar` object, the client application includes a student ID in that request. The client application must use the object reference that the `RegistrarFactory` object returns to make all subsequent invocations on a `Registrar` object on a particular student's behalf, because the object reference returned by the factory is group-specific. Therefore, for example, when the client application subsequently invokes the `get_student_details()` operation on the `Registrar` object, the client application can be assured that the `Registrar` object is active in the server group associated with the database containing data for that student. To show how this works, consider the following execution scenario, which is implemented in the Production sample application:

1. The client application invokes the `find_registrar()` operation on the `RegistrarFactory` object. Included in this invocation is the student ID 1000003.
2. The BEA Tuxedo domain routes the client request to any `RegistrarFactory` object.
3. The `RegistrarFactory` object uses the student ID to create an object reference to a `Registrar` object in ORA_GRP1, based on the routing information in the `UBBCONFIG` file, and returns that object reference to the client application.

4. The client application invokes the `register_for_courses()` operation on the Registrar object.

5. The BEA Tuxedo domain receives the client request and routes it to the server group specified in the object reference. In this case, the client request goes to the University server process in ORA_GRP1, which is on Production Machine 1.

6. The University server process instantiates a `Registrar` object and sends the client invocation to it.

The `RegistrarFactory` object from the preceding scenario returns to the client application a unique reference to a `Registrar` object that can be instantiated only in ORA_GRP1, which runs on Production Machine 1 and has a database containing student data for students with IDs in the range 100001 to 100005. Therefore, when the client application sends subsequent requests to this `Registrar` object on behalf of a given student, the `Registrar` object interacts with the correct database.

Ensuring That the Teller Object Is Instantiated in the Correct Server Group

When the `Registrar` object needs a `Teller` object, the `Registrar` object invokes the `TellerFactory` object, using the `TellerFactory` object reference cached in the University Server object, as described in “Sending Requests to the Teller Object” on page -10.

However, because factory-based routing is used in the `TellerFactory` object, the `Registrar` object passes the student’s account number when the `Registrar` object requests a reference to a `Teller` object. This way, the `TellerFactory` object creates a reference to a `Teller` object in the group that has the correct database.

Note: For the Production sample application to work properly, it is essential that the system administrator configures the server groups and the databases properly. In particular, the system administrator must make sure that a match exists between the routing criteria specified in the routing tables and the databases to which requests using those criteria are routed. Using the Production sample as an example, the database in a given group must contain the correct student and account information for the requests that are routed to that group.

How the Production Server Application Can Be Scaled Further

In the future, the system administrator of the Production sample application may want to add capacity to the BEA Tuxedo domain. For example, the University may eventually have a large

increase in the student population, or the Production application may be scaled up to accommodate the course registration process for an entire state university system encompassing several campuses. This can be done without modifying or rebuilding the application.

The system administrator has the following tools available to continually add capacity:

- Replicating the Production sample application server groups across additional machines.

  Doing this requires modifying the `UBBCONFIG` file to specify the additional groups, what server processes run in those groups, and what machines they run on.

- Changing the factory-based routing tables

  For example, instead of routing to the two groups shown earlier in this chapter, the system administrator can modify the routing rules in the `UBBCONFIG` file to partition the application further among the new groups added to the BEA Tuxedo domain. Any modification to the routing tables must be consistent with any changes or additions made to the server groups and machines configured in the `UBBCONFIG` file.

Note: If you add capacity to an application that uses a database, you must also consider the impact on how the database is set up, particularly when you are using factory-based routing. For example, if the Production sample application is spread across six machines, the database on each machine must be set up appropriately and in accordance with the routing tables in the `UBBCONFIG` file.

Choosing Between Stateless and Stateful Objects

In general, you need to balance the costs of implementing stateless objects against the costs of implementing stateful objects.

In the case where the cost to initialize an object with its durable state is expensive— because, for example, the object's data takes up a great deal of space, or the durable state is located on a disk very remote to the servant that activates it—it may make sense to keep the object stateful, even if the object is idle during a conversation. In the case where the cost to keep an object active is expensive in terms of machine resource usage, it may make sense to make such an object stateless.

By managing object state in a way that's efficient and appropriate for your application, you can maximize your application's ability to support large numbers of simultaneous client applications that use large numbers of objects. You generally do this by assigning the `method` activation policy to these objects, which has the effect of deactivating idle object instances so that machine resources can be allocated to other object instances. However, your specific application characteristics and needs may vary.

Note: BEA Tuxedo Release 8.0 or later provides support for parallel objects, as a performance enhancement. This feature allows you to designate all business objects in a particular application as stateless objects. For complete information, see Chapter 3, "TP Framework," in the *CORBA Programming Reference*.

When You Want Stateless Objects

Stateless objects generally provide good performance and optimal usage of server resources, because server resources are never used when objects are idle. Stateless objects are generally a good approach to implementing server applications. Stateless objects are particularly appropriate in the following situations:

- The client application typically waits for user input between invocations on the object.
- The client request typically contains all the data needed by the server application, and the server can process the client request using only that data.
- The object has very high access rates, but low access rates from any one particular client application.

By making an object stateless, you can generally assure that server application resources are not being tied up for an arbitrarily long time waiting for input from the client application.

Note the following characteristics about an application that employs a stateless object model:

- Information about and associated with an invocation is not maintained after the server application has finished executing a client request.
- An incoming client request is sent to the first available server process: after the request has been satisfied, the application state vanishes and the server application is available for another client application request.
- Durable state information for the object exists outside the server process. With each invocation on this object, the durable state is read into memory.
- The BEA Tuxedo domain may direct successive requests on an object from a given client application to a different server process.
- The overall system performance of a machine that is running stateless objects is usually enhanced.

When You Want Stateful Objects

A stateful object, once activated, remains in memory until a specific event occurs, such as the process in which the object exists is shut down, or the transaction in which the object is activated is completed.

Stateful objects are typically appropriate in the following situations:

- When an object is used very frequently by a large number of client applications. This is the case for long-lived, well-known objects like factories. When the server application keeps these objects active, the client application typically experiences minimal response time in accessing them. Since these active objects are shared by many client applications, there are relatively few objects of this type in memory.

Note: Plan carefully how process objects are potentially involved in a transaction. Any object that is involved in a transaction cannot be invoked by another client application or object. Process objects meant to be used by a large number of client applications can create problems if they are involved in transactions frequently or for long durations.

- When a client application must invoke successive operations on an object to complete a transaction, and the client application is not idle while waiting for user input between those invocations. In this case, if the object were deactivated between invocations, there would be a degradation of response time because state would be written and read between each invocation; such behavior may not be appropriate for transactions. You can trade holding server resources for better response time.

Note the following behavior with stateful objects:

- State information is maintained between server invocations, and the servant typically remains dedicated to a given client application for a specified duration.
- Even though data is sent and received between the client and server applications, the server process maintains additional context or application state information in memory.
- In cases where one or more stateful objects are using a lot of machine resources, server performance for tasks and processes not associated with the stateful object may be worse than with a stateless server model.

  For example, if an object has a lock on a database and is caching a lot of data in memory, that database and the memory used by that stateful object are unavailable to other objects, potentially for the entire duration of a transaction.