BEA Tuxedo®

Getting Started with BEA Tuxedo CORBA Applications

Version 10.0
Document Released: September 28, 2007

Contents

1. Overview of the BEA Tuxedo CORBA Environment

Introduction to the BEA Tuxedo CORBA Environment 1-1

Features of the BEA Tuxedo CORBA Environment 1-3

2. The BEA Tuxedo CORBA Programming Environment

Overview of the BEA Tuxedo CORBA Programming Features 2-1

IDL Compilers 2-2

Development Commands. 2-2

Administration Tools 2-3

BEA Tuxedo CORBA Object Services. 2-5

BEA Tuxedo CORBA Architectural Components 2-6

Bootstrapping the BEA Tuxedo Domain 2-7

IIOP Listener/Handler 2-9

ORB 2-10

TP Framework 2-11

How BEA Tuxedo CORBA Client and Server Applications Interact 2-13

Step 1: The CORBA Server Application Is Initialized 2-13

Step 2: The CORBA Client Application Is Initialized. 2-14

Step 3: The CORBA Client Application Authenticates Itself to the BEA Tuxedo Domain 2-15

Step 4: The CORBA Client Application Obtains a Reference to the CORBA Object Needed to Execute Its Business Logic 2-15

Step 5: The CORBA Client Application Invokes an Operation on the CORBA Object 2-17

3. Developing BEA Tuxedo CORBA Applications

Overview of the Development Process for BEA Tuxedo CORBA Applications 3-2
The Simpapp Sample Application 3-3
Step 1: Write the OMG IDL Code 3-4
Step 2: Generate CORBA client Stubs and Skeletons 3-6
Step 3: Write the CORBA Server Application. 3-7
Writing the Methods That Implement the Operations for Each Interface. 3-7
Creating the CORBA server Object 3-9
Defining an Object’s Activation Policies 3-11
Creating and Registering a Factory. 3-12
Releasing the CORBA Server Application. 3-13
Step 4: Write the CORBA Client Application 3-14
Step 5: Create an XA Resource Manager. 3-16
Step 6: Create a Configuration File 3-17
Step 7: Create the TUXCONFIG File 3-20
Step 8: Compile the CORBA Server Application 3-20
Step 9: Compile the CORBA Client Application. 3-20
Step 10: Start the BEA Tuxedo CORBA Application 3-21
Additional BEA Tuxedo CORBA Sample Applications 3-21

4. Using Security

Overview of the Security Service. 4-2
How Security Works 4-2
The Security Sample Application. 4-4
Development Steps. 4-5
Step 1: Define the Security Level in the Configuration File. 4-6

Step 2: Write the CORBA Client Application 4-6

5. Using Transactions

Overview of the Transaction Service 5-2

What Happens During a Transaction 5-2

Transactions Sample Application 5-4

Development Steps 5-6

Step 1: Write the OMG IDL Code 5-6

Step 2: Define Transaction Policies for the Interfaces 5-9

Step 3: Write the CORBA Client Application 5-10

Step 4: Write the CORBA Server Application 5-11

Step 5: Create a Configuration File 5-14

Index

CHAPTER 1

Overview of the BEA Tuxedo CORBA Environment

This topic includes the following sections:

- Introduction to the BEA Tuxedo CORBA environment
- Features of the BEA Tuxedo CORBA Environment

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

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

Introduction to the BEA Tuxedo CORBA Environment

The CORBA environment in the BEA Tuxedo product is based on the CORBA standard as a programming model for developing enterprise applications with high performance, scalability, and reliability. BEA Tuxedo CORBA extends the Object Request Broker (ORB) model with online transaction processing (OLTP) functions. The BEA Tuxedo CORBA deployment infrastructure delivers secure, transactional, distributed applications in a managed environment.

CORBA objects built with the BEA Tuxedo product are accessible from Web-based applications that communicate using the CORBA Object Management Group (OMG) Internet Inter-ORB

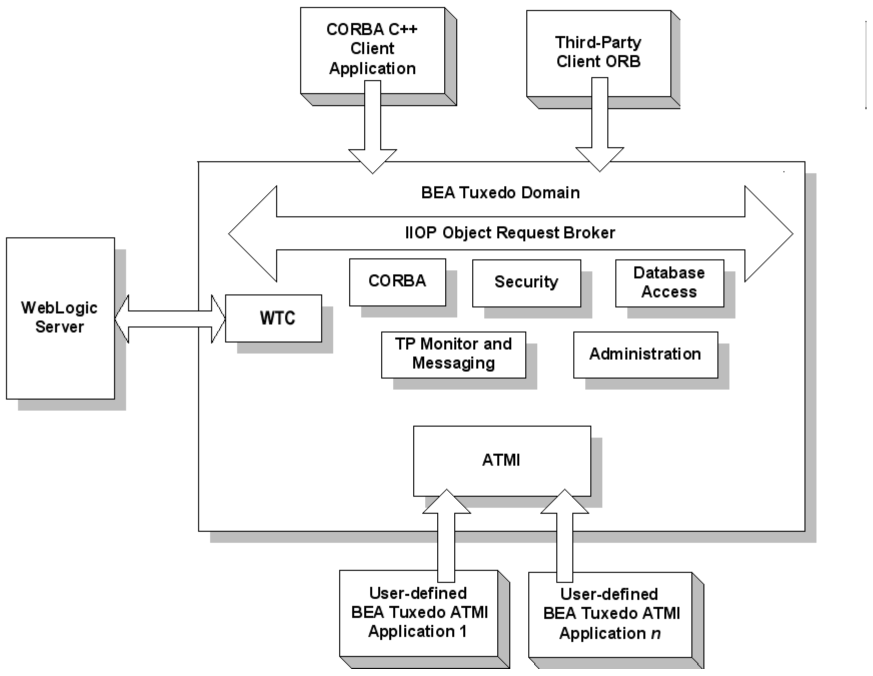
Protocol (IIOP). IIOP is the standard protocol for communications running on the Internet or on an intranet within an enterprise.

BEA Tuxedo CORBA has a native implementation of IIOP, ensuring high-performance, interoperable, distributed-object applications for the Internet, intranets, and enterprise computing environments. You can build integrated enterprise applications using multiple programming models. CORBA and Application-to- Transaction-Monitor-Interface (ATMI) applications can be developed with fully integrated transaction management, security, administration, and reliability capabilities.

The interoperability technology incorporated into BEA Tuxedo CORBA provides for scalable connectivity between the CORBA and WebLogic Server environments. For information on interoperability see *BEA Tuxedo Interoperability* in the BEA Tuxedo online documentation.

Figure 1-1 illustrates the BEA Tuxedo CORBA environment.

Figure 1-1 BEA Tuxedo CORBA


CORBA C++ Client Application
Third-Party Client ORB
BEA Tuxedo Domain
IIOP Object Request Broker
CORBA
Security
Database Access
WebLogic Server
WTC
TP Monitor and Messaging
Administration
ATMI
User-defined BEA Tuxedo ATMI Application 1
User-defined BEA Tuxedo ATMI Application *n*

The following sections outline the features of the CORBA environment.

Features of the BEA Tuxedo CORBA Environment

The CORBA environment in the BEA Tuxedo product provides the following set of features:

- A C++ server-side ORB
- Client application options including:
  - CORBA C++ client
  - Third-party client ORBs
- A proven run-time infrastructure for hosting e-commerce transaction applications, including client connection concentrators, high-performance message routing and load balancing, and high-availability features.
- A Transaction Processing (TP) Framework for object state and transaction management in CORBA applications.
- A Management Information Base (MIB) that defines the key management attributes of CORBA applications. In addition, programming interfaces and scripting capabilities are available to access the MIBs.
- An Administration Console graphical user interface (GUI) for the management of CORBA applications.
- The CORBA Transaction Service (OTS) to ensure the integrity of your data even when transactions span multiple programming models, databases, and applications.
- A security service that handles authentication for principals that need to access resources in a CORBA object in the CORBA environment.
- The Secure Sockets Layer (SSL) protocol to encrypt client to server communication on the wire. SSL support includes IIOP connection pools.
- A Security Service Plug-In Interface (SPI) for CORBA that allows integration of third-party security plug-ins.
- A Notification Service that receives event posting messages, filters them, and distributes the messages to subscribers. The Notification Service provides two sets of interfaces: a CORBA-based interface and a simplified BEA-proprietary interface.

- An implementation of the CosLifeCycle service.
- An implementation of CosNaming that allows BEA Tuxedo CORBA server applications to advertise object references using logical names.
- An interface repository that stores meta information about BEA Tuxedo CORBA objects. Meta information includes information about modules, interfaces, operations, attributes, and exceptions.
- Dynamic Invocation Interface (DII) support. DII allows BEA Tuxedo CORBA client applications to create requests dynamically for objects that were not defined at compile time.

The remainder of this manual describes the programming environment for BEA Tuxedo CORBA and the development process for CORBA applications.

CHAPTER 2

The BEA Tuxedo CORBA Programming Environment

This topic includes the following sections:

- Overview of the BEA Tuxedo CORBA Programming Features
- BEA Tuxedo CORBA Object Services
- BEA Tuxedo CORBA Architectural Components
- How BEA Tuxedo CORBA Client and Server Applications Interact

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

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

Overview of the BEA Tuxedo CORBA Programming Features

BEA Tuxedo offers a robust CORBA programming environment that simplifies the development and management of distributed objects. The following topics describe the features of the programming environment:

- IDL Compilers
- Development Commands
- Administration Tools

IDL Compilers

The BEA Tuxedo CORBA programming environment supplies Interface Definition Language (IDL) compilers to facilitate the development of CORBA objects:

- `idl`—compiles the OMG IDL file and generates client stub and server skeleton files required for interface definitions being implemented in C++.

For a description of how to use the IDL compiler, see Chapter 3, "Developing BEA Tuxedo CORBA Applications." For a description of the `idl` command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Development Commands

Table 2-1 lists the commands that the BEA Tuxedo CORBA programming environment provides for developing CORBA applications and managing the Interface Repository.

Table 2-1 BEA Tuxedo CORBA Development Commands

Development Command	Description
`buildobjclient`	Constructs a C++ client application.
`buildobjserver`	Constructs a C++ server application.
`genicf`	Generates an Implementation Configuration File (ICF). The ICF file defines activation and transaction policies for C++ server applications.
`idl2ir`	Creates the Interface Repository and loads interface definitions into it.
`ir2idl`	Shows the content of the Interface Repository.
`irdel`	Deletes the specified object from the Interface Repository.

For a description of how to use the development commands to develop client and server applications, see Chapter 3, "Developing BEA Tuxedo CORBA Applications."

For a description of the development commands, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Administration Tools

The BEA Tuxedo CORBA programming environment provides a complete set of tools for administering your CORBA applications. You can manage BEA Tuxedo CORBA applications through commands, through a graphical user interface, or by including administration utilities in a script.

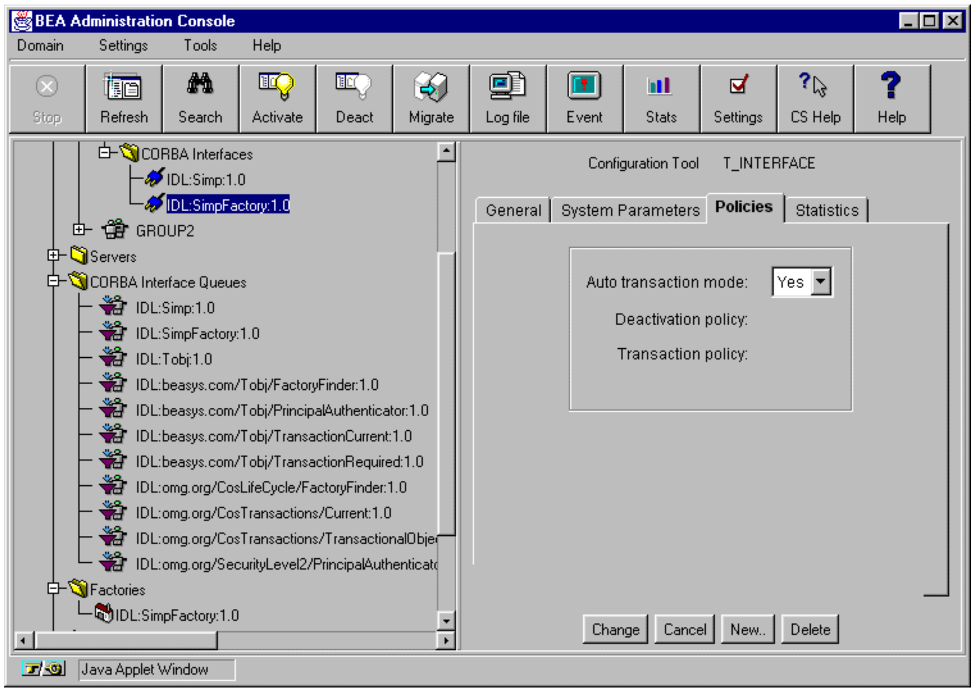
You can use the commands listed in Table 2-2 to perform administration tasks for your CORBA application.

Table 2-2 Administration Commands

Administration Command	Description
`tmadmin`	Displays information about current configuration parameters.
`tmboot`	Activates the BEA Tuxedo CORBA application referenced in the specified configuration file. Depending on the options used, the entire application or parts of the application are started.
`tmconfig`	Dynamically updates and retrieves information about the configuration of a BEA Tuxedo CORBA application.
`tmloadcf`	Parses the configuration file and loads the binary version of the configuration file.
`tmshutdown`	Shuts down a set of specified server applications, or removes interfaces from a configuration file.
`tmunloadcf`	Unloads the configuration file.

The Administration Console is a Java-based applet that you can download into your Internet browser and use to manage your BEA Tuxedo CORBA applications remotely. The Administration Console allows you to perform administration tasks, such as monitoring system events, managing system resources, creating and configuring administration objects, and viewing system statistics. Figure 2-1 shows the main window of the Administration Console.

Figure 2-1 Administration Console Main Window


BEA Administration Console
Domain
Settings
Tools
Help
Stop
Refresh
Search
Activate
Deact
Migrate
Log file
Event
Stats
Settings
CS Help
Help
CORBA Interfaces
IDL:Simp:1.0
IDL:SimpFactory:1.0
GROUP2
Servers
CORBA Interface Queues
IDL:Simp:1.0
IDL:SimpFactory:1.0
IDL:Tobj:1.0
IDL:beasys.com/Tobj/FactoryFinder:1.0
IDL:beasys.com/Tobj/PrincipalAuthenticator:1.0
IDL:beasys.com/Tobj/TransactionCurrent:1.0
IDL:beasys.com/Tobj/TransactionRequired:1.0
IDL:omg.org/CosLifeCycle/FactoryFinder:1.0
IDL:omg.org/CosTransactions/Current:1.0
IDL:omg.org/CosTransactions/TransactionalObje
IDL:omg.org/SecurityLevel2/PrincipalAuthenticat
Factories
IDL:SimpFactory:1.0
Configuration Tool T_INTERFACE
General
System Parameters
Policies
Statistics
Auto transaction mode: Yes
Deactivation policy:
Transaction policy:
Change
Cancel
New..
Delete
Java Applet Window

In addition, a set of utilities called the AdminAPI is provided for directly accessing and manipulating system settings in the Management Information Bases (MIBs) for the BEA Tuxedo product. The advantage of the AdminAPI is that it can be used to automate administrative tasks, such as monitoring log files and dynamically reconfiguring an application, thus eliminating the need for manual intervention.

For information about the Administration commands, see *File Formats, Data Descriptions, MIBs, and System Processes Reference* in the BEA Tuxedo online documentation.

For a description of the Administration Console and how it works, see the online help that is integrated into the Administration Console graphical user interface (GUI).

For information about the AdminAPI, see *Setting Up a BEA Tuxedo Application* in the BEA Tuxedo online documentation.

BEA Tuxedo CORBA Object Services

The BEA Tuxedo product includes a set of environmental objects that provide object services to CORBA client applications in a BEA Tuxedo domain. You access the environmental objects through a bootstrapping process that accesses the services in a particular BEA Tuxedo domain.

BEA Tuxedo CORBA provides the following services:

- Object Life Cycle service

  The Object Life Cycle service is provided through the FactoryFinder environmental object. The FactoryFinder object is a CORBA object that can be used to locate a factory, which in turn can create object references for CORBA objects. Factories and FactoryFinder objects are implementations of the CORBA Services Life Cycle Service. BEA Tuxedo CORBA applications use the Object Life Cycle service to find object references.

  For information about using the Object Life Cycle Service, see "How BEA Tuxedo CORBA Client and Server Applications Interact" on page 2-13.

- Security service

  The Security service is accessed through either the SecurityCurrent environmental object or the PrincipalAuthenticator object. The SecurityCurrent and PrincipalAuthenticator objects are used to authenticate a client application into a BEA Tuxedo domain with the proper security. The BEA Tuxedo software provides an implementation of the CORBA Services Security Service.

  For information about using security, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

- Transaction service

  The Transaction service is accessed through either the TransactionCurrent environmental object or the TransactionFactory object. The TransactionCurrent and TransactionFactory objects allow a client application to participate in a transaction. The BEA Tuxedo software provides an implementation of the CORBA Services Object Transaction Service (OTS).

  For information about using transactions, see *Using CORBA Transactions* in the BEA Tuxedo online documentation.

- Interface Repository service

  The Interface Repository service is accessed through the InterfaceRepository object. The InterfaceRepository object is a CORBA object that contains interface definitions for all the available CORBA interfaces and the factories used to create object references to the

CORBA interfaces. The InterfaceRepository object is used with client applications that use DII.

For information about using DII, see *Creating CORBA Client Applications*.

BEA Tuxedo CORBA provides environmental objects for the following programming environments:

- C++

BEA Tuxedo CORBA also supports the use of the OMG CORBA Interoperable Naming Service (INS) by third-party clients, to obtain initial object references.

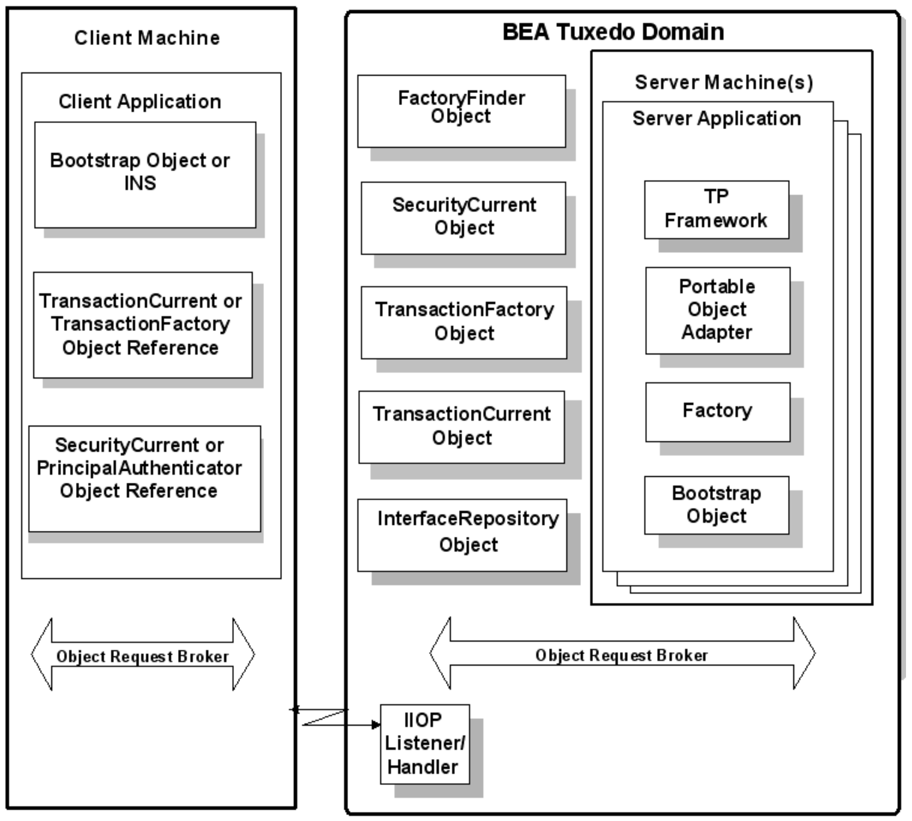
BEA Tuxedo CORBA Architectural Components

This section provides an introduction to the following architectural components of the BEA Tuxedo CORBA programming environment:

- Bootstrapping the BEA Tuxedo Domain
- IIOP Listener/Handler
- ORB
- TP Framework

Figure 2-2 illustrates the components in a BEA Tuxedo CORBA application.

Figure 2-2 Components in a BEA Tuxedo CORBA Application



Bootstrapping the BEA Tuxedo Domain

A domain is a way of grouping objects and services together as a management entity. A BEA Tuxedo domain has at least one IIOP Listener/Handler and is identified by a name. One client application can connect to multiple BEA Tuxedo domains using different Bootstrap objects.

Bootstrapping the BEA Tuxedo domain establishes communication between a client application and the domain. There are two mechanisms available for bootstrapping, the BEA mechanism and the CORBA Interoperable Naming Service (INS) bootstrapping mechanism specified by the OMG. Use the BEA mechanism if you are using BEA CORBA client software. Use the CORBA INS mechanism if you are using a client ORB from another vendor. For more information about bootstrapping the BEA Tuxedo domain, see the *CORBA Programming Reference* in the BEA Tuxedo online documentation.

One of the first things that client applications do after startup is create a Bootstrap object by supplying the host and port of the IIOP Listener/Handler using one of the following URL address formats:

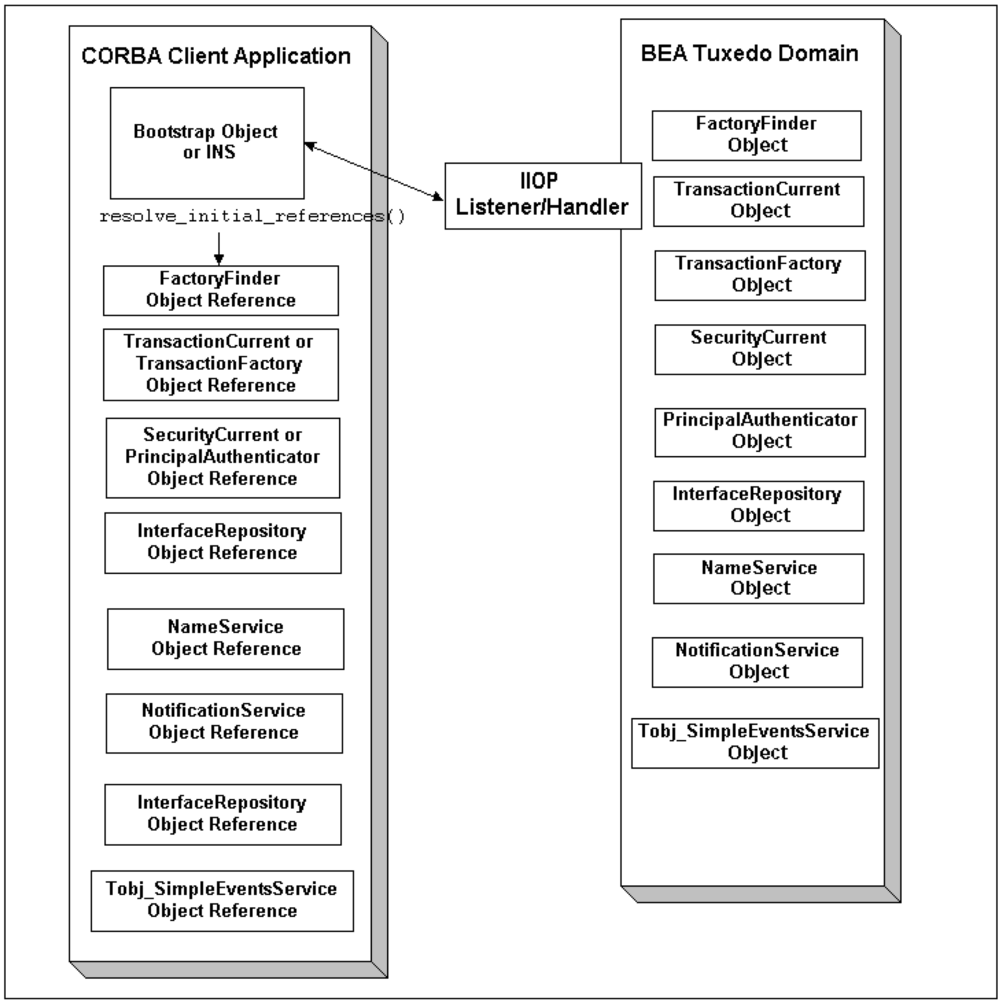
- `//host:port`
- `corbaloc://host:port`
- `corbalocs://host:port`

For more information about the Bootstrap URL address formats, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

The client application then uses the Bootstrap object or the INS bootstrapping mechanism to obtain references to the objects in a BEA Tuxedo domain. Once the Bootstrap object is instantiated, the `resolve_initial_references()` method is invoked by the client application, passing in a `string id`, to obtain a reference to the objects in the BEA Tuxedo domain that provide CORBA services.

Figure 2-3 illustrates how the Bootstrap object or INS mechanism operates in a BEA Tuxedo domain.

Figure 2-3 How the Bootstrap Object or INS Operates


CORBA Client Application
Bootstrap Object or INS
resolve_initial_references()
FactoryFinder Object Reference
TransactionCurrent or TransactionFactory Object Reference
SecurityCurrent or PrincipalAuthenticator Object Reference
InterfaceRepository Object Reference
NameService Object Reference
NotificationService Object Reference
InterfaceRepository Object Reference
Tobj_SimpleEventsService Object Reference
IIOP Listener/Handler
BEA Tuxedo Domain
FactoryFinder Object
TransactionCurrent Object
TransactionFactory Object
SecurityCurrent Object
PrincipalAuthenticator Object
InterfaceRepository Object
NameService Object
NotificationService Object
Tobj_SimpleEventsService Object

IIOP Listener/Handler

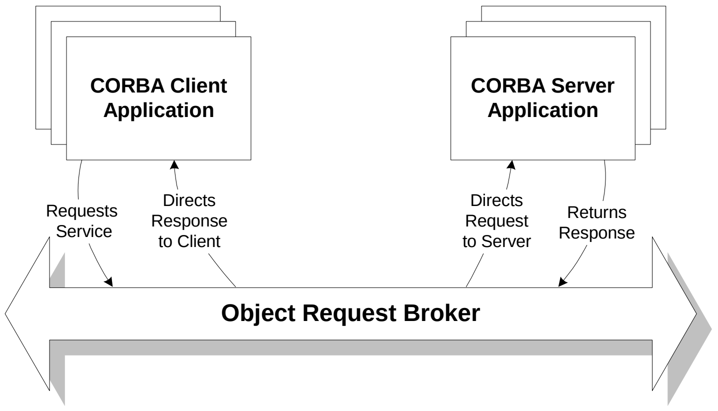
The IIOP Listener/Handler is a process that receives the CORBA client request, which is sent using IIOP, and delivers that request to the appropriate CORBA server application. The IIOP Listener/Handler serves as a communication concentrator, providing a critical scalability feature. The IIOP Listener/Handler removes from the CORBA server application the burden of maintaining client connections. For information about configuring the IIOP Listener/Handler, see *Setting Up a BEA Tuxedo Application* and the description of the ISL command in the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

ORB

The ORB serves as an intermediary for requests that CORBA client applications send to CORBA server applications, so that these applications do not need to contain information about each other. The ORB is responsible for all the mechanisms required to find the implementation that can satisfy the request, to prepare an object's implementation to receive the request, and to communicate the data that makes up the request. The BEA Tuxedo CORBA product includes a C++ client/server ORB.

Figure 2-4 shows the relationship between an ORB, a CORBA client application, and a CORBA server application.

Figure 2-4 The ORB in a CORBA Client/Server Environment



When the client application uses IIOP to send a request to the BEA Tuxedo domain, the ORB performs the following functions:

- Validates each request and its arguments to ensure that the client application supplied all the required arguments.

- Manages the mechanisms required to find the CORBA object that can satisfy the request from the CORBA client application. To do this, the ORB interacts with the Portable Object Adapter (POA). The POA prepares an object's implementation to receive the request and communicates the data in the request.
- Marshals data. The ORB on the client machine writes the data associated with the request into a standard form. The ORB receives this data and converts it into the format appropriate for the machine on which the server application is running. When the server application sends data back to the client application, the ORB marshals the data back into its standard form and sends it back to the ORB on the client machine.

TP Framework

The TP Framework provides a programming model that achieves high levels of performance while shielding the application programmer from the complexities of the CORBA interfaces. The TP Framework supports the rapid construction of CORBA applications, which makes it easier for application programmers to adhere to design patterns associated with successful TP applications.

The TP Framework interacts with the Portable Object Adapter (POA) and the CORBA application, thus eliminating the need for direct POA calls in an application. In addition, the TP Framework integrates transactions and state management into the BEA Tuxedo CORBA application.

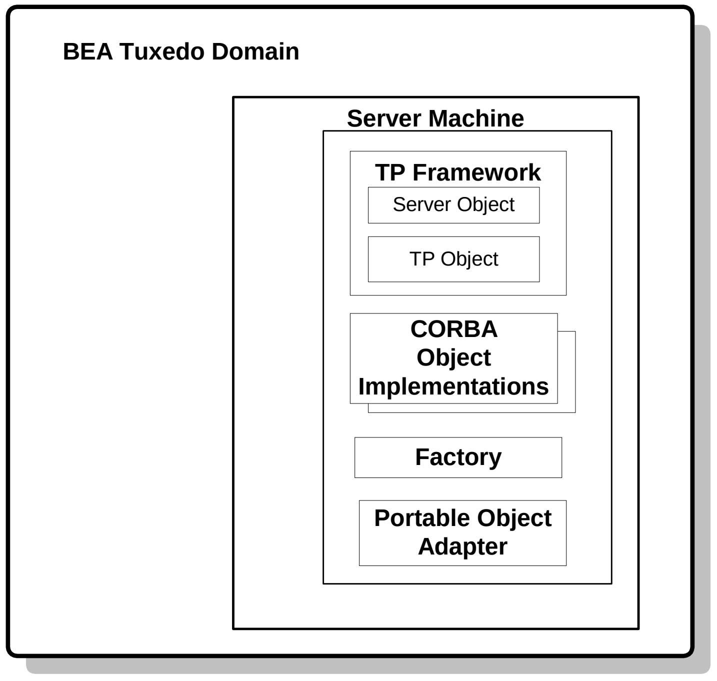
The application programmer uses an application programming interface (API) that automates many of the functions required in a standard CORBA application. The application programmer is responsible only for writing the business logic of the CORBA application and overriding default actions provided by the TP Framework.

The TP Framework API provides routines that perform the following functions required by a CORBA application:

- Initializing the CORBA server application and executing startup and shutdown routines
- Creating object references
- Registering and unregistering object factories
- Managing objects and object state
- Tying the CORBA server application to BEA Tuxedo CORBA system resources
- Getting and initializing the ORB
- Performing object housekeeping

The TP Framework ensures that the execution of a client request takes place in a coordinated, predictable manner. The TP Framework calls the objects and services available in the BEA Tuxedo application at the appropriate time, in the correct sequence. In addition, the TP Framework maximizes the reuse of system resources by objects. Figure 2-5 illustrates the TP Framework.

Figure 2-5 The TP Framework



The TP Framework is not a single object, but is rather a collection of objects that work together to manage the CORBA objects that contain and implement the data and business logic in your CORBA application.

One of the TP Framework objects is the Server object. The Server object is a user-written programming entity that implements operations that perform tasks such as initializing and releasing the server application. For server applications the TP Framework instantiates the CORBA objects needed to satisfy a client request.

If a client request arrives requiring an object that is not currently active and in memory in the server application, the TP Framework coordinates all the operations that are required to instantiate the object. This includes coordinating with the ORB and the POA to get the client request to the appropriate object implementation code.

How BEA Tuxedo CORBA Client and Server Applications Interact

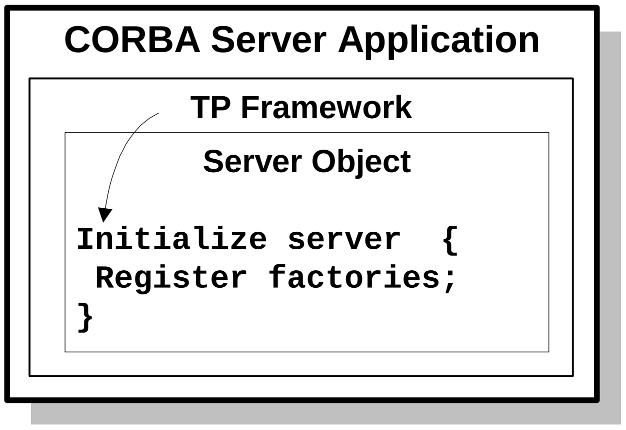
The interaction between BEA Tuxedo CORBA client and server applications includes the following steps:

1. The CORBA server application is initialized.
2. The CORBA client application is initialized.
3. The CORBA client application authenticates itself to the BEA Tuxedo domain.
4. The CORBA client application obtains a reference to the CORBA object needed to execute its business logic.
5. The CORBA client application invokes an operation on the CORBA object.

The following topics describe what happens during each step.

Step 1: The CORBA Server Application Is Initialized

The system administrator enters the `tmboot` command on a machine in the BEA Tuxedo domain to start the BEA Tuxedo CORBA server application. The TP Framework invokes the `initialize()` operation in the `Server` object to initialize the server application.

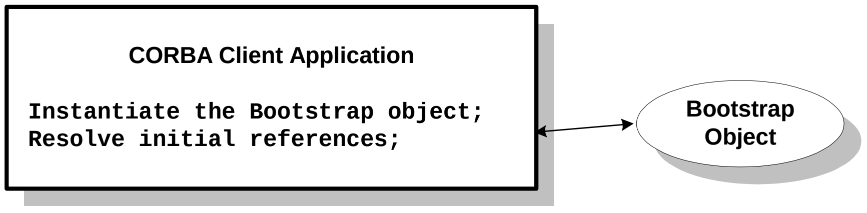


During the initialization process, the Server object does the following:

1. Uses the Bootstrap object or INS to obtain a reference to the FactoryFinder object.
2. Typically registers any factories with the FactoryFinder object.
3. Optionally gets an object reference to the ORB.
4. Performs any process-wide initialization.

Step 2: The CORBA Client Application Is Initialized

During initialization, the CORBA client application obtains initial references to the objects available in the BEA Tuxedo domain.


CORBA Client Application
Instantiate the Bootstrap object;
Resolve initial references;
Bootstrap Object

The Bootstrap object returns references to the FactoryFinder, SecurityCurrent, TransactionCurrent, NameService, and InterfaceRepository objects in the BEA Tuxedo domain.

Step 3: The CORBA Client Application Authenticates Itself to the BEA Tuxedo Domain

If the BEA Tuxedo domain has a security model in effect, the CORBA client application needs to authenticate itself to the BEA Tuxedo domain before it can invoke any operations in the CORBA server application. To authenticate itself to the BEA Tuxedo domain using authentication, the CORBA client application completes these steps:

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object.
2. Invokes the `logon()` operation of the PrincipalAuthenticator object, which is retrieved from the SecurityCurrent object.

Note: For information about using certificate based authentication, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

Step 4: The CORBA Client Application Obtains a Reference to the CORBA Object Needed to Execute Its Business Logic

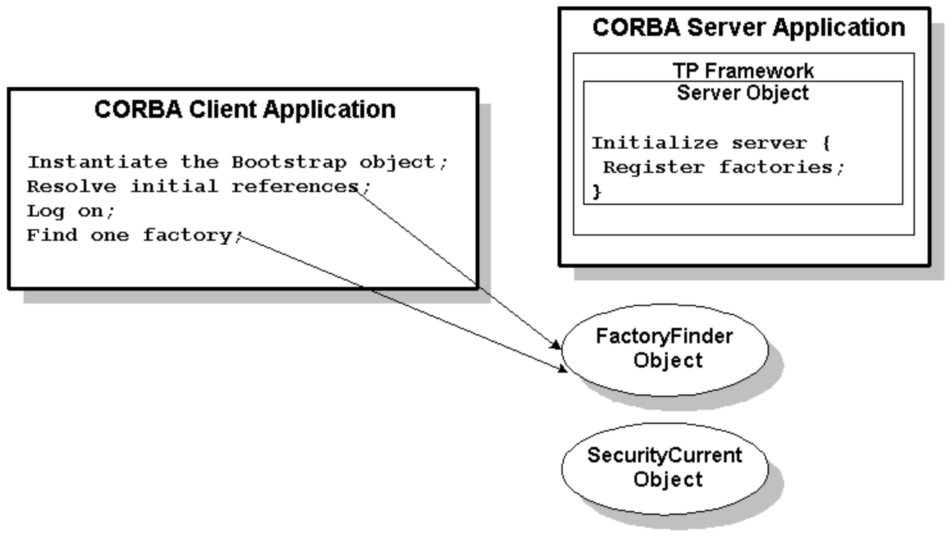
The CORBA client application needs to perform the following steps:

1. Obtain a reference to the factory for the object it needs.

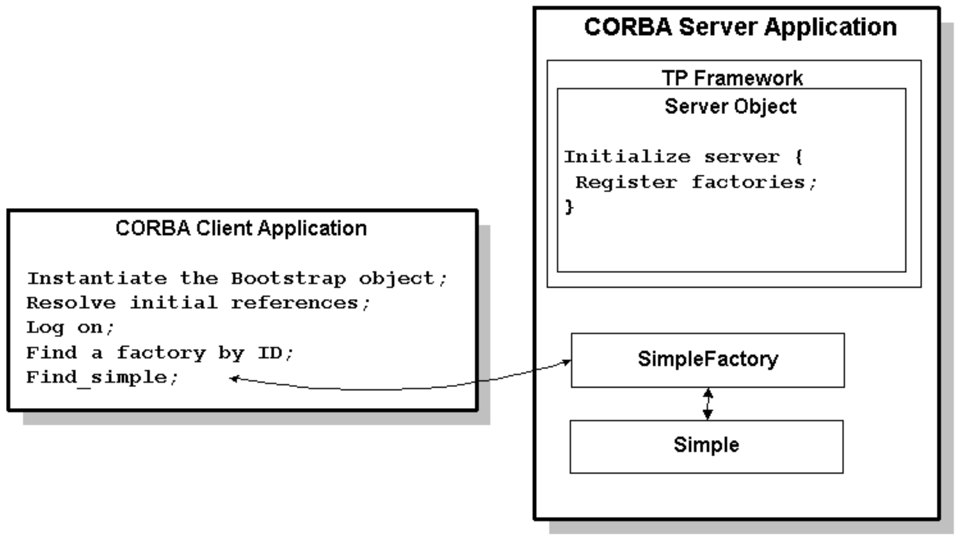
   For example, the client application needs a reference to the SimpleFactory object. The client application obtains this factory reference from the FactoryFinder object, shown in the following figure.

2. Invoke the SimpleFactory object to get a reference to the `Simple` object.

   If the SimpleFactory object is not active, the TP Framework instantiates the SimpleFactory object by invoking the `Server::create_servant` method on the Server object, shown in the following figure.



3. The TP Framework invokes the `activate_object()` and `find_simple()` operations on the SimpleFactory object to get a reference to the Simple object, shown in the following figure.

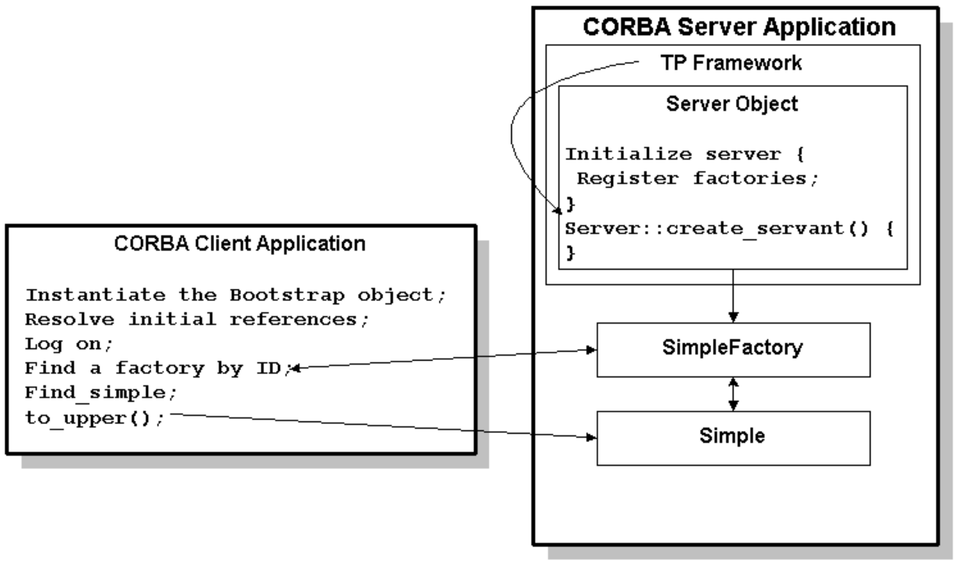


The SimpleFactory object then returns the object reference to the `Simple` object to the client application.

Note: Because the TP Framework activates objects by default, the Simpapp sample application does not explicitly use the `activate_object()` operation for the SimpleFactory object.

Step 5: The CORBA Client Application Invokes an Operation on the CORBA Object

Using the reference to the CORBA object that the factory has returned to the client application, the client application invokes an operation on the object. For example, now that the client application has an object reference to the Simple object, the client application can invoke the `to_upper()` operation on it. The instance of the Simple object required for the client request is created as shown in the following figure.



CHAPTER 3

Developing BEA Tuxedo CORBA Applications

This topic includes the following sections:

- Overview of the Development Process for BEA Tuxedo CORBA Applications
- The Simpapp Sample Application
- Step 1: Write the OMG IDL Code
- Step 2: Generate CORBA client Stubs and Skeletons
- Step 3: Write the CORBA Server Application
- Step 4: Write the CORBA Client Application
- Step 5: Create an XA Resource Manager
- Step 6: Create a Configuration File
- Step 7: Create the TUXCONFIG File
- Step 8: Compile the CORBA Server Application
- Step 9: Compile the CORBA Client Application
- Step 10: Start the BEA Tuxedo CORBA Application
- Additional BEA Tuxedo CORBA Sample Applications

For an in-depth discussion of creating BEA Tuxedo CORBA client and server applications, see the following in the BEA Tuxedo online documentation:

- *Creating CORBA Client Applications*
- *Creating CORBA Server Applications*

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

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

Overview of the Development Process for BEA Tuxedo CORBA Applications

Table 3-1 outlines the development process for BEA Tuxedo CORBA applications.

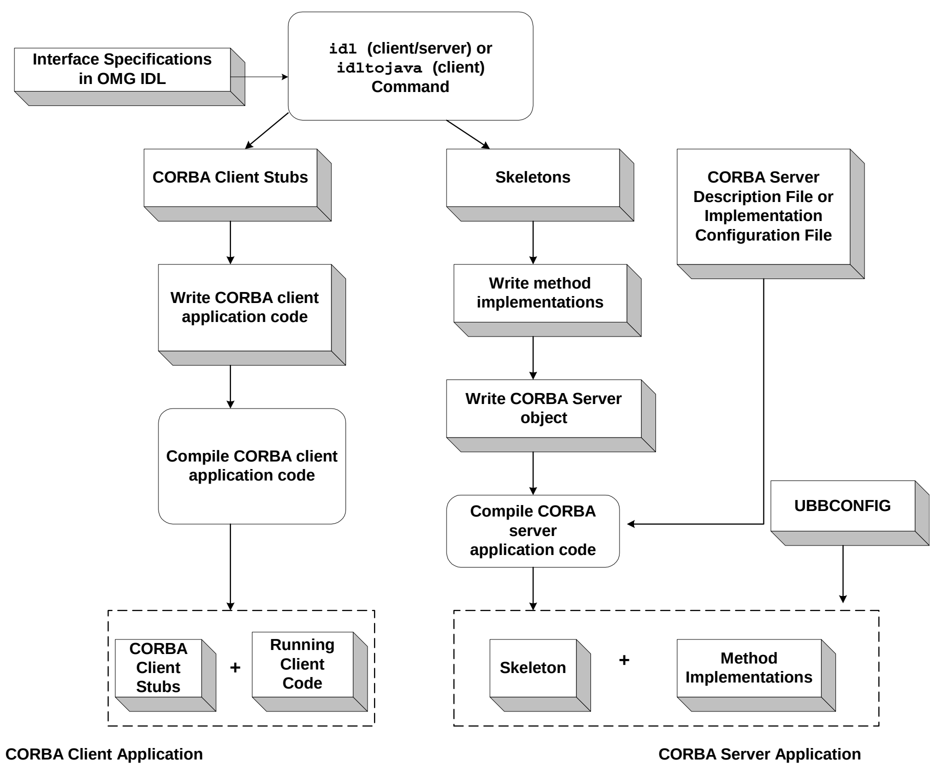
Table 3-1 Development Process for BEA Tuxedo CORBA Applications

Step	Description
1	Write the Object Management Group (OMG) Interface Definition Language (IDL) code for each CORBA interface you want to use in your BEA Tuxedo application.
2	Generate the CORBA client stubs and the skeletons.
3	Write the CORBA server application.
4	Write the CORBA client application.
5	Create an XA resource manager.
6	Create a configuration file.
7	Create a `TUXCONFIG` file.
8	Compile the CORBA server application.
9	Compile the CORBA client application.
10	Start the BEA Tuxedo CORBA application.

The steps in the development process are described in the following sections.

Figure 3-1 illustrates the process for developing BEA Tuxedo CORBA applications.

Figure 3-1 Development Process for BEA Tuxedo CORBA Applications


Interface Specifications in OMG IDL
idl (client/server) or idltojava (client) Command
CORBA Client Stubs
Skeletons
CORBA Server Description File or Implementation Configuration File
Write CORBA client application code
Write method implementations
Write CORBA Server object
Compile CORBA client application code
UBBCONFIG
Compile CORBA server application code
CORBA Client Stubs
+
Running Client Code
Skeleton
+
Method Implementations
CORBA Client Application
CORBA Server Application

The Simpapp Sample Application

Throughout this topic, the Simpapp sample application is used to demonstrate the development steps.

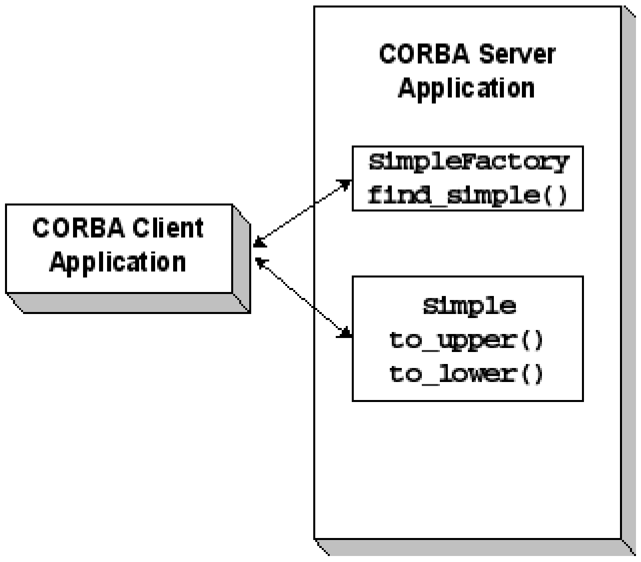
The CORBA server application in the Simpapp sample application provides an implementation of a CORBA object that has the following two methods:

– The `upper` method accepts a string from the CORBA client application and converts the string to uppercase letters.

– The `lower` method accepts a string from the CORBA client application and converts the string to lowercase letters.

Figure 3-2 illustrates how the Simpapp sample application works.

Figure 3-2 Simpapp Sample Application



The source files for the Simpapp sample application are located in the `$TUXDIR\samples\corba\simpapp` directory of the BEA Tuxedo software. Instructions for building and running the Simpapp sample applications are in the `Readme.txt` file in the same directory.

Note: The Simpapp sample applications demonstrate building CORBA C++ client and server applications.

BEA Tuxedo offers a suite of sample applications that demonstrate and aid in the development of BEA Tuxedo CORBA applications. For an overview of the available sample applications, see *Samples* in the BEA Tuxedo online documentation.

Step 1: Write the OMG IDL Code

The first step in writing a BEA Tuxedo CORBA application is to specify all of the CORBA interfaces and their methods using the Object Management Group (OMG) Interface Definition

Language (IDL). An interface definition written in OMG IDL completely defines the CORBA interface and fully specifies each operation's arguments. OMG IDL is a purely declarative language. This means that it contains no implementation details. Operations specified in OMG IDL can be written in and invoked from any language that provides CORBA bindings.

The Simpapp sample application implements the CORBA interfaces listed in Table 3-2.

Table 3-2 CORBA Interfaces for the Simpapp Sample Application

Interface	Description	Operation
`SimpleFactory`	Creates object references to the `Simple` object	`find_simple()`

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

Listing 3-1 shows the `simple.idl` file that defines the CORBA interfaces in the Simpapp sample application.

Listing 3-1 OMG IDL Code for the Simpapp 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)
     void to_upper(inout string val);
};

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

Step 2: Generate CORBA client Stubs and Skeletons

The interface specification defined in OMG IDL is used by the IDL compiler to generate CORBA client stubs for the CORBA client application, and skeletons for the CORBA server application. The CORBA client stubs are used by the CORBA client application for all operation invocations. You use the skeleton, along with the code you write, to create the CORBA server application that implements the CORBA objects.

During the development process, use one of the following commands to compile the OMG IDL file and produce CORBA client stubs and skeletons for BEA Tuxedo CORBA client and server applications:

- If you are creating CORBA C++ client and server applications, use the `idl` command. For a description of the `idl` command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Table 3-3 lists the files that are created by the `idl` command.

Table 3-3 Files Created by the idl Command

File	Default Name	Description
CORBA client stub file	*`application`*`_c.cpp`	Contains generated code for sending a request.
CORBA 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. During run time, the skeleton maps CORBA client requests to the appropriate operation in the CORBA server application.
Skeleton header file	*`application`*`_s.h`	Contains the skeleton class definitions.
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.

Step 3: Write the CORBA Server Application

The BEA Tuxedo software supports CORBA C++ server applications. The steps for creating CORBA server applications are:

1. Write the methods that implement the operations for each interface.
2. Create the CORBA server object.
3. Define object activation policies.
4. Create and register a factory.
5. Release the CORBA server application.

Writing the Methods That Implement the Operations for Each Interface

After you compile the OMG IDL file, you need to write methods that implement the operations for each interface in the file. An implementation file contains the following:

- Method declarations for each operation specified in the OMG IDL file
- Your application's business logic
- Constructors for each interface implementation (implementing these is optional)
- The `activate_object()` and `deactivate_object()` methods (optional)

  Within the `activate_object()` and `deactivate_object()` methods, you write code that performs any particular steps related to activating or deactivating the object. For more information, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

You can write the implementation file manually. The `idl` command provides an option for generating a template for implementation files.

Listing 3-2 includes the C++ implementation of the `Simple` and `SimpleFactory` interfaces in the Simpapp sample application.

Listing 3-2 C++ Implementation of the Simple and SimpleFactory Interfaces

```
// Implementation of the Simple_i::to_lower method which converts
// a string to lower case.

char* Simple_i::to_lower(const char* value)
{
     CORBA::String_var var_lower = CORBA::string_dup(value);
     for (char* ptr = var_lower; ptr && *ptr; ptr++) {
         *ptr = tolower(*ptr);
     }
     return var_lower._retn();
}
// Implementation of the Simple_i::to_upper method which converts
// a string to upper case.

void Simple_i::to_upper(char*& value1)
{
     CORBA::String_var var_upper = value1;
     var_upper = CORBA::string_dup(var_upper.in());
     for (char* ptr = var_upper; ptr && *ptr; ptr++) {
         *ptr = toupper(*ptr);
     }
     value = var_upper._retn();
}
// Implementation of the SimpleFactory_i::find_simple method which
// creates an object reference to a Simple object.

Simple_ptr SimpleFactory_i::find_simple()
{
     CORBA::Object_var var_simple_oref =
         TP::create_object_reference(
             _tc_Simple->id(),
             "simple",
             CORBA::NVList::_nil()
         );
     }
```

Creating the CORBA server Object

The Server object performs the following tasks:

- Initializes the CORBA server application, including registering factories, allocating resources needed by the CORBA server application, and, if necessary, opening an XA resource manager.
- Performs CORBA server application shutdown and cleanup procedures.
- Instantiates CORBA objects needed to satisfy CORBA client requests.

In CORBA server applications, the Server object is already instantiated and a header file for the Server object is available. You implement methods that initialize and release the server application, and, if desired, create servant objects.

Listing 3-3 includes the C++ code from the Simpapp sample application for the Server object.

Listing 3-3 CORBA C++ Server Object

```
static CORBA::Object_var static_var_factory_reference;

// Method to start up the server


CORBA::Boolean Server::initialize(int argc, char* argv[])
{
       // Create the Factory Object Reference

       static_var_factory_reference =
           TP::create_object_reference(
              _tc_SimpleFactory->id(),
              "simple_factory",
              CORBA::NVList::_nil()
           );
       // Register the factory reference with the FactoryFinder

       TP::register_factory(
            static_var_factory_reference.in(),
            _tc_SimpleFactory->id()
        );
        return CORBA_TRUE;
}
// Method to shutdown the server

void Server::release()
{
// Unregister the factory.

  try {
       TP::unregister_factory(
           static_var_factory_reference.in(),
           _tc_SimpleFactory->id()
       );
  }
  catch (...) {
       TP::userlog("Couldn't unregister the SimpleFactory");
  }
}
// Method to create servants

Tobj_Servant Server::create_servant(const char*
 interface_repository_id)
{
       if (!strcmp(interface_repository_id,
       _tc_SimpleFactory->id())) {
              return new SimpleFactory_i();
       }
       if (!strcmp(interface_repository_id,
       _tc_Simple->id())) {
              return new Simple_i();
       }
       return 0;
}
```

Defining an Object’s Activation Policies

As part of CORBA server development, you determine what events cause an object to be activated and deactivated by assigning object activation policies.

For CORBA server applications, specify object activation policies in the Implementation Configuration File (ICF). A template ICF file is created by the `genicf` command.

Note: You also define transaction policies in the ICF file. For information about using transactions in your BEA Tuxedo CORBA application, see *Using CORBA Transactions* in the BEA Tuxedo online documentation.

The BEA Tuxedo software supports the activation policies listed in Table 3-4.

Table 3-4 Activation Policies

Activation Policy	Description
`method`	Causes the object to be active only for the duration of the invocation on one of the object’s operations. This is the default activation policy.
`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 method `TP::deactivateEnable()` (C++) has been invoked on the object.

The `Simple` interface in the Simpapp sample application is assigned the default activation policy of method. For more information about managing object state and defining object activation policies, see *Creating CORBA Server Applications* in the BEA Tuxedo online documentation.

Creating and Registering a Factory

If your CORBA server application manages a factory that you want CORBA client applications to be able to locate easily, you need to write the code that registers that factory with the FactoryFinder object.

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

1. Create an object reference to the factory.

   You include an invocation to the `create_object_reference()` method, specifying the Interface Repository ID of the factory's OMG IDL interface or the object ID (OID) in string format. In addition, you can specify routing criteria.

2. Register the factory with the BEA Tuxedo domain.

   Use the `register_factory()` method to register the factory with the FactoryFinder object in the BEA Tuxedo domain. The `register_factory()` method requires the object reference for the factory and a string identifier.

Listing 3-4 includes the code from the Simpapp sample application that creates and registers a factory.

Listing 3-4 Example of Creating and Registering a Factory

```
...
CORBA::Object_var v_reg_oref =
       TP:create_object_reference(
              _tc.SimpleFactory->id(),    //Factory Interface ID
              “simplefactory”,            //Object ID
              CORBA::NVList::_nil()       //Routing Criteria
       );
       TP::register_factory(
              CORBA::Object_var v_reg_oref.in(),
              _tc_SimpleFactory->id(),
       );
...
```

In Listing 3-4, notice the following:

- `tc.SimpleFactory->id()` specifies the SimpleFactory object's Interface Repository ID by extracting it from its typecode.
- `CORBA::NVList::_nil()` specifies that no routing criteria are used, with the result that an object reference created for the Simple object is routed to the same group as the SimpleFactory object that created the object reference.

Releasing the CORBA Server Application

You need to include code in your CORBA server application to perform a graceful shutdown of the CORBA server application. The `release()`method is provided for that purpose. Within the `release()` method, you may perform any application-specific cleanup tasks that are specific to the CORBA server application, such as:

- Unregistering object factories managed by the CORBA server application
- Deallocating resources
- Closing any databases
- Closing an XA resource manager

Once a CORBA server application receives a request to shut down, the CORBA server application can no longer receive requests from other remote objects. This has implications on the order in which CORBA 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 `release()` method to the first server process.

During server shutdown, you may want to unregister each of the server application's factories. The invocation of the `unregister_factory()` method should be one of the first actions in the `release()` implementation. The `unregister_factory()` method 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 the Interface Repository ID of the object's OMG IDL interface

Listing 3-5 includes C++ code that releases a server application and unregisters the factories in the CORBA server application.

Listing 3-5 Example of Releasing a BEA Tuxedo CORBA server Application

```
...
public void release()
{
       TP::unregister_factory(
              factory_reference.in(),
              SimpleFactoryHelper->id
              );
}
...
```

Step 4: Write the CORBA Client Application

The BEA Tuxedo software supports the following types of CORBA client applications:

- CORBA C++

The steps for creating CORBA client applications are as follows:

1. Initialize the ORB.
2. Use the Bootstrap object or the CORBA INS bootstrapping mechanism to establish communication with the BEA Tuxedo domain.
3. Resolve initial references to the FactoryFinder environmental object.
4. Use a factory to get an object reference for the desired CORBA object.
5. Invoke methods on the CORBA object.

The CORBA client development steps are illustrated in Listing 3-6 which include code from the Simpapp sample application. In the Simpapp sample application, the CORBA client application uses a factory to get an object reference to the Simple object and then invokes the `to_upper()` and `to_lower()` methods on the Simple object.

Listing 3-6 CORBA Client Application from the Simpapp Sample Application

```
int main(int argc, char* argv[])
{
    try {
       // Initialize the ORB
       CORBA::ORB_var var_orb = CORBA::ORB_init(argc, argv, "");

       // Create the Bootstrap object
       Tobj_Bootstrap bootstrap(var_orb.in(), "");

       // Use the Bootstrap object to find the FactoryFinder
       CORBA::Object_var var_factory_finder_oref =
         bootstrap.resolve_initial_references("FactoryFinder");

       // Narrow the FactoryFinder
       Tobj::FactoryFinder_var var_factory_finder_reference =
        Tobj::FactoryFinder::_narrow
        (var_factory_finder_oref.in());

       // Use the factory finder to find the Simple factory
       CORBA::Object_var var_simple_factory_oref =
       var_factory_finder_reference->find_one_factory_by_id(
       _tc_SimpleFactory->id()
       );

       // Narrow the Simple factory
       SimpleFactory_var var_simple_factory_reference =
         SimpleFactory::_narrow(
                                var_simple_factory_oref.in());

       // Find the Simple object
       Simple_var var_simple =
         var_simple_factory_reference->find_simple();

       // Get a string from the user
       cout << "String?";
       char mixed[256];
```

```
        cin >> mixed;

        // Convert the string to upper case :
        CORBA::String_var var_upper = CORBA::string_dup(mixed);
        var_simple->to_upper(var_upper.inout());
        cout << var_upper.in() << endl;

        // Convert the string to lower case
        CORBA::String_var var_lower = var_simple->to_lower(mixed);
        cout << var_lower.in() << endl;

        return 0;
        }
}
```

Step 5: Create an XA Resource Manager

When using transactions in a BEA Tuxedo CORBA application, you need to create a CORBA server process for the resource manager that interacts with a database on behalf of the BEA Tuxedo CORBA application. The resource manager you use must conform to the X/OPEN XA specification and you need the following information about the resource manager:

- The name of the structure of type `xa_switch_t` that contains the name of the XA resource manager.
- Flags indicating the capabilities of the XA resource manager and function pointers for the actual XA functions.
- The name of the object files that provide the services of the XA interface.
- The commands needed to open and close the XA resource manager. This information is specified in the `OPENINFO` and `CLOSEINFO` parameters in the `UBBCONFIG` configuration file.

When integrating a new XA resource manager into the BEA Tuxedo system, the file `$TUXDIR/udataobj/RM` must be updated to include information about the XA resource manager. The information is used to include the correct libraries for the XA resource manager and to set up the interface between the transaction manager and the XA resource manager automatically and correctly. The format of this file is as follows:

rm_name:*rm_structure_name*:*library_names*

where *rm_name* is the name of the XA resource manager, *rm_structure_name* is the name of the `xa_switch_t` structure that defines the name of the XA resource manager, and *library_names* is the list of the object files for the XA resource manager. White space (tabs and/or spaces) is allowed before and after each of the values and may be embedded within the *library_names*. The colon (:) character may not be embedded within any of the values. Lines beginning with a pound sign (#) are treated as comments and are ignored.

Use the `buildtms` command to build a server process for the XA resource manager. The files that result from the `buildtms` command need to be installed in the `$TUXDIR/bin` directory.

For more information about the `buildtms` command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Step 6: Create a Configuration File

Because the BEA Tuxedo software offers great flexibility and many options to application designers and programmers, no two CORBA applications are alike. An application, for example, may be small and simple (a single client and server running on one machine) or complex enough to handle transactions among thousands of client and server applications. For this reason, for every BEA Tuxedo CORBA application being managed, the system administrator must provide a configuration file that defines and manages the components (for example, domains, server applications, client applications, and interfaces) of that application.

When system administrators create a configuration file, they are describing the BEA Tuxedo CORBA application using a set of parameters that the BEA Tuxedo software interprets to create a runnable version of the application. During the setup phase of administration, the system administrator's job is to create a configuration file. The configuration file contains the sections listed in Table 3-5.

Table 3-5 Sections in the Configuration File for BEA Tuxedo CORBA Applications

Sections in the Configuration File	Description
`RESOURCES`	Defines defaults (for example, user access and the main administration machine) for the BEA Tuxedo CORBA application.
`MACHINES`	Defines hardware-specific information about each machine running in the BEA Tuxedo CORBA application.

Table 3-5 Sections in the Configuration File for BEA Tuxedo CORBA Applications (Continued)

Sections in the Configuration File	Description
GROUPS	Defines logical groupings of server applications or CORBA interfaces.
SERVERS	Defines the server application processes (for example, the Transaction Manager) used in the BEA Tuxedo CORBA application.
SERVICES	Defines parameters for services provided by the BEA Tuxedo application.
INTERFACES	Defines information about the CORBA interfaces in the BEA Tuxedo CORBA application.
ROUTING	Defines routing criteria for the BEA Tuxedo CORBA application.

Listing 3-7 shows the configuration file for the Simpapp sample application.

Listing 3-7 Configuration File for Simpapp Sample Application

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

*MACHINES
       "PCWIZ"
       LMID         = SITE1
       APPDIR       = "C:\TUXDIR\MY_SIM~1"
       TUXCONFIG    = "C:\TUXDIR\MY_SIM~1\results\tuxconfig"
       TUXDIR       = "C:\TUXDIR"
       MAXWSCLIENTS = 10

*GROUPS
       SYS_GRP
       LMID    = SITE1
       GRPNO   = 1
       APP_GRP
       LMID    = SITE1
       GRPNO   = 2
*SERVERS
       DEFAULT:
          RESTART = Y
          MAXGEN  = 5
       TMSYSEVT
          SRVGRP  = SYS_GRP
          SRVID   = 1
       TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 2
          CLOPT   = "-A -- -N -M"
       TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 3
          CLOPT   = "-A -- -N"
       TMFFNAME
          SRVGRP  = SYS_GRP
          SRVID   = 4
          CLOPT   = "-A -- -F"
       simple_server
          SRVGRP  = APP_GRP
          SRVID   = 1
          RESTART = N
       ISL
          SRVGRP  = SYS_GRP
          SRVID   = 5
          CLOPT   = "-A -- -n //PCWIZ:2468"
*SERVICES
```

Step 7: Create the TUXCONFIG File

There are two forms of the configuration file:

- An ASCII version of the file, created and modified with any editor. Throughout the BEA Tuxedo documentation, the ASCII version of the configuration file is referred to as the `UBBCONFIG` file. You can choose any name for the configuration file.
- The `TUXCONFIG` file, a binary version of the `UBBCONFIG` file created using the `tmloadcf` command. When the `tmloadcf` command is executed, the environment variable `TUXCONFIG` must be set to the name and directory location of the `TUXCONFIG` file. The `tmloadcf` command converts the configuration file to binary form and writes it to the location specified in the command.

For more information about the `tmloadcf` command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Step 8: Compile the CORBA Server Application

You use the `buildobjserver` command to compile and link C++ 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.

When you create a server application to support multithreading, you must specify the `-t` option on the `buildobjserver` command when you build the application. For complete information on creating a server application to support multithreading, see *Creating CORBA Server Applications*.

Step 9: Compile the CORBA Client Application

The final step in the development of the CORBA client application is to produce the executable client application. To do this, you need to compile the code and then link against the client stub.

When creating CORBA C++ client applications, use the `buildobjclient` command to construct a BEA Tuxedo CORBA client application executable. The command combines the

CORBA client stubs for interfaces that use static invocation, and the associated header files, with the standard BEA Tuxedo libraries to form a CORBA client executable. For the syntax of the `buildobjclient` command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Step 10: Start the BEA Tuxedo CORBA Application

Use the `tmboot` command to start the server processes in your BEA Tuxedo CORBA application. The CORBA application is usually booted from the machine designated as the `MASTER` in the `RESOURCES` section of the `UBBCONFIG` file.

For the `tmboot` command to find executables, the BEA Tuxedo system processes must be located in the `$TUXDIR/bin` directory. Server applications should be in `APPDIR,` as specified in the configuration file.

When booting server applications, the `tmboot` command uses the `CLOPT`, `SEQUENCE`, `SRVGRP`, `SRVID`, and `MIN` parameters from the configuration file. Server applications are booted in the order in which they appear in the configuration file.

For more information about using the `tmboot` command, see *File Formats, Data Descriptions, MIBs, and System Processes Reference* in the BEA Tuxedo online documentation.

Additional BEA Tuxedo CORBA Sample Applications

Sample applications demonstrate the tasks involved in developing a BEA Tuxedo CORBA application, and provide sample code that can be used by CORBA client and server programmers to build their own BEA Tuxedo CORBA application. Code from the sample applications are used throughout the information topics in the BEA Tuxedo product to illustrate the development and administrative steps.

Table 3-6 describes the additional BEA Tuxedo CORBA sample applications.

Table 3-6 The BEA Tuxedo CORBA Sample Applications

BEA Tuxedo CORBA Sample Application	Description
Simpapp	Provides a CORBA C++ client application and a C++ server application. The C++ server application contains two operations that manipulate strings received from the C++ client application.
Basic	Describes how to develop BEA Tuxedo CORBA client and server applications and configure the BEA Tuxedo application. Building C++ server applications and CORBA C++ applications, CORBA client applications are demonstrated.
Security	Demonstrates adding BEA Tuxedo authentication to a BEA Tuxedo CORBA application. For information about building and running the Security sample application, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.
Transactions	Adds transactional objects to the CORBA C++ server application and CORBA client applications in the Basic sample application. The Transactions sample application demonstrates how to use the Implementation Configuration File (ICF) to define transaction policies for CORBA objects. For information about building and running the Transactions sample application, see *Using CORBA Transactions* in the BEA Tuxedo online documentation.
Wrapper	Demonstrates how to wrap an existing BEA Tuxedo ATMI application as a CORBA object.
Production	Demonstrates replicating server applications, creating stateless objects, and implementing factory-based routing in server applications.

Table 3-6 The BEA Tuxedo CORBA Sample Applications (Continued)

BEA Tuxedo CORBA Sample Application	Description
Secure Simpapp	Implements the necessary development and administrative changes to the Simpapp sample application to support certificate authentication. For information about building and running the Secure Simpapp sample application, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.
Introductory Events	Demonstrates how to use joint CORBA client/server applications and callback objects to implement events in a BEA Tuxedo CORBA application. The C++ version uses the BEA Simple Events API. For information about building and running the Introductory Events sample application, see *Using the CORBA Notification Service* in the BEA Tuxedo online documentation.
Advanced Events	Provides a more complex implementation of events in a BEA Tuxedo CORBA application with transient and persistent subscriptions and data filtering. The C++ version uses the Advanced CosNotification API. For information about building and running the Advanced Events sample application, see *Using the CORBA Notification Service* in the BEA Tuxedo online documentation.

CHAPTER 4

Using Security

This topic includes the following sections:

- Overview of the Security Service
- How Security Works
- The Security Sample Application
- Development Steps

Notes: This chapter describes how to use authentication. For a complete description of all the security features available in the CORBA security environment and instructions for implementing the features, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

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

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

Overview of the Security Service

The CORBA environment in the BEA Tuxedo product offers a security model based on the CORBA Services Security Service. The BEA Tuxedo CORBA security model implements the authentication portion of the CORBA Services Security Service.

In the CORBA environment security information is defined on a domain basis. The security level for the domain is defined in the configuration file. Client applications use the SecurityCurrent object to provide the necessary authentication information to log on to the BEA Tuxedo domain.

The following levels of authentication are provided:

- `TOBJ_NOAUTH`

  No authentication is needed; however, the client application may still authenticate itself, and may specify a username and a client application name, but no password.

- `TOBJ_SYSAUTH`

  The client application must authenticate itself to the BEA Tuxedo domain and must specify a username, client application name, and application password.

- `TOBJ_APPAUTH`

  In addition to the `TOBJ_SYSAUTH` information, the client application must provide application-specific information. If the default BEA Tuxedo CORBA authentication service is used in the application configuration, the client application must provide a user password; otherwise, the client application provides authentication data that is interpreted by the custom authentication service in the application.

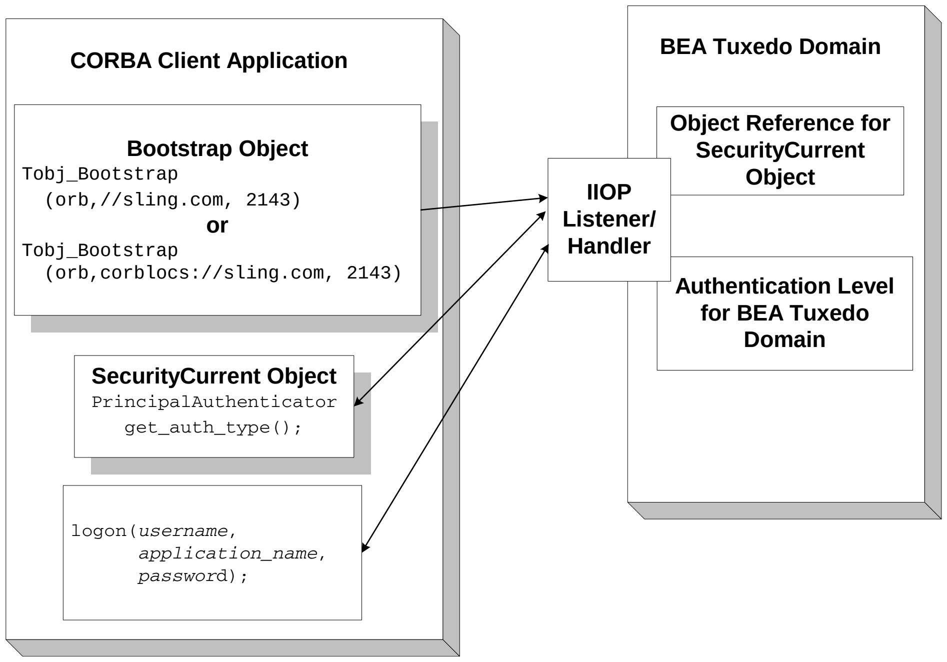
Note: If a client application is not authenticated and the security level is `TOBJ_NOAUTH`, the IIOP Listener/Handler of the BEA Tuxedo domain registers the client application with the username and client application name sent to the IIOP Listener/Handler.

In the BEA Tuxedo CORBA security environment, only the PrincipalAuthenticator and Credentials properties on the SecurityCurrent object are supported. For a description of the `SecurityLevel1::Current` and `SecurityLevel2::Current` interfaces, see the *CORBA Programming Reference* in the BEA Tuxedo online documentation.

How Security Works

Figure 4-1 illustrates how CORBA security works in a BEA Tuxedo domain.

Figure 4-1 How CORBA Security Works on BEA Tuxedo Domain



The steps are as follows:

1. The client application uses the Bootstrap object to return an object reference to the SecurityCurrent object for the BEA Tuxedo domain.
2. The client application obtains the PrincipalAuthenticator.
3. The client application uses the `Tobj::PrincipalAuthenticator::get_auth_type()` method to get the authentication level for the BEA Tuxedo domain.
4. The proper authentication level is returned to the client application.
5. The client application uses the `Tobj::PrincipalAuthenticator::logon()` method to log on to the BEA Tuxedo domain with the proper authentication information.

Note: BEA Tuxedo CORBA also supports the use of the CORBA Interoperable Naming Service (INS) to obtain an initial object reference for the Security Service. For information on the INS bootstrapping mechanism, see the *CORBA Programming Reference*.

The Security Sample Application

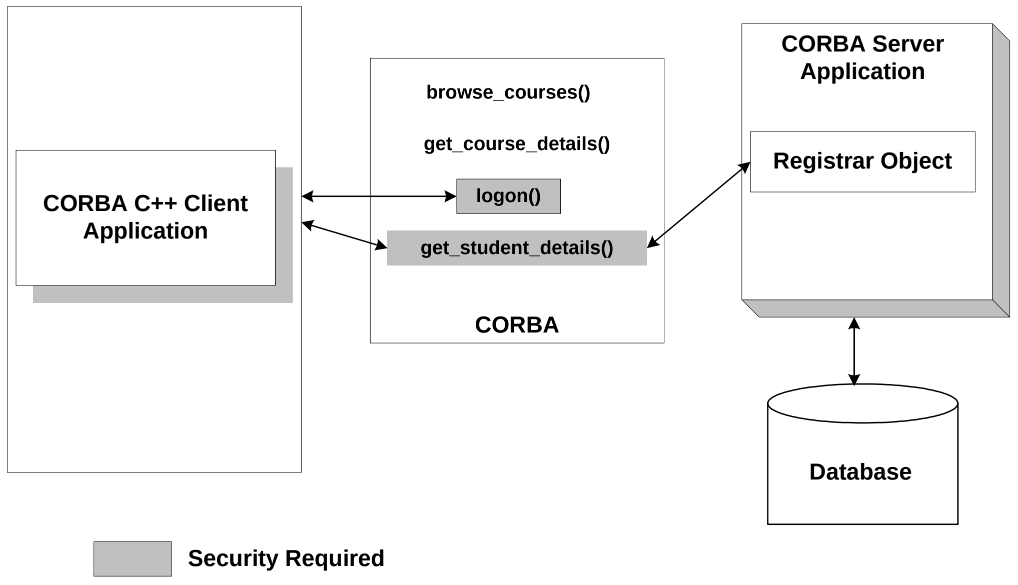
The Security sample application demonstrates how to use password authentication. The Security sample application requires that each student using the application has an ID and a password. The Security sample application works in the following manner:

- The client application has a `logon()` operation. This operation invokes operations on the PrincipalAuthenticator object, which is obtained as part of the process of logging on to access the domain.
- The server application implements a `get_student_details()` operation on the `Registrar` object to return information about a student. After the user is authenticated, logon is complete and the `get_student_details()` operation accesses the student information in the database to obtain the student information needed by the client logon operation.
- The database in the Security sample application contains course and student information.

Note: Certificate authentication is illustrated in the Secure Simpapp sample application.

Figure 4-2 illustrates the Security sample application.

Figure 4-2 Security Sample Application



The source files for the Security sample application are located in the `\samples\corba\university` directory in the BEA Tuxedo software. For information about building and running the Security sample application, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

Development Steps

Table 4-1 lists the development steps for writing a BEA Tuxedo CORBA application that employs authentication security.

Table 4-1 Development Steps for BEA Tuxedo CORBA Applications That Have Security

Step	Description
1	Define the security level in the configuration file.
2	Write the CORBA client application.

Step 1: Define the Security Level in the Configuration File

The security level for a BEA Tuxedo domain is defined by setting the `SECURITY` parameter in the `RESOURCES` section of the configuration file to the desired security level. Table 4-2 lists the options for the `SECURITY` parameter.

Table 4-2 Options for the SECURITY Parameter

Option	Definition
`NONE`	No security is implemented in the domain. This option is the default. This option maps to the `TOBJ_NOAUTH` level of authentication.
`APP_PW`	Requires that client applications provide an application password during initialization. The `tmloadcf` command prompts for an application password. This option maps to the `TOBJ_SYSAUTH` level of authentication.
`USER_AUTH`	Requires an application password and performs a per-user authentication during the initialization of the client application. This option maps to the `TOBJ_APPAUTH` level of authentication.

In the Security sample application, the `SECURITY` parameter is set to `APP_PW` for application-level security. For information about adding security to a BEA Tuxedo CORBA application, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

Step 2: Write the CORBA Client Application

Write client application code that does the following:

1. Uses the Bootstrap object to obtain a reference to the SecurityCurrent object for the specific BEA Tuxedo domain.
2. Gets the PrincipalAuthenticator object from the SecurityCurrent object.
3. Uses the `get_auth_type()` operation of the PrincipalAuthenticator object to return the type of authentication expected by the BEA Tuxedo domain.

Listing 4-1 include the portions of the CORBA C++ client applications in the Security sample application that illustrate the development steps for security.

Listing 4-1 Example of Security in a CORBA C++ Client Application

```
CORBA::Object_var var_security_current_oref =
     bootstrap.resolve_initial_references(“SecurityCurrent”);
SecurityLevel2::Current_var var_security_current_ref =
     SecurityLevel2::Current::_narrow(var_security_current_oref.in());


//Get the PrincipalAuthenticator
SecurityLevel2::PrincipalAuthenticator_var var_principal_authenticator_oref =
     var_security_current_ref->principal_authenticator();
//Narrow the PrincipalAuthenticator
Tobj::PrincipalAuthenticator_var var_bea_principal_authenticator =
     Tobj::PrincipalAuthenticator::_narrow (
                                 var_principal_authenticator_oref.in());

//Determine the security level
Tobj::AuthType auth_type = var_bea_principal_authenticator->get_auth_type();
Security::AuthenticationStatus status = var_bea_principalauthenticator->logon(
                                                   user_name,
                                                   client_name,
                                                   system_password,
                                                   user_password,
                                                   0);
```

CHAPTER 5

Using Transactions

This topic includes the following sections:

- Overview of the Transaction Service
- What Happens During a Transaction
- Transactions Sample Application
- Development Steps

Note: This topic describes using the C++ interface to the CORBA Services Object Transaction service. For a complete description of all the transaction features available in the CORBA environment of the BEA Tuxedo product and instructions for implementing the transaction features, see *Using CORBA Transactions* in the BEA Tuxedo online documentation.

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

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

Overview of the Transaction Service

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

The BEA Tuxedo system uses the following:

- The CORBA Services Object Transaction Service (OTS)

The CORBA environment in the BEA Tuxedo product provides a C++ interface to the Object Transaction Service. The OTS is accessed through the TransactionCurrent environmental object. For information about using the TransactionCurrent environmental object, see *Creating CORBA Client Applications* in the BEA Tuxedo online documentation.

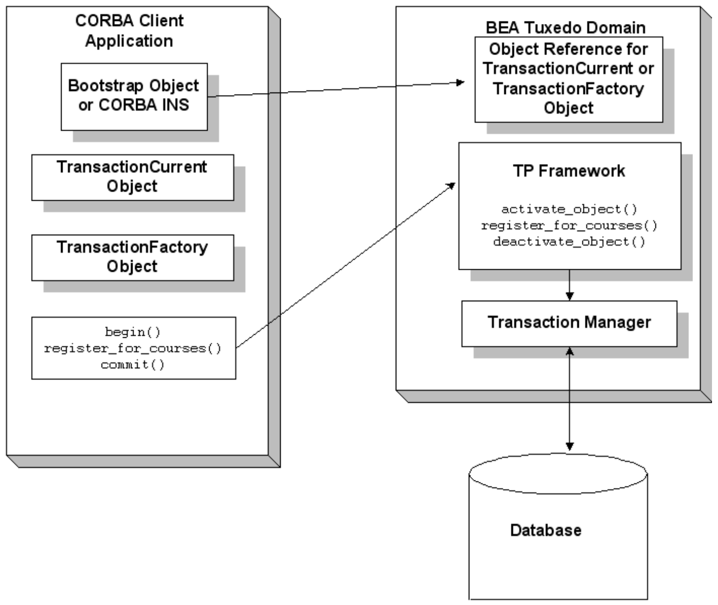
OTS provides the following support for your business transactions:

- Creates a global transaction identifier when a client application initiates a transaction.
- Works with the TP Framework to track objects that are involved in a transaction and, therefore, need to be coordinated when the transaction is ready to commit.
- Notifies the resource managers—which are, most often, databases—when they are accessed on behalf of a transaction. Resource managers then lock the accessed records until the end of the transaction.
- Orchestrates the two-phase commit when the transaction completes, which ensures that all the participants in the transaction commit their updates simultaneously. It coordinates the commit with any databases that are being updated using the Open Group XA protocol. Almost all relational databases support this standard.
- Executes the rollback procedure when the transaction must be stopped.
- Executes a recovery procedure when failures occur. It determines which transactions were active in the machine at the time of the crash, and then determines whether the transaction should be rolled back or committed.

What Happens During a Transaction

Figure 5-1 illustrates how transactions work in a BEA Tuxedo CORBA application.

Figure 5-1 How Transactions Work in a BEA Tuxedo CORBA Application



A basic transaction works in the following way:

1. The client application uses the Bootstrap object to return an object reference to the TransactionCurrent object for the BEA Tuxedo domain.

2. A client application begins a transaction using the `Tobj::TransactionCurrent::begin()` method, and issues a request to the CORBA interface through the TP Framework. All operations on the CORBA interface execute within the scope of a transaction.

   – If a call to any of these operations raises an exception (either explicitly or as a result of a communication failure), the exception can be caught and the transaction can be rolled back.

   – If no exceptions occur, the client application commits the current transaction using the `Tobj::TransactionCurrent::commit()` method. This method ends the transaction and starts the processing of the operation. The transaction is committed only if all of the participants in the transaction agree to commit.

3. The `Tobj::TransactionCurrent:commit()` method causes the TP Framework to call the Transaction Manager to complete the transaction.

4. The Transaction Manager updates the database.

Note: BEA Tuxedo CORBA also supports the use of the CORBA Interoperable Naming Service (INS) to obtain an initial object reference for the Security Service. For information on the INS bootstrapping mechanism, see the *CORBA Programming Reference*.

Transactions Sample Application

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

The Transactions sample application works in the following way:

1. Students submit a list of courses for which they want to be registered.

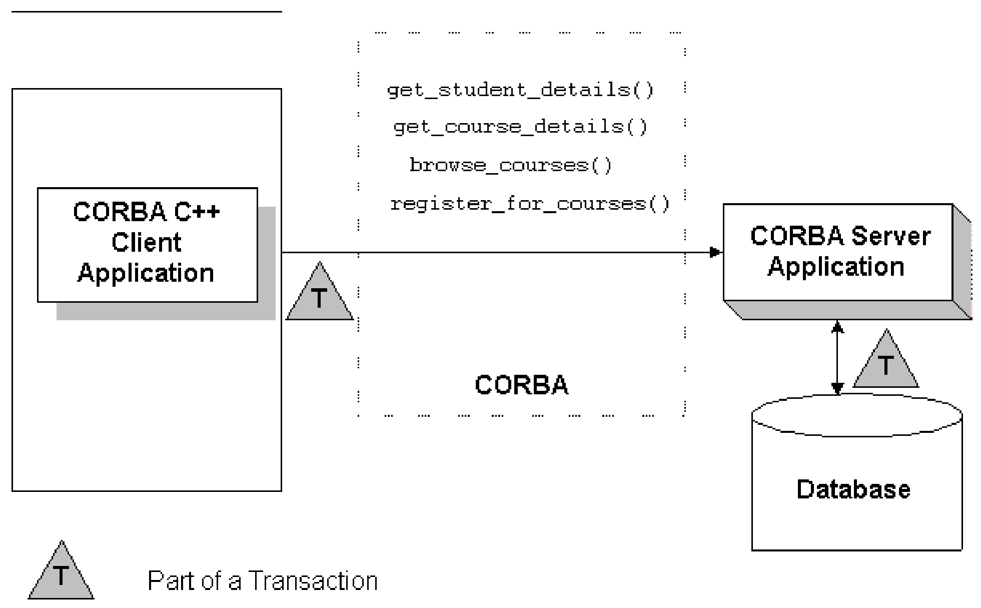
2. For each course in the list, the CORBA server application checks whether:
   - The course is in the database.
   - The student is already registered for a course.
   - The student exceeds the maximum number of credits the student can take.

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

that the request was rejected. The CORBA client application then rolls back the transaction.

Figure 5-2 illustrates how the Transactions sample application works.

Figure 5-2 Transactions Sample Application



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

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

Development Steps

This topic describes the development steps for writing a BEA Tuxedo CORBA application that includes transactions. Table 5-1 lists the development steps.

Table 5-1 Development Steps for BEA Tuxedo CORBA Applications That Have Transactions

Step	Description
1	Write the OMG IDL code for the transactional CORBA interface.
2	Define the transaction policies for the CORBA interface in the Implementation Configuration file (ICF).
3	Write the CORBA client application.
4	Write the CORBA server application.
5	Create a configuration file.

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

Step 1: Write the OMG IDL Code

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

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

Listing 5-1 includes the OMG IDL code for the Transactions sample application.

Listing 5-1 OMG IDL Code for the Transactions Sample Application

```
#pragma prefix "beasys.com"
module UniversityT
{
       typedef unsigned long CourseNumber;
       typedef sequence<CourseNumber> CourseNumberList;

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

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

       void destroy();
       };
       typedef unsigned short Days;
       const Days MONDAY    =  1;
       const Days TUESDAY   =  2;
       const Days WEDNESDAY =  4;
       const Days THURSDAY  =  8;
       const Days FRIDAY    = 16;
//Classes restricted to same time block on all scheduled days,
//starting on the hour
struct ClassSchedule
{
       Days           class_days; // bitmask of days
       unsigned short start_hour; // whole hours in military time
       unsigned short duration;   // minutes
};
struct CourseDetails
{
       CourseNumber   course_number;
       double         cost;
       unsigned short number_of_credits;
       ClassSchedule  class_schedule;
       unsigned short number_of_seats;
       string         title;
       string         professor;
       string         description;
};
       typedef sequence<CourseDetails> CourseDetailsList;
       typedef unsigned long StudentId;
struct StudentDetails
{
       StudentId         student_id;
       string            name;
       CourseDetailsList registered_courses;
};
enum NotRegisteredReason
{
       AlreadyRegistered,
       NoSuchCourse
};
struct NotRegistered
{
       CourseNumber        course_number;
       NotRegisteredReason not_registered_reason;
};
       typedef sequence<NotRegistered> NotRegisteredList;
exception TooManyCredits
{
       unsigned short maximum_credits;
};
//The Registrar interface is the main interface that allows
//students to access the database.
interface Registrar
{
       CourseSynopsisList
       get_courses_synopsis(
              in string                    search_criteria,
              in unsigned long             number_to_get,
              out unsigned long            number_remaining,
              out CourseSynopsisEnumerator rest);
       CourseDetailsList get_courses_details(in CourseNumberList
        courses);
       StudentDetails get_student_details(in StudentId student);
       NotRegisteredList register_for_courses(
              in StudentId        student,
              in CourseNumberList courses
       ) raises (
              TooManyCredits
       );
};
// The RegistrarFactory interface finds Registrar interfaces.
interface RegistrarFactory
{
       Registrar find_registrar(
       );
};
```

Step 2: Define Transaction Policies for the Interfaces

Transaction policies are used on a per-interface basis. During design, it is decided which interfaces within a BEA Tuxedo CORBA application will handle transactions. The transaction policies are listed in the following table.

Transaction Policy	Description
`always`	The interface must always be part of a transaction. If the interface is not part of a transaction, a transaction will be automatically started by the TP Framework.
`ignore`	The interface is not transactional; however, requests made to this interface within a scope of a transaction are allowed. The `AUTOTRAN` parameter, specified in the `UBBCONFIG` file for this interface, is ignored.
`never`	The interface is not transactional. Objects created for this interface can never be involved in a transaction. The BEA Tuxedo system generates the `INVALID_TRANSACTION` exception if an interface with this policy is involved in a transaction.
`optional`	The interface might be transactional. Objects can be involved in a transaction if the request is transactional. This transaction policy is the default. **Note:** To define transactional properties for a request you can also use the `autotran` parameter.

During development, you decide which interfaces will execute in a transaction by assigning transaction policies.

For CORBA server applications, you specify transaction policies in the Implementation Configuration File (ICF). A template ICF file is created when you run the `genicf` command.

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

Step 3: Write the CORBA Client Application

The CORBA client application needs code that performs the following tasks:

1. Obtains a reference to the TransactionCurrent or TransactionFactory object from the Bootstrap object.

2. Begins a transaction by invoking the `Tobj::TransactionCurrent::begin()` operation on the TransactionCurrent object.

3. Invokes operations on the object. In the Transactions sample application, the CORBA client application invokes the `register_for_courses()` operation on the `Registrar` object, passing a list of courses.

Listing 5-2 illustrates the portion of the CORBA C++ client application in the Transactions sample application that illustrates the development steps for transactions.

Listing 5-2 Transactions Code for CORBA C++ Client Applications

```
CORBA::Object_var var_transaction_current_oref =
     Bootstrap.resolve_initial_references("TransactionCurrent");
CosTransactions::Current_var var_transaction_current_ref=
     CosTransactions::Current::_narrow(var_transaction_current_oref.in());
//Begin the transaction
var_transaction_current_ref->begin();
try     {
                   // Perform the operation inside the transaction
    pointer_Registar_ref->register_for_courses(student_id, course_number_list);
     // ...
     // If operation executes with no errors, commit the transaction:
     CORBA::Boolean report_heuristics = CORBA_TRUE;
     var_transaction_current_ref->commit(report_heuristics);
}
catch (...) {
      // If the operation has problems executing, rollback the
      // transaction. Then throw the original exception again.
      // If the rollback fails, ignore the exception and throw the
      // original exception again.
      try {
           var_transaction_current_ref->rollback();
      }
      catch (...) {
            TP::userlog("rollback failed");
      }
      throw;
}
```

Step 4: Write the CORBA Server Application

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

If your BEA Tuxedo CORBA application uses a database, you need to include code in the CORBA server application that opens and closes an XA resource manager. These operations are included in the `Server::initialize()` and `Server::release()` operations of the Server object.

Listing 5-3 shows the portion of the code for the Server object in the Transactions sample application that opens and closes the XA resource manager.

Note: For a complete example of a C++ server application that implements transactions, see the Transactions sample application in *Using CORBA Transactions* in the BEA Tuxedo online documentation.

Listing 5-3 C++ Server Object in Transactions Sample Application

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

void Server::release()
{
       TRACE_METHOD("Server::release");
       cleanup();
}
static void cleanup()
{
       unregister_factory();
       end_transactional();
       close_database();
}

// Utilities to manage transaction resource manager
CORBA::Boolean s_became_transactional = CORBA_FALSE;
static void begin_transactional()
{
       TP::open_xa_rm();
       s_became_transactional = CORBA_TRUE;
}
static void end_transactional()
{
       if(!s_became_transactional){
            // cleanup not necessary
            return;
       }
       try {
            TP::close_xa_rm ();
       }
       catch (CORBA::Exception& e) {
              LOG("CORBA Exception : " << e);
       }
       catch (...) {
              LOG("unexpected exception");
       }
       s_became_transactional = CORBA_FALSE;
}
```

Step 5: Create a Configuration File

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

- In the `SERVERS` section specify the transactional group for the CORBA server application and for the application that manages the database.
- In the `GROUPS` section define the server group. In the `OPENINFO` and `CLOSEINFO` parameters of the `GROUPS` section, include information to open and close the XA resource manager for the database. You obtain this information from the product documentation for your database. Note that the default version of the `com.beasys.Tobj.Server.initialize()` operation automatically opens the resource manager.
- Include the pathname to the transaction log (`TLOG`) in the `TLOGDEVICE` parameter. For more information about the transaction log, see *Administering a BEA Tuxedo Application at Run Time* in the BEA Tuxedo online documentation.

Listing 5-4 includes the portions of the configuration file that define this information for the Transactions sample application.

Listing 5-4 Configuration File for Transactions Sample Application

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

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

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

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

*SERVERS
       DEFAULT:
       RESTART = Y
       MAXGEN  = 5


       TMSYSEVT
         SRVGRP  = SYS_GRP
         SRVID   = 1

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

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

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

       TMIFRSVR
         SRVGRP  = SYS_GRP
         SRVID   = 5

       UNIVT_SERVER
         SRVGRP  = ORA_GRP
         SRVID   = 1
         RESTART = N

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

*SERVICES
```

For information about the transaction log and defining parameters in the Configuration file, see *Setting Up a BEA Tuxedo Application* in the BEA Tuxedo online documentation.

Index

A

activation policies
- defining in Implementation Configuration file 3-11
- defining in Server Description file 3-11
- Simpapp sample application 3-11
- Simple interface 3-11
- supported 3-11

AdminAPI
- description 2-4

administration commands
- tmadmin command 2-3
- tmboot command 2-3
- tmconfig command 2-3
- tmloadcf command 2-3
- tmshutdown command 2-3
- tmunloadcf command 2-3

Administration console
- description 2-3

administration tools
- AdminAPI 2-4
- administration commands 2-3
- Administration console 2-3

Authenticates 2-15

authentication
- client application 2-15
- levels 4-2

B

BEA Tuxedo
- administration tools 2-3
- development commands 2-2
- how CORBA client and server applications work 2-13
- IDL compilers 2-2
- object services 2-5

BEA Tuxedo CORBA
- description of components 2-6
- features 1-3
- illustrated 2-7
- managing
  - tmconfig command 2-3
  - tmunloadcf command 2-3

BEA Tuxedo CORBA applications
- defining security levels 4-6
- how they work 2-13
- managing
  - tmadmin command 2-3
  - tmboot command 2-3
  - tmloadcf command 2-3
  - tmshutdown command 2-3
- using CORBAservices Object Transaction Service 5-2
- using Java Transaction Service 5-2

BEA Tuxedo CORBA components
- ORB 2-10
- TP Framework 2-11

BEA Tuxedo domain
- adding security to 4-4

Bootstrap object
- illustrated 2-8
- Simpapp sample application 3-14

building
- C++ client applications 3-20
  - buildobjclient command 2-2

C++ server applications
buildobjserver command 2-2
genicf command 2-2
Java client applications 3-20
buildobjclient command
building C++ client applications 2-2
description 2-2
format 3-21
in the Simpapp sample application 3-20
buildobjserver command
building C++ server applications 2-2
description 2-2
format 3-20
in the Simpapp sample application 3-20

C

client applications
authenticating into the BEA Tuxedo domain 2-15
initialization process 2-14
invoking objects 2-17
using transactions 5-3
writing
Security sample application 5-10
Simpapp sample application 3-14
Transactions sample application 5-10
client stubs
generating 3-6
in Simpapp sample application 3-6
code example
C++ client application for Simpapp sample application 3-15
C++ implementation of the Simple interface 3-8
C++ Server object 3-9
C++ server object that supports transactions 5-12
configuration file for Simpapp sample application 3-18
OMG IDL for Transactions sample application 5-7
security in C++ client applications 4-7
transactions in C++ client application 5-11
UBBCONFIG file for Transactions sample application 5-14
compiling
C++ client applications 3-20
C++ server applications 3-20
Java client applications 3-20
CORBAservices Object Transaction Service
using in BEA Tuxedo CORBA applications 5-2
create_servant method 2-15

D

development commands
buildobjclient command 2-2
buildobjserver command 2-2
genicf command 2-2
idl2ir command 2-2
ir2idl command 2-2
irdel command 2-2
development process
activation policies 3-11
BEA Tuxedo CORBA applications 3-2
client applications
Security sample application 4-6
Simpapp sample application 3-14
Transactions sample application 5-10
defining object activation policies 3-11
illustrated 3-3
Implementation Configuration file 3-11
OMG IDL
Simpapp sample application 3-4
Transactions sample application 5-6
Security sample application 4-5
server applications
Simpapp sample application 3-7
Transactions sample application 5-11

Server Description file 3-11
Simpapp sample application 3-3
steps for creating BEA Tuxedo CORBA applications 3-2
Transactions sample application 5-6
writing a configuration file 3-17
writing server application code 3-7
writing the client application code 3-14
writing the OMG IDL 3-4

E

environmental objects
and client initialization 2-14
description 2-5

F

factories
finding 2-15
registering 2-15
FactoryFinder object
description 2-5
example use of 2-15

G

genicf command
creating an ICF file 2-2
description 2-2

I

idl command 2-2
description 2-2
files created by 3-6
generating client stubs 3-6
generating skeletons 3-6
IDL compiler
idl command 2-2
supported 2-2
idl2ir command
description 2-2
Implementation Configuration file
defining activation policies 3-11
defining transaction policies 5-9
initialize method
summary 2-13
INS
See Interoperable Naming Service 2-7
Interface Definition Language 2-2
Interface Repository
creating 2-2
deleting objects from 2-2
idl2ir command 2-2
ir2idl command 2-2
irdel command 2-2
loading interface definitions into 2-2
InterfaceRepository object
description 2-5
interfaces
writing methods to implement operations 3-7
Interoperable Naming Service 2-7
ir2idl command
description 2-2
irdel command
description 2-2

J

Java Transaction Service
using in BEA Tuxedo CORBA applications 5-2

M

m3idltojava command
deprecated 2-2
Management Information Base
see MIB 1-3
managing
BEA Tuxedo CORBA applications
tmadmin command 2-3

        tmboot command 2-3
        tmconfig command 2-3
        tmloadcf command 2-3
        tmshutdown command 2-3
        tmunloadcf command 2-3
method implementations
    C++ 3-7
    writing 3-7
MIB
    for BEA Tuxedo CORBA applications 1-3

O

Object Life Cycle service
    description 2-5
object request broker
    see ORB 2-10
object services
    Interface Repository 2-5
    Object Life Cycle service 2-5
    Security service 2-5
    Transaction service 2-5
objects
    invoking 2-17
OMG IDL
    compiling 3-6
    generating client stubs 3-6
    generating skeletons 3-6
    Simple interface 3-5
    SimpleFactory interface 3-5
    Transactions sample application 5-6
ORB
    description 2-10
    illustrated 2-10

P

POA
    description 2-11
    interaction with TP Framework 2-11
Portable Object Adapter
    see POA 2-11
PrincipalAuthenticator object
    using in client applications 4-4
programming tools 2-2

R

register_factory method
    example of 2-15
resolve_initial_references method 2-14

S

Security sample application
    defining security level 4-6
    description 4-4
    development process 4-5
    illustrated 4-4
    location of files 4-5
    PrincipalAuthenticator object 4-4
    SecurityCurrent object 4-4
    using the PrincipalAuthenticator object 4-6
    using the SecurityCurrent object 4-6
    writing the client application 4-6
Security service
    description 2-5
    functional description 4-2
SecurityCurrent object
    description 2-5
    using in client applications 4-4
server applications
    defining object activation policies 3-11
    Implementation Configuration file 3-11
    Server Description file 3-11
    writing
        Simpapp sample application 3-7
        Transactions sample application 5-11
    writing method implementations 3-7
    writing the Server object 3-9
Server Description file
    defining activation policies 3-11
Server object 5-12
    description 2-12

Transactions sample application 5-12
writing 3-9
Simpapp sample application
compiling
C++ client application 3-20
C++ server application 3-20
Java client application 3-20
configuration file 3-17
description 3-3
file location 3-4
illustrated 3-4
interfaces defined for 3-5
OMG IDL 3-5
using the Bootstrap object 3-14
using the buildobjserver command 3-20
writing the client application code 3-14
Simple interface
activation policy 3-11
OMG IDL 3-5
SimpleFactory interface
OMG IDL 3-5
skeletons
generating 3-6
in Simpapp sample application 3-6
supporting databases 5-12

T

TLOGDEVICE parameter 5-14
tmadmin command
description 2-3
tmboot command
description 2-3
tmconfig command
description 2-3
tmloadcf command
creating a configuration file 3-20
description 2-3
tmshutdown command
description 2-3
tmunloadcf command
description 2-3
Tobj_Bootstrap 2-14
TP Framework
description 2-11
illustrated 2-12
transaction policies
defined 5-9
Transaction server application
writing the server application 5-11
Transaction service
description 2-5, 5-2
features 5-2
TransactionCurrent object
description 2-5
transactions
functional overview 5-2
illustrated 5-3
in client applications 5-3
Transactions sample application
description 5-4
file location 5-6
illustrated 5-5
OMG IDL 5-6
starting server application 5-11
transaction policies 5-10
UBBCONFIG file 5-14
writing client applications 5-10
writing server applications 5-11
Transactions sample application development process 5-6
TUXCONFIG file
description 3-20

U

UBBCONFIG file
adding transactions 5-14
description 3-20
sections in 3-17
setting the security level 4-6
user exceptions

Transactions sample application 5-4
UserTransaction object
description 2-5