BEA Tuxedo®

Using CORBA Server-to-Server Communication

Version 10.0
Document Released: September 28, 2007

Contents

3. Java Joint Client/Server Applications

CHAPTER 1

Understanding CORBA Server-to-Server Communication

This topic includes the following sections:

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

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

Overview of CORBA Server-to-Server Communication

CORBA server-to-server communication allows BEA Tuxedo applications to invoke CORBA objects and handle invocations from those CORBA objects (referred to as callback objects). The CORBA objects can be either inside or outside of a BEA Tuxedo domain.

The BEA Tuxedo product offers an implementation of the Internet Inter-ORB Protocol (IIOP) version 1.2, which provides inbound and outbound communication with the CORBA objects. Server-to-server communication provides more efficient use of network resources and provides

integration with third-party Object Request Brokers (ORBs). In addition, server-to-server communication is supported with CORBA objects that are implemented using IIOP versions 1.0 and 1.1.

Joint Client/Server Applications

Server-to-server communication allows client applications to act as server applications for requests from other client applications. In addition, server-to-server communication allows BEA Tuxedo server applications to invoke objects on other ORBs.

Note: In earlier versions of the BEA Tuxedo and WebLogic Enterprise products, client applications invoked operations defined in Object Management Group (OMG) Interface Definition Language (IDL) on a CORBA object. The server applications implemented the operations of the CORBA object. The CORBA objects in the server application used BEA Tuxedo TP Framework and environmental objects to implement state management, security, and transactions. These CORBA objects are referred to as BEA Tuxedo objects. Server applications could act as client applications for other server applications; however, client applications could not act as server applications for other client applications.

The server-to-server communication functionality is available through a callback object. A callback object has two purposes:

- It invokes operations on either BEA Tuxedo or CORBA objects.
- It implements the operations of a CORBA object.

Callback objects do not use the TP Framework and are not subject to BEA Tuxedo administration. You should use them only when transactional behavior, security, reliability, and scalability are not important.

Callback objects are implemented in joint client/server applications. A joint client/server application consists of the following:

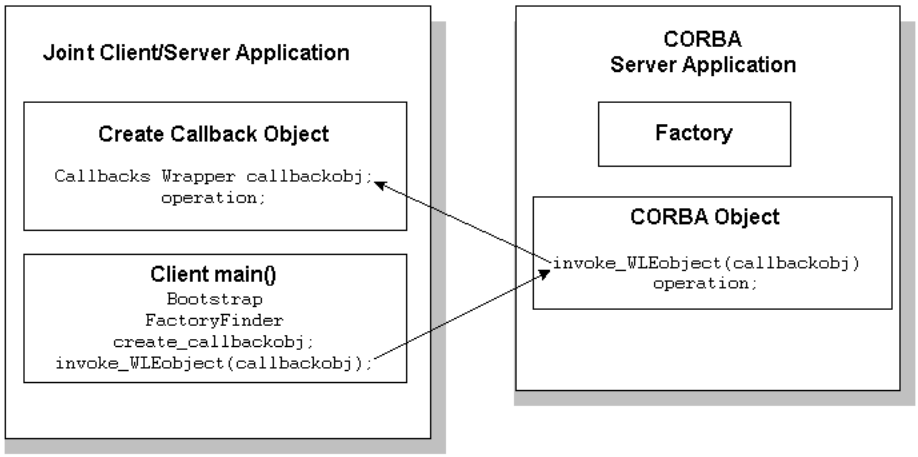
- Program logic that performs BEA Tuxedo client application functions, such as initializing the ORB, using the BEA Tuxedo environmental objects to establish connections, resolving initial references to the FactoryFinder object, and using factories to create BEA Tuxedo objects
- Program logic that creates a servant for a callback object and activates the callback object using an object ID

Note: Release 8.0 of the CORBA environment in the BEA Tuxedo CORBA product continues to include the BEA client environmental objects provided in earlier releases of BEA

WebLogic Enterprise for use with the BEA Tuxedo 8.0 CORBA clients. BEA Tuxedo 8.0 clients should use these environmental objects to resolve initial references to bootstrapping, security, and transaction objects. In this release, support has been added for using the OMG Interoperable Naming Service (INS) to resolve initial references to bootstrapping, security, and transaction objects. For information on INS, see Chapter 4, "CORBA Bootstrapping Programming Reference" in the *CORBA Programming Reference*.

Figure 1-1 shows the structure of a joint client/server application.

Figure 1-1 Structure of a Joint Client/Server Application



C++ joint client/server applications are supported.

Joint client/server applications use IIOP to communicate with the BEA Tuxedo server applications. IIOP can work in the following ways, depending on the version of the IIOP protocol you are using:

- Bidirectional

  Joint client/server applications are always connected to the same IIOP Server Handler (ISH) in the BEA Tuxedo domain. That ISH reuses the same connection to send requests to and receive requests from the joint client/server application.

- Dual-paired connection

Joint client/server applications use the register_callback_port method of the Bootstrap object to register the listening port of the joint client/server application in the ISH. Invocations from server applications on the callback object in the joint client/server application are routed through the ISH connected to the joint client/server application. This ISH uses a second outbound connection to send requests to and receive replies from the connected joint client/server application. The outbound connection is paired with the incoming connection. This differs from bidirectional IIOP, which uses only one connection.

- Asymmetric

  Joint client/server applications can invoke on any callback object, and are not restricted to invoking callback objects implemented in joint client/server applications connected to an ISH. Asymmetric IIOP forces the ORB infrastructure to search for an available ISH to handle the invocation.

Note: Depending on the type of remote object and the desired outbound IIOP configuration, you may have to perform additional programming tasks.

Table 1-1 lists the requirements for each type of object and outbound IIOP configuration.

Table 1-1 Programming Requirements for Using Outbound IIOP

Types of Objects	Asymmetric Requirements	Paired-connection Requirements	Bidirectional Requirements
Remote joint client/servers	Set `ISL CLOPT -O` option.	Use the `Tobj_Bootstrap::register_callback_port` method to register the callback port.	Use the `CORBA::ORB::create_policy` method to set `BiDirPolicy` on the POA.
Foreign (non BEA-Tuxedo) ORBs	Set `ISL CLOPT -O` option.	Not applicable.	If the foreign ORB supports the POA and `BiDirPolicy`, use the `CORBA::ORB::create_policy` method to set `BiDirPolicy` on the POA.
Remote clients	Remote clients are not servers, so outbound IIOP is not possible.		
Native joint client/servers	Outbound IIOP is not used.		
Native clients	Outbound IIOP is not used.		

For a more detailed description of bidirectional, dual-paired connnection, and asymmetric IIOP, see the *CORBA Programming Reference*.

Object Policies for Callback Objects

Callback objects are assigned policies that control how long an object reference is valid and how an object ID is assigned to the object. Object policies are defined when the reference to the callback object is created. In addition, they can be defined in the Callbacks Wrapper object, which simplifies the development of joint client/server applications.

The following object policies are supported for callback objects:

- Transient/System ID—the object reference for this type of callback object is valid only for the life of the joint client/server application. The object ID is assigned by the BEA Tuxedo system. This type of object is used for invocations that a joint client/server application wants to receive only until it terminates.
- Persistent/System ID—the object reference for this type of callback object is valid across multiple invocations in a joint client/server application. The object ID is assigned by the BEA Tuxedo system. This type of object is useful in joint client/server applications that stop and restart over a period of time. When the joint client/server application is running, it can receive requests on a particular callback object with that object reference. Typically, the joint client/server application creates the object reference once, saves it in its own permanent storage area, and reactivates the servant for the object every time the joint client/server application is started.
- Persistent/User ID—this object policy is the same as Persistent/System ID, except that the object ID is assigned by the joint client/server application.

When creating a callback object with an object policy of transient, the object reference is valid only until the joint client/server application is terminated or until the `stop_all_objects` method is called.

When creating a callback object with an object policy of persistent, the object reference is valid even after the termination of the joint client/server application. If the joint client/server application terminates, restarts, and activates a servant for the same object ID, the servant accepts requests made on that object reference.

Note: If you are creating a native joint client/server application (that is, a joint client/server application that is located in the same BEA Tuxedo domain as the server applications that invoke it), you cannot use the Persistent/System ID or Persistent/User ID object policies.

CHAPTER 2

Developing C++ Joint Client/Server Applications

This topic includes the following sections:

Development Process

Table 2-1 outlines the development process for C++ joint client/server applications.

Table 2-1 Development Process for C++ Joint Client/Server Applications

Step	Description
1	Write the OMG IDL for the callback interface and for the CORBA interfaces you want to use in your BEA Tuxedo application.
2	Generate the skeletons and client stubs.
3	Write the methods that implement the operations for each object.
4	Write the client portion of the joint client/server application.
5	Create a callback object using the Callbacks Wrapper object.
6	Invoke operations on a BEA Tuxedo object by passing the object reference for the callback object.
7	Specify configuration information.
8	Compile the joint client/server application.

These steps are explained in detail in subsequent topics.

Because the callback object in a joint client/server application is not transactional and has no object management capabilities, you do not need to create an Implementation Configuration File (*`filename`*`.icf`) for it. However, you still need to create an ICF file for the BEA Tuxedo objects in your BEA Tuxedo application. For information about writing an ICF file, see *Creating CORBA Server Applications*.

Chat Room Sample Application

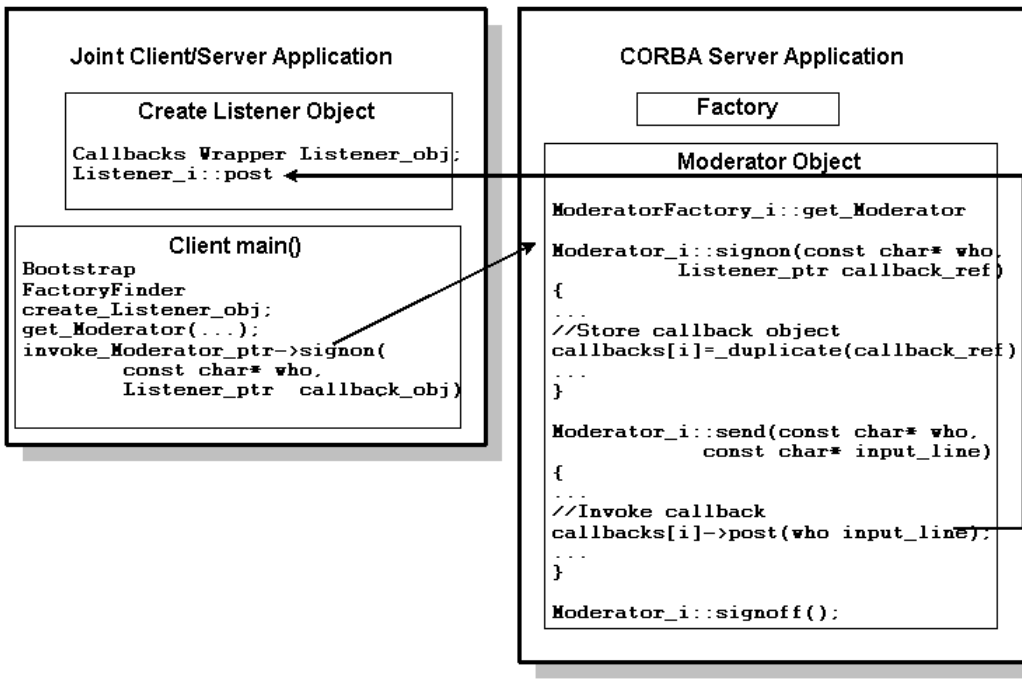
Throughout this topic, the Chat Room sample application is used to demonstrate the development steps. A chat room is an application that allows several people at different locations to communicate with each other. Think of the chat room as a moderator whose job it is to keep track of client applications that have logged in, and to distribute messages to those client applications.

A client application logs in to the moderator, supplying a username. When messages are entered at the keyboard, the client application invokes the moderator, and passes the messages to the moderator. The moderator then distributes the messages to all the other client applications by making an invocation on the callback object.

The Chat Room sample application consists of a C++ joint client/server application and a BEA Tuxedo server application. The joint client/server application receives keyboard input and makes invocations on the moderator. The joint client/server application also sets up the callback object to listen for messages from the moderator (that is, to receive invocations from the moderator). The BEA Tuxedo server application in the Chat Room sample application implements the moderator.

Figure 2-1 illustrates how the Chat Room sample application works.

Figure 2-1 How the Chat Room Sample Application Works



The Chat Room sample application works as follows:

1. The joint client/server application implements the logic for the callback object (the Listener object), creates a servant for the Listener object, and activates the Listener object.
2. The joint client/server application creates an object reference for the Listener object and passes it to the Moderator object as part of the `signon` operation.
3. The server application in the Chat Room sample application checks the keyboard for messages.
4. When messages are generated at the keyboard, the Chat Room sample application sends the messages to the Moderator object via the `send` operation.
5. The Chat Room sample application temporarily passes control over to the ORB to allow the Listener object in the joint client/server application to receive `post` invocations from the Moderator object.
6. The Listener object in the joint client/server application saves the posted messages until a client application requests them.

The source files for the Chat Room sample application are located in the *`TUXDIR`*`\samples\corba\chatroom` directory in the BEA Tuxedo software directory. See “Building and Running the Chat Room Sample Application” on page -23 for more information.

Step 1: Writing the OMG IDL

You use Object Management Group (OMG) Interface Definition Language (IDL) to describe available CORBA interfaces to client applications. 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. For more information about OMG IDL, see *Creating CORBA Client Applications.*

The Chat Room sample application implements the CORBA interfaces listed in Table 2-2.

Table 2-2 CORBA Interfaces for the Chat Room Sample Application

Interface	Description	Operation
`Listener`	The callback object	`post()`

Table 2-2 CORBA Interfaces for the Chat Room Sample Application

Interface	Description	Operation

`Moderator`	Receives input from client applications and uses the callback object to forward messages back to the joint client/server application	`signon()`
		`send()`
		`signoff()`

`ModeratorFactory`	Creates object references to the Moderator object	`get_moderator()`

Listing 2-1 shows the `chatclient.idl` that defines the `Listener` interface.

Listing 2-1 OMG IDL for the Listener Interface

```
module ChatClient{
       interface Listener  {
              oneway void post (in string from,
                                in string output_line);
       };
};
```

Listing 2-2 shows the `chatroom.idl` that defines the `Moderator` and `ModeratorFactory` interfaces for the Chat Room sample application. The `#include` is used to resolve references to interfaces in another OMG IDL file. In the Chat Room sample application, the `signon` method requires a Listener object as a parameter and its IDL file must `#include` the OMG IDL file that defines the Listener interface.

Listing 2-2 OMG IDL for the Moderator and ModeratorFactory Interfaces

```
#include "ChatClient.idl"

module ChatRoom {

       interface Moderator {
           exception IdAlreadyUsed{};
```

```
            exception NoRoomLeft{};
            exception IdNotKnown{};

            void signon( in string who,
                         in ChatClient::Listener callback_ref )
                             raises( IdAlreadyUsed, NoRoomLeft );

            void send  (in string who,
                        in string input_line )
                             raises( IdNotKnown );

            void signoff(in string who )
                            raises( IdNotKnown );
        };
        interface ModeratorFactory {
           Moderator get_moderator( in string chatroom_name );
        };
};
```

Step 2: Generating Skeletons and Client Stubs

The interface specification defined in OMG IDL is used by the IDL compiler to generate skeletons and client stubs. Note that a joint client/sever application uses the client stub for the BEA Tuxedo object and the skeleton and client stub for the callback object.

For example, in the Chat Room sample application, the joint client/server application uses the skeleton and client stub for the Listener object (that is, the callback object) to implement the object. The joint client/server application also uses the client stubs for the `Moderator` and `ModeratorFactory` interfaces to invoke operations on the objects. The BEA Tuxedo server application uses the skeletons for the Moderator and ModeratorFactory objects to implement the objects and the client stub for the Listener object to invoke operations on the object.

During the development process, use the `idl` command with the `-P` and `-i` options to compile the OMG IDL file that defines the callback object (for example, the `chatclient.idl` file in the Chat Room sample application). The options work as follows:

- The `-P` option creates a skeleton class that inherits directly from the `PortableServer::ServantBase` class. Inheriting from `PortableServer::ServantBase` means the joint client/server application must explicitly create a servant for the callback object and initialize the servant's state. The servant for the callback object cannot use the `activate_object` and `deactivate_object` methods as they are members of the `PortableServer::ServantBase` class.
- The `-i` option results in the generation of an implementation template file. This file is a template for the code that implements the interfaces defined in the OMG IDL for the Listener object.

You then need to compile the OMG IDL file that defines the interfaces in the BEA Tuxedo server application (for example, the `chatroom.idl` file in the Chat Room sample application). Use the `idl` command with only the `-i` option to compile that OMG IDL file.

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

Note: In the Chat Room sample application, the generated template files for the `ChatClient.idl` and `ChatRoom.idl` files have been renamed to reflect the objects (Listener and Moderator) they implement. In addition, the template file for the Moderator object includes the implementation for the ModeratorFactory object.

Table 2-3 Files Produced by the idl Command

File	Files in the Chat Room Sample Application Created by the `idl` Command	Description

Client stub file	`Listener_c.cpp`
	`Listener_c.h`
	`Moderator_c.cpp`
	`Moderator_c.h`
		Contains client stubs for each interface specified in the OMG IDL file. The client stubs are used to send a request to an object.

Table 2-3 Files Produced by the idl Command (Continued)

File	Files in the Chat Room Sample Application Created by the `idl` Command	Description
Implementation file	`Listener_i.cpp` `Moderator_i.cpp`	Contains signatures for the methods that implement the operations of the `Listener`, `Moderator`, and `ModeratorFactory` interfaces specified in the OMG IDL file. The `Listener_i.h` file contains implementation files that inherit from the `POA_ChatClient::Listener` class.

Skeleton file	`Listener_s.cpp`
	`Listener_s.h`
	`Moderator_s.cpp`
	`Moderator_s.h`
		Contains skeletons for each interface specified in the OMG IDL file. During run time, the skeleton maps client requests to the appropriate operation in the server application. The `Listener_s.h` file contains `POA_`*`skeleton`* class definitions (for example, `POA_ChatClient::Listener`).

Step 3: Writing the Methods That Implement the Operations for Each Object

After you compile each of the OMG IDL files, you need to write methods that implement the operations for each object. In a joint client/server application, you write the implementation file for the callback object (that is, the Listener object). You write the implementation for a callback object as you would write the implementation for any other CORBA object, except that you use the POA instead of the TP Framework. You also write implementation files for the BEA Tuxedo objects (that is, the Moderator and ModeratorFactory objects) in the BEA Tuxedo server application.

An implementation file contains the following:

- Method declarations for each operation specified in the OMG IDL file
- Business logic for your application
- Constructors for each interface implementation (implementing these is optional)

- Optionally, for BEA Tuxedo objects, the `com.beasys.Tobj_Servant.activate_object` and `com.beasys.Tobj_Servant.deactivate_object` methods

  Within the `activate_object` and `deactivate_object` methods, you write code that performs any particular steps related to activating or deactivating an object.

Listing 2-3 includes the implemention file for the Listener object, and Listing 2-4 includes the implementation file for the Moderator and ModeratorFactory objects.

Note: Additional methods and data were added to the implementation file for the Moderator and ModeratorFactory objects. The template for the implementation file was created by the `idl -i` command.

Listing 2-3 Implementation File for the Listener Object

```
// This module contains the definition of the implementation class
// Listener_i

#ifndef _Listener_i_h
#define _Listener_i_h

#include "ChatClient_s.h"

class Listener_i : public POA_ChatClient::Listener {
       public:

              Listener_i ();
              virtual ~Listener_i();

              void post (
                     const char * from,
                     const char * output_line);
...
};

#endif
```

Listing 2-4 Implementation File for Moderator and ModeratorFactory Objects

```
// This module contains the definitions of the implementation class
// Moderator and ModeratorFactory

#ifndef _Moderator_i_h
#define _Moderator_i_h

#include "ChatRoom_s.h"

const int CHATTER_LIMIT = 5;
// the most chatters allowed

class Moderator_i : public POA_ChatRoom::Moderator {
       public:

       //Define the operations
       void signon ( const char*      who,
                     ChatClient::Listener_ptr   callback_ref);
       void send ( const char *       who,
                   const char *       input_line);

       void signoff ( const char * who);

       //Define the Framework functions
       virtual void activate_object  ( const char* stroid );
       virtual void deactivate_object( const char* stroid,
                                  TobjS::DeactivateReasonValue
                                  reason);
       private:

       // Define function to find name on list
       int find( const char * handle );

       // Define name of the chat room overseen by the Moderator
       char* m_chatroom_name;

       // Data for maintaining list
```

```
        // Chatter[n] id
        CORBA::String            chatters[CHATTER_LIMIT];

        // Chatter[n] callback ref
        ChatClient::Listener_var callbacks[CHATTER_LIMIT];
};

class ModeratorFactory_i : public POA_ChatRoom::ModeratorFactory {
        public:
         ChatRoom::Moderator_ptr get_moderator ( const char*
                                                 chatroom_name );
};
#endif
```

Step 4: Writing the Client Portion of the Joint Client/Server Application

During development of a joint client/server application, you write the client portion of the joint client/server application as you would write any BEA Tuxedo client application. The client application needs to include code that does the following:

1. Initializes the ORB. The BEA Tuxedo system activates an ORB using the correct protocol (in this case, IIOP).
2. Uses the Bootstrap object to establish communication with the BEA Tuxedo domain.
3. Resolves initial references to the FactoryFinder object.
4. Uses a factory to get an object reference for the desired BEA Tuxedo object (that is, the Moderator object).

Note: Release 8.0 of the CORBA environment of the BEA Tuxedo product continues to include the BEA client environmental objects provided in earlier releases of BEA WebLogic Enterprise for use with the BEA Tuxedo 8.0 CORBA clients. BEA Tuxedo 8.0 clients should use these environmental objects to resolve initial references to bootstrapping, security, and transaction objects. In this release, support has been added for using the OMG Interoperable Naming Service (INS) to resolve initial references to bootstrapping, security, and transaction objects. For information on INS, see Chapter 4, “CORBA Bootstrapping Programming Reference” in the *CORBA Programming Reference*.

The client development steps are illustrated in Listing 2-5, which includes code from the Chat Room sample application. In the Chat Room sample application, the client portion of the joint client/server application uses a factory to get an object reference to the Moderator object, and then invokes the `signon`, `send`, and `signoff` methods on the Moderator object.

Listing 2-5 Client Portion of the Chat Room Joint Client/Server Application

```
...
// Initialize the ORB

orb_ptr = CORBA::ORB_init(argc, argv, "BEA_IIOP");

// Create a Bootstrap object to establish communication with the
// domain

bootstrap = new Tobj_Bootstrap(orb_ptr,"");

// Get a FactoryFinder object, use it to find a Moderator factory,
// and get a Moderator.

// Use the Bootstrap object to find the FactoryFinder object

CORBA::Object_var var_factory_finder_oref =
          bootstrap->resolve_initial_references("FactoryFinder");

// Narrow the FactoryFinder object

Tobj::FactoryFinder_var var_factory_finder =
       Tobj::FactoryFinder::_narrow(var_factory_finder_oref.in());

// Use the FactoryFinder object to find a factory for the Moderator

CORBA::Object_var var_moderator_factory_oref =
       var_factory_finder->find_one_factory_by_id(
       "ModeratorFactory" );

// Narrow the Moderator Factory
```

```
ChatRoom::ModeratorFactory_var var_moderator_factory =
       ChatRoom::ModeratorFactory::_narrow(
       var_moderator_factory_oref.in() );

// Get a Moderator
// The chatroom name is passed as a command line parameter

var_moderator_oref =
            var_moderator_factory->get_moderator
            (var_chat_room_name.in() );
...
```

Step 5: Creating a Callback Object Using the Callbacks Wrapper Object

Since the basic steps for creating a callback object are always the same, the BEA Tuxedo product provides a Callbacks Wrapper object that simplifies the development of callback objects.

The Callbacks Wrapper object does the following:

- Defines the object policy for the callback object. The following object policies are supported:
  - Transient/System ID (`_transient`)
  - Persistent/System ID (`_persistent/systemid`)
  - Persistent/User ID (`_persistent/userid`)

  For a complete description of the object policies for callback objects, see “Object Policies for Callback Objects” on page -5.
- Creates a servant for the callback object.
- Sets the ORB and the POA to the state in which they will accept requests on the callback object.
- Returns an object reference to the activated callback object. The object ID can be generated by the system or supplied by the user.

- Tells the ORB to stop accepting requests on either a single servant or all the active servants.

For a complete description of the Callbacks Wrapper object and its methods, see the *CORBA Programming Reference*.

Listing 2-6 shows how a Callbacks Wrapper object is used in the Chat Room sample application.

Listing 2-6 Using the Callbacks Wrapper Object in the Chat Room Sample Application

```
...
// Use the Callbacks object to create a servant for the
// Listener object, activate the Listener object, and create an
// object reference for the Listener object.
BEAWrapper::Callbacks* callbacks =
                       new BEAWrapper::Callbacks( orb_ptr );
Listener_i * listener_callback_servant = new Listener_i();
CORBA::Object_var v_listener_oref=callbacks->start_transient(
                                listener_callback_servant,
                                ChatClient::_tc_Listener->id());
ChatClient::Listener_var v_listener_callback_oref =
                                ChatClient::Listener::_narrow(
                                var_listener_oref.in());
...
```

Step 6: Invoking Operations on an Object by Passing a Reference to the Callback Object

Once you have an object reference to a callback object, you can pass the callback object reference as a parameter to a method of a BEA Tuxedo object. In the Chat Room sample application, the Moderator object uses an object reference to the Listener object as a parameter to the `signon` method. Listing 2-7 illustrates this step.

Listing 2-7 Invoking the signon Method

```
// Sign on to the Chat room using a user-defined handle and a
// reference to the Listener object (the callback object) to receive
// input from other client applications logged into the Chat room.

var_moderator_reference->signon(handle,
                                var_listener_callback_oref.in() );
```

Step 7: Specifying Configuration Information

When running remote joint client/server applications that use IIOP, the object references for the callback object must contain a host and port number, as follows.

- For transient callback objects, any valid port number (as defined by TCP/IP) can be used, and it can be obtained dynamically by the ORB.
- For persistent callback objects, the ORB must be configured to accept requests for the callback object on the same port on which the object reference for the callback object was created.

The user specifies the port number from the user range of port numbers, rather than from the dynamic range. Assigning port numbers from the user range prevents joint client/server applications from using conflicting ports. To specify a particular port for the joint client/server application to use, include the following on the command line that starts the process for the joint client/server application:

`-ORBport` *nnn*

where *nnn* is the number of the port to be used by the ORB when creating invocations and listening for invocations on the callback object in the joint client/server application.

Use this command when you want the object reference for the callback object in a joint client/server application to be persistent and when you want to stop and restart the joint client/server application. If this command is not used, the ORB uses a random port. If the joint client/server application is stopped and then started, invocations to callback objects in the the joint client/server application will fail.

The port number is part of the input to the `argv` argument of the `CORBA::orb_init` member function. When the `argv` argument is passed, the ORB reads that information, establishing the

port for any object references created in that process. You can also use the bootstrap object's `register_callback_port` operation for the same purpose.

For a joint client/server application to communicate with a BEA Tuxedo object in the same domain, a configuration file for the server application is needed. The configuration file should be written as part of the development of the server application. The binary version of the configuration file, the `TUXCONFIG` file, must exist before the joint client/server application is started. The `TUXCONFIG` file is created using the `tmloadcf` command. For information about creating a `TUXCONFIG` file, see *Getting Started with BEA Tuxedo CORBA Applications* and *Setting Up a BEA Tuxedo Application*.

If you are using a joint client/server application that uses IIOP version 1.0 or 1.1, the administrator needs to boot the IIOP Server Listener (ISL) with startup parameters that enable outbound IIOP to invoke callback objects not connected to an IIOP Server Handler (ISH). The `-O` option of the ISL command enables outbound IIOP. Additional parameters allow administrators to obtain the optimum configuration for their BEA Tuxedo application. For more information about the ISL command, see the *BEA Tuxedo Command Reference*.

Step 8: Compiling Joint Client/Server Applications

The final step in the development of a joint client/server application is to produce the executable program. To do this, you need to compile the code and link against the skeleton and client stub.

Use the `buildobjclient` command with the `-P` option to construct a joint client/server application executable. To build an executable program, the command combines the client stub for the BEA Tuxedo object, the client stub for the callback object, the skeleton for the callback object, and the implementation for the callback object with the appropriate POA libraries.

Note: Before you can use the `-P` option of the `buildobjclient` command, you must have used the `-P` option of the `idl` command when you created the skeleton and client stub for the callback object.

Using the POA to Create a Callback Object

You can use the POA directly to create a callback object. You would use the POA directly when you want to use POA features and object policies not available through the Callbacks Wrapper object. For example, if you want to use the POA optimization features, you need to use the POA directly. The following topics describe how to use the POA to create callback objects with the supported object policies.

Note: Only a subset of the POA interfaces are supported in this version of the BEA Tuxedo product. For a list of supported interfaces, see the *CORBA Programming Reference*.

Creating a Callback Object with a Transient Object Policy

To use the POA to create a callback object with a transient object policy, you need to write code that does the following:

1. Establishes a connection with a POA.
2. Creates a child POA.

   Since the root POA does not allow use of bidirectional IIOP, you need to create a child POA. The child POA can use the defaults for `LifespanPolicy (TRANSIENT)` and `IDAssignmentPolicy (SYSTEM)`. You must specify a `BiDirPolicy` policy of `BOTH`.

   IIOP version 1.2 supports reuse of the TCP/IP connection for both incoming and outgoing requests. Allowing reuse of a TCP/IP connection is the choice of the ORB. To allow reuse, you create an ORB policy object that allows reuse of a TCP/IP connection, and you use that policy object in the list of policies when initializing an ORB. The policy object is created using the `CORBA::ORB::create_policy` operation. For more information about the `CORBA::ORB::create_policy` operation, see the *CORBA Programming Reference*.
3. Creates a servant for the callback object.
4. Informs the POA that the servant is ready to accept requests for the callback object.

   In this step, the joint client/server application activates the callback object in the POA using an object ID.
5. Activates the POA.
6. Creates an object reference for the callback object.
7. Makes an invocation on a BEA Tuxedo object using the object reference for the callback object as a parameter to one of the methods of the BEA Tuxedo object.

Listing 2-8 shows the portion of the Chat Room sample application that uses the POA to create the Listener object.

Listing 2-8 Using the POA to Create the Listener Object

```
// Establish communication with the POA
```

```
orb_ptr = CORBA::ORB_init(argc, argv, "BEA_IIOP");
CORBA::PolicyList policy_list(1);
CORBA::Any val;

CORBA::Object_ptr o_init_poa;
o_init_poa = orb_ptr->resolve_initial_references("RootPOA");
// Narrow to get the Root POA

root_poa_ptr = PortableServer::POA::_narrow(o_init_poa);
CORBA::release(o_init_poa);


// Specify an IIOP Policy of Bidirectional for the POA

val <<= BiDirPolicy::BOTH;
CORBA::Policy_ptr bidir_pol_ptr = orb_ptr->create_policy(
                      BiDirPolicy::BIDIRECTIONAL_POLICY_TYPE, val);
policy_list.length ( 1 );
policy_list[0] = bidir_pol_ptr;

// Create the BiDirectional POA

bidir_poa_ptr = root_poa_ptr->create_POA("BiDirPOA",
                                         root_poa_ptr->
                                         the_POAManager(),
                                         policy_list);
// Activate the POA

root_poa_ptr->the_POAManager()->activate();

// Create the Listener object

ChatClient::Listener_var v_listener_callback_ref;

// Create a servant for Listener object and activate it

listener_callback_servant = new Listener_i();
  CORBA::Object_var           v_listener_oref;
```

```
  PortableServer::ObjectId_var temp_OId =
     bidir_poa_ptr ->activate_object(listener_callback_servant );

// Create object reference for the Listener object with a
// system generated Object Id

v_listener_oref = bidir_poa_ptr->create_reference_with_id
                                  (temp_OId,
                                  ChatClient::_tc_Listener->id() );
v_listener_callback_ref = ChatClient::Listener::_narrow
                                  ( v_listener_oref.in() );
```

Creating a Callback Object with a Persistent/User ID Object Policy

To use the POA to create a callback object with a Persistent/User ID object policy, you must write code that does the following:

1. Uses a string to store the user ID and converts the string to the object ID.
2. Creates a child POA with a `LifespanPolicy` set to `PERSISTENT` and `IDAssignmentPolicy` set to `USERID`.
3. Creates a servant for the Listener object.
4. Creates an object reference for the Listener object using the stringified object ID and the repository ID of the Listener object.
5. Activates the Listener object.

Note: The Persistent/User ID object policy is only used with remote joint client/server applications (that is, a joint client/server application that is not in a BEA Tuxedo domain).

Listing 2-9 shows code that performs these steps.

Note: The code example does not use bidirectional IIOP.

Listing 2-9 Example Code for Listener Object with Persistent/User ID Object Policy

```
// Declare a string and convert it to an object Id.
const char* oid_string = "783";
PortableServer::ObjectID_var oid=
             PortableServer::string_to_ObjectId(oid_string);

// Find the root POA
CORBA::Object_var oref =
             orb_ptr->resolve_initial_references("RootPOA");
PortableServer::POA_var root_poa =
             PortableServer::POA::_narrow(oref);

// Create and activate a Persistent/UserID POA
CORBA::PolicyList policies(2);
policies.length(2);
policies[0] = root_poa->create_lifespan_policy(
             PortableServer::PERSISTENT);
policies[1] = root_poa->create_id_assignment_policy(
             PortableServer::USER_ID );
PortableServer::POA_var poa_ref =
             root_poa->create_POA("poa_ref",
             root_poa->the_POAManager(),policies);
root_poa->the_POAManager()->activate();

// Create object reference for the Listener object.
oref = poa_ref->create_reference_with_id(oid,
             ChatClient::_tc_Listener->id());
ChatClient::Listener_ptr Listener_oref =
             ChatClient::Listener::_narrow( oref );

// Create Listener_i servant and activate the Listener object
Listener_i* my_Listener_i = new Listener_i();
poa_ref->activate_object_with_id( oid, my_Listener_i);

// Make call passing the reference to the Listener object
v_moderator_ref->signon( handle, Listener_oref);
```

Creating a Callback Object with a Persistent/System ID Object Policy

To use the POA to create a callback object with a Persistent/System ID object policy, you need to write code that does the following:

1. Creates a child POA with a `LifespanPolicy` set to `PERSISTENT` and `IDAssignmentPolicy` set to the default.
2. Creates a servant for the Listener object.
3. Creates an object reference for the Listener object using a system generated object ID (the repository ID of the Listener object).
4. Activates the Listener object.

Note: The Persistent/System ID object policy is only used with remote joint client/server applications (that is, a joint client/server application that is not in a BEA Tuxedo domain).

Listing 2-10 shows code that performs these steps.

Listing 2-10 Example Code for Listener Object with Persistent/System ID Object Policy

```
// Find the root POA
CORBA::Object_var oref=
       orb_ptr->resolve_initial_references("RootPOA")
PortableServer::POA_var root_poa =
       PortableServer::POA::_narrow(oref);

// Create and activate a Persistent/System ID POA
CORBA::PolicyList policies(1);
policies.length(1);
policies[0] = root_poa->create_lifespan_policy(
       PortableServer::PERSISTENT);

//IDAssignmentPolicy is the default so you do not need to specify it
PortableServer::POA_var poa_ref = root_poa->create_POA(
        "poa_ref", root_poa->the_POAManager(), policies);
root_poa->the_POAManager()->activate();
```

```
// Create Listener_i servant and activate the Listener object
Listener_i* my_Listener_i = new Listener_i();
PortableServer::ObjectId_var temp_OId =
       poa_ref->activate_object ( my_Listener_i );

// Create object reference for Listener object with returned
// system object Id
oref = poa_ref->create_reference_with_id(
       temp_OId, ChatClient::_tc_Listener->id() );
ChatClient::Listener_var Listener_oref =
       ChatClient::Listener::_narrow(oref);

// Make the call passing the reference to the Listener object
v_moderator_ref->signon( handle, Listener_oref );
```

Threading Considerations for C++ Joint Client/Server Applications

A joint client/server application can first function as a client application and then switch to functioning as a server application. To do this, the joint client/server application gives complete control of the thread to the ORB by invoking the following:

```
orb -> run();
```

If a method in the server portion of a joint client/server application invokes `ORB::shutdown()`, all server activity stops and control is returned to the statement after `ORB::run()` is invoked in the server portion of the joint client/server application. Only under this condition does control return to the client functionality of the joint client/server application.

Since a client application has only a single thread, the client functionality of the joint client/server application must share the central processing unit (CPU) with the server functionality of the joint client/server application. This sharing is accomplished by occasionally checking with the ORB to see if the joint client/server application has server application work to perform. Use the following code to perform the check with the ORB:

```
if ( orb->work_pending() ) orb->perform_work();
```

After the ORB completes the server application work, the ORB returns to the joint client/server application, which then performs client application functions. The joint client/server application must remember to occasionally check with the ORB; otherwise, the joint client/server application will never process any invocations.

The ORB cannot service callbacks while the joint client/server application is blocking on a request. If a joint client/server application invokes an object in another BEA Tuxedo server application, the ORB blocks while it waits for the response. While the ORB is blocking, it cannot service any callbacks, so the callbacks are queued until the request is completed.

Building and Running the Chat Room Sample Application

Perform the following steps to build and run the Chat Room sample application:

1. Copy the files for the Chat Room sample application into a work directory.
2. Change the protection attribute on the files for the Chat Room sample application.
3. Verify the settings of the environment variables.
4. Execute the `ChatSetup` command.

The following sections describe these steps.

Copying the Files for the Chat Room Sample Application into a Work Directory

You need to copy the files for the Chat Room sample application into a work directory on your local machine. The files for the Chat Room sample application are located in the following directories:

Windows

drive`:\`*TUXDIR*`\samples\corba\chatroom`

UNIX

`/usr/local/`*TUXDIR*`/samples/corba/chatroom`

Use the files listed in Table 2-4 to build and run the Chat Room sample application.

Table 2-4 Files Included in the Chat Room Sample Application

File	Description
`ChatRoom.idl`	The OMG IDL code that declares the `Moderator` and `ModeratorFactory` interfaces.
`ChatClient.idl`	The OMG IDL code that declares the `Listener` interface.
`Listener_i.h` `Listener_i.cpp`	The C++ source code for method implementations of the Listener object in the joint client/server application.
`Moderator_i.h` `Moderator_i.cpp`	The C++ source code for method implementations of the Moderator and ModeratorFactory objects in the BEA Tuxedo server application.
`ChatClientMain.cpp`	The C++ source code for the joint client/server application.
`ChatRoomServer.cpp`	The C++ source code for the BEA Tuxedo server application.
`KeyboardManager.h` `KeyboardManager.cpp`	The C++ source code that handles input from the keyboard in the Chat Room sample application. This code is used by `ChatClientMain.cpp`.
`ChatRoom.icf`	The Implementation Configuration File (ICF) for the Moderator and ModeratorFactory objects in the BEA Tuxedo server application in the Chat Room sample application.
`ChatRoom.ksh`	For UNIX systems, a script that sets the environment variables and builds the Chat Room sample application.
`ChatRoom.cmd`	For Windows systems, a command procedure that sets the environment variables and builds the Chat Room sample application.
`ChatRoom.mk`	The UNIX operating system `makefile` for the Chat Room sample application.

Table 2-4 Files Included in the Chat Room Sample Application (Continued)

File	Description
`ChatRoom.nt`	The Windows operating system `makefile` for the Chat Room sample application.
`Readme.txt`	The file that provides the latest information about building and running the Chat Room sample application.

Changing the Protection Attribute on the Files for the Chat Room Sample Application

During the installation of the BEA Tuxedo software, the sample application files are marked read-only. Before you can edit or build the files in the Chat Room sample application, you need to change the protection attribute of the files you copied into your work directory, as follows:

Windows

```
prompt> attrib /S -r drive:\workdirectory\*.*
```

UNIX

```
prompt> /bin/ksh
```

```
ksh prompt> chmod u+w /workdirectory/*.*
```

On UNIX operating system platforms, you also need to change the permission of `ChatRoom.ksh` to give execute permission to the file, as follows:

```
ksh prompt> chmod +x ChatRoom.ksh
```

Verifying the Setting of the TUXDIR Environment Variable

Before building and running the Chat Room sample application, you need to ensure that the `TUXDIR` environment variable is set on your system. In most cases, this environment variable is set as part of the installation procedure. The `TUXDIR` environment variable defines the directory path where you installed the BEA Tuxedo software. For example:

Windows

```
TUXDIR=C:\TUXDIR
```

UNIX

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

To verify that the information for the environment variables defined during installation is correct, perform the following steps:

Windows

1. From the Start menu, select Settings.
2. From the Settings menu, select the Control Panel.

   The Control Panel appears.
3. Click the System icon.

   The System Properties window appears.
4. Click the Environment tab.

   The Environment page appears.
5. Check the setting for `TUXDIR`.

UNIX

```
ksh prompt>printenv TUXDIR
```

To change the settings, perform the following steps:

Windows

1. On the Environment page in the System Properties window, click the `TUXDIR` environment variable.
2. Enter the correct information for the environment variable in the Value field.
3. Click OK to save the changes.

UNIX

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

Executing the ChatSetup Command

The `ChatSetup` command automates the following steps:

1. Sets the system environment variables.
2. Creates and loads the configuration file.

3. Compiles the code for the client application.
4. Compiles the code for the server application.

Before running the `ChatSetup` command, you need to check the following:

- Ensure that you have the appropriate administrative privileges to build and run applications.
- On Windows, make sure `nmake` is in the path of your machine.
- On UNIX, make sure the `make` executable program is included in the `PATH` variable.

To build and run the sample application, enter the `ChatSetup` command, as follows:

Windows

```
prompt>cd workdirectory
prompt> ChatSetup.cmd
```

UNIX

```
ksh prompt> cd workdirectory
ksh prompt> ./ChatSetup.ksh
```

Starting the Server Application

Start the server application and the system server processes in the Chat Room sample application by entering the following command:

```
prompt> tmboot -y
```

This command starts the following server processes:

- `TMSYSEVT`

  The system EventBroker. This server process is used only by the BEA Tuxedo system.

- `TMFFNAME`

  The following three `TMFFNAME` server processes are started:

  – The `TMFFNAME` server process started with the `-N` and `-M` options is the Master NameManager service. The NameManager service maintains a mapping of the application-supplied names to object references. This server process is used only by the BEA Tuxedo system.

  - The `TMFFNAME` server process started with only the `-N` option is the Slave NameManager service.
  - The `TMFFNAME` server process started with the `-F` option contains the FactoryFinder object.

- `ChatRoom`

  The server application process for the Chat Room sample application.

- `ISL`

  The IIOP Listener/Handler process.

Starting the Client Application

Start the client application in the Chat Room sample application by entering the following command:

```
prompt> ChatClient chatroom_name -ORBport nnn
```

where *`chatroom_name`* is the name of a chat room to which you want to connect. You can enter any value. You will be prompted for a handle to identify yourself. You can enter any value. If the handle you chose is in use, you will be prompted for another handle.

To optimize the usefulness of the Chat Room sample application, you should run a second client application using the same chat room name.

To exit the client application, enter `\`.

Stopping the Chat Room Sample Application

Before using another sample application, enter the following commands to stop the Chat Room sample application and to remove unnecessary files from the work directory:

Windows

```
prompt> tmshutdown -y

prompt> Admin\setenv

prompt> nmake -f ChatRoom.nt superclean

prompt> nmake -f ChatRoom.nt adminclean
```

UNIX

```
ksh prompt> tmshutdown -y
```

```
ksh prompt> . ./Admin/setenv.ksh
ksh prompt> make -f ChatRoom.mk superclean
ksh prompt> make -f ChatRoom.nt adminclean
```

CHAPTER 3

Java Joint Client/Server Applications

This topic includes the following sections:

- Development Process
- Support for Joint Client/Server Applications

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

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

Development Process

Table 3-1 outlines the development process for Java joint client/server applications.

Table 3-1 Development Process for Java Joint Client/Server Applications

Step	Description
1	Write the OMG IDL for the callback interface and the CORBA interfaces you want to use in your BEA Tuxedo application.
2	Generate the skeletons and client stubs.
3	Write the methods that implement the operations for each interface.
4	Initialize the ORB.
5	Write the client main portion of the joint client/server application.
6	Create a callback object using the Callbacks Wrapper object.
7	Establish communication with an ISH.
8	Invoke operations on the BEA Tuxedo object by passing an object reference for the callback object.
9	Specify configuration information.
10	Compile the joint client/server application.

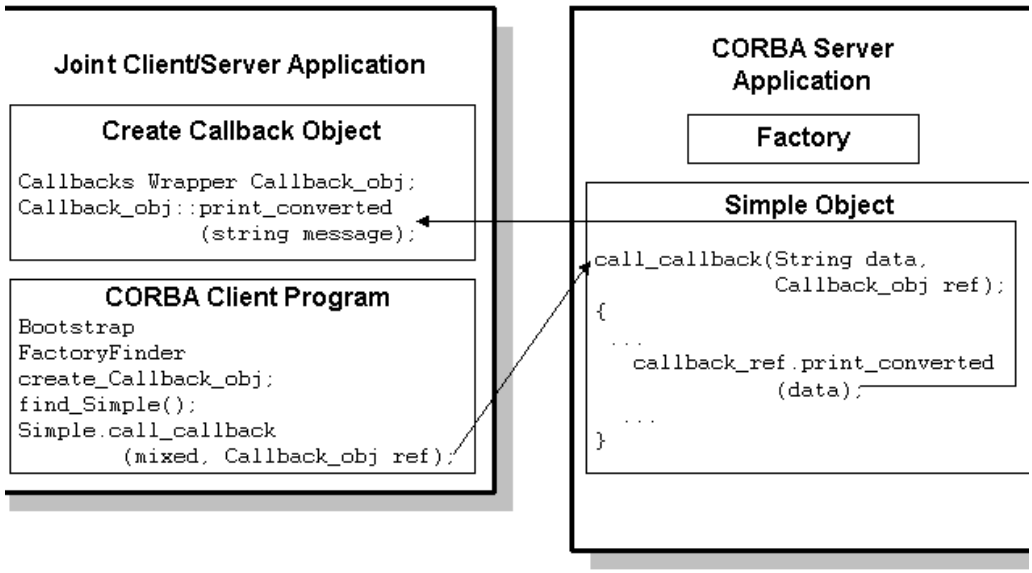
Because the callback object in a joint client/server application is not transactional and has no object management capabilities, you do not need to create a Server Description File (*filename*.xml) for it. However, you still need to create a Server Description File for the BEA Tuxedo objects in your BEA Tuxedo application.

Support for Joint Client/Server Applications

Notes: Release 8.0 of the CORBA environment of the BEA Tuxedo product does not support Java servers. Support for Java servers was included in versions 5.0 and 5.1 of the BEA WebLogic Enterprise product. That support was removed when BEA WebLogic Enterprise was merged with BEA Tuxedo in release 8.0.

An implementation of a joint client/server employs a callback object. Figure 3-1 illustrates the concept of a joint client/server application using a callback object.

Figure 3-1 The Concept of a Joint Client/Server Application



For a complete example of a joint client/server application, see Chapter 6, “Building the Advanced Sample Application,” in *Using the CORBA Notification Service*. The subscriber component in the Advanced sample application implements a joint client/server application.