BEA Tuxedo®

Using the CORBA Name Service

Version 10.0
Document Released: September 28, 2007

Contents

1. Overview of the CORBA Name Service

The CORBA Name Service 1-1

Understanding the CORBA Name Service 1-3

2. CORBA Name Service Reference

CORBA Name Service Commands 2-2

cns 2-3

cnsbind 2-6

cnsls 2-9

cnsunbind 2-11

Capabilities and Limitations of the CORBA Name Service 2-12

Getting the Initial Reference to the NameService Environmental Object 2-13

The CosNaming Data Structures Used by the CORBA Name Service 2-13

The NamingContext Object 2-14

CosNaming::NamingContext::bind() 2-15

CosNaming::NamingContext::bind_context() 2-16

CosNaming::NamingContext::bind_new_context() 2-17

CosNaming::NamingContext::destroy() 2-18

CosNaming::NamingContext::list() 2-19

CosNaming::NamingContext::new_context() 2-20

CosNaming::NamingContext::rebind() 2-21

CosNaming::NamingContext::rebind_context() 2-22

CosNaming::NamingContext::resolve() 2-23
CosNaming::NamingContext::unbind() 2-24
The NamingContextExt Object 2-24
CosNaming::NamingContextExt::resolve_str() 2-26
CosNaming::NamingContextExt::to_name() 2-27
CosNaming::NamingContextExt::to_string() 2-28
CosNaming::NamingContextExt::to_URL() 2-29
The BindingIterator Object 2-29
CosNaming::BindingIterator::destroy() 2-31
CosNaming::BindingIterator::next_n() 2-32
CosNaming::BindingIterator::next_one() 2-33
Exceptions Raised by the CORBA Name Service 2-33
AlreadyBound 2-33
CannotProceed 2-34
InvalidAddress 2-35
InvalidName 2-36
NotEmpty 2-37
NotFound. 2-38

3. Managing a BEA Tuxedo Namespace

Installing the CORBA Name Service. 3-2
Starting the Server Process for the CORBA Name Service. 3-2
Making the Namespace Persistent 3-3
Compressing the Persistent Storage File 3-4
Removing Orphan NamingContext Objects. 3-5
Federating the Namespace 3-5
Inbound Federation 3-6
Outbound Federation. 3-7

Federation Across BEA Tuxedo Domains 3-7
Managing Binding Iterators 3-8
Using the CORBA Name Service in Secure BEA Tuxedo Applications. 3-8

4. Using the CORBA Name Service Sample Application

How the Name Service Sample Application Works. 4-1
Building and Running the Name Service Sample Application. 4-3
Step 1: Copy the Files for the Name Service Sample Application into a Work Directory. 4-3
CORBA C++ Client and Server Version of the Name Service Sample Application 4-3
Step 2: Change the Protection Attribute on the Files for the Name Service Sample Application 4-5
Step 3: Verify the Settings of the Environment Variables 4-6
Step 4: Execute the runme Command 4-7

5. Developing an Application That Uses the CORBA Name Service

Development Steps 5-1
Step 1: Obtain the OMG IDL for the CosNaming Interfaces. 5-2
Step 2: Include the Declarations and Prototypes for the CosNaming Interfaces 5-6
Step 3: Connect to the BEA Tuxedo Namespace 5-6
Step 4: Bind an Object to the BEA Tuxedo Namespace 5-8
Step 5: Use a Name to Locate an Object in the BEA Tuxedo Namespace. 5-9

CHAPTER 1

Overview of the CORBA Name Service

This topic includes the following sections:

- The CORBA Name Service
- Understanding the CORBA Name Service

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.

The CORBA Name Service

The BEA Tuxedo Name Service (referred to throughout this document as the CORBA Name Service) allows BEA Tuxedo CORBA server applications to advertise object references using logical names. BEA Tuxedo CORBA client applications can then locate an object by asking the CORBA Name Service to look up the name.

The CORBA Name Service provides:

- An implementation of the Object Management Group (OMG) Interoperable Name Service (INS) specification.

- Application programming interfaces (APIs) for mapping object references into an hierarchical naming structure (referred to as a namespace).
- Commands for displaying bindings and for binding and unbinding naming context objects and application objects into the namespace.

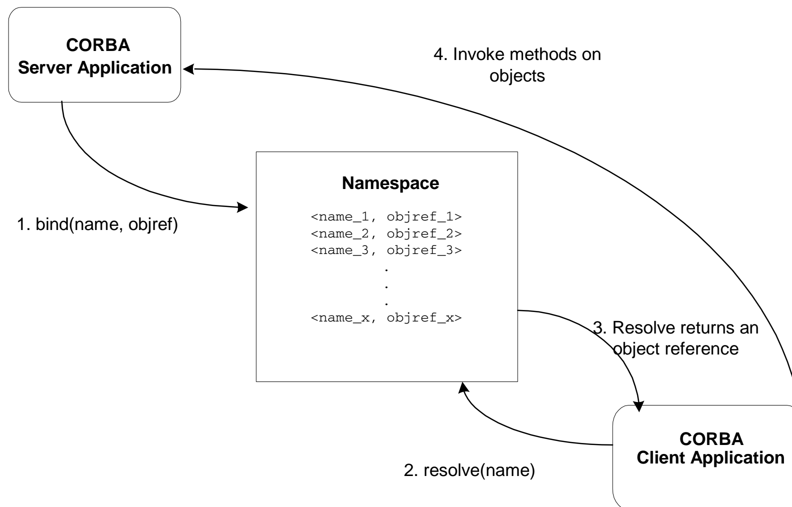
The CORBA Name Service is a layered product. The CORBA Name Service is installed as part of the BEA Tuxedo product. For a complete description of the supported platforms and the installation procedure, see *Installing the BEA Tuxedo System*.

When using the CORBA Name Service:

1. BEA Tuxedo CORBA server applications bind a name to one of its application objects or a naming context object within a namespace.
2. BEA Tuxedo CORBA client applications can then use the namespace to resolve a name and obtain an object reference to the application object or the naming context object.

Figure 1-1 presents an overview of the CORBA Name Service.

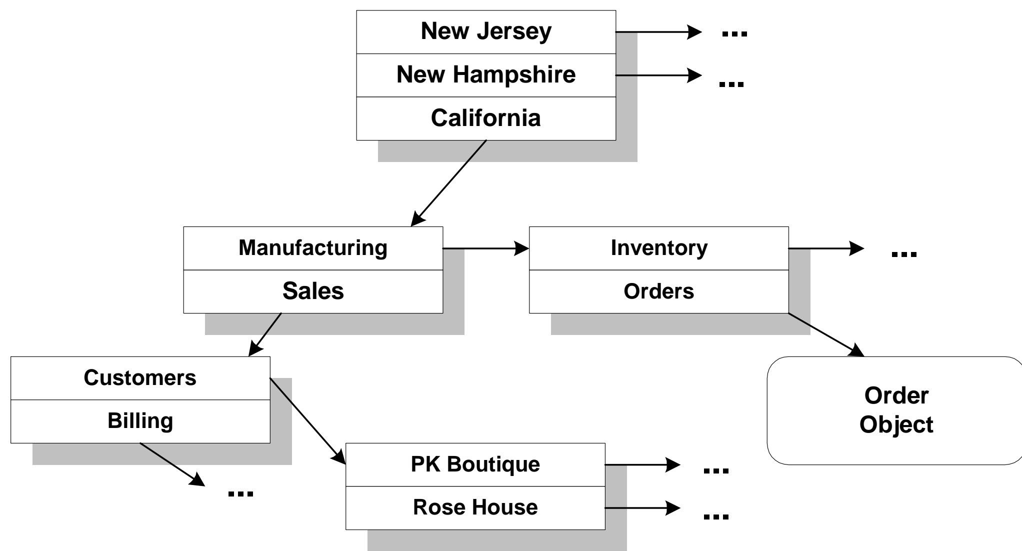
Figure 1-1 CORBA Name Service



Understanding the CORBA Name Service

Figure 1-2 shows how a namespace might be used to store objects that make up an order entry application.

Figure 1-2 A BEA Tuxedo Namespace



The illustrated application organizes its namespace by geographic region, then by department. To implement the namespace using the objects in the CORBA Name Service, each shadowed box would be implemented by a `NamingContext` object. A `NamingContext` object contains a list of `CosNaming::Name` data structures that have been bound to application objects or to other `NamingContext` objects. `NamingContext` objects are traversed to locate a particular name. For example, the logical name `California.Manufacturing.Order` can be used to locate the `Order` object.

A `CosNaming::Name` data structure is not simply a string of alphanumeric characters; it is a sequence of one or more `CosNaming::NameComponent` data structures. Each `CosNaming::NameComponent` data structure contains two strings, `id` and `kind`. The CORBA

Name Service does not interpret or manage these strings, except to ensure that each ID is unique within a given `NamingContext` object.

BEA Tuxedo CORBA server applications use the `bind()` method of the `NamingContext` object to bind a name to an application object contained in the server application. BEA Tuxedo CORBA client applications use the `resolve` method of a `NamingContext` object to locate an object using a binding.

The CORBA Name Service also provides a `BindingIterator` object and a `NamingContextExt` object. The `BindingIterator` object allows a client application to obtain a specified number of bindings in each call. The `NamingContextExt` object provides methods to use Uniform Resource Locators (URL) and stringified names.

For a complete description of the objects in the CORBA Name Service and their interfaces, see Chapter 2, "CORBA Name Service Reference."

CHAPTER 2

CORBA Name Service Reference

This topic includes the following sections:

- CORBA Name Service Commands
- Capabilities and Limitations of the CORBA Name Service
- Getting the Initial Reference to the NameService Environmental Object
- The CosNaming Data Structures Used by the CORBA Name Service
- The NamingContext Object
- The NamingContextExt Object
- The BindingIterator Object
- Exceptions Raised by the CORBA Name Service

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.

CORBA Name Service Commands

The CORBA Name Service provides the following commands to manage the server process for the CORBA Name Service, bind and unbind objects to names in the namespace, and display the contents of the namespace:

- `cns`
- `cnsbind`
- `cnsls`
- `cnsunbind`

The following sections describe these commands.

cns

Synopsis

Controls the server process for the CORBA Name Service.

Syntax

```
cns CLOPT="[-A] [servopts options] --
     [-b bucketcount]
     [-c]
     [-d]
     [-f filename]
     [-M maxiterators]
     [-p [persiststoragefilename] ]"
```

Description

The server process for the CORBA Name Service provides a CORBA CosNaming compliant name service. You need to define the server process for the CORBA Name Service and its options in the `UBBCONFIG` file for your BEA Tuxedo application as you do any other server process used by your BEA Tuxedo application. Enter the `cns` command-line options after the double dash (`--`) in the `CLOPT` parameter of the `UBBCONFIG` file. The command-line options are as follows:

`-b` *`bucketcount`*

Specifies the hash table bucket count used internally by the server process to locate naming contexts in-memory. Each naming context has its own hash table. If your BEA Tuxedo application uses a small number of bindings in each naming context, use a small bucket count (for example, 4 or 5). If your BEA Tuxedo application uses a large number of bindings (for example, 1,000) in each naming context, use a larger number such as 50 for the bucket count.

`-c`

Compresses the persistent storage file when the server process for the CORBA Name Service starts. Over time the persistent storage file can grow in size as naming context and application objects are added and removed from the namespace. Compression reduces the size of the persistent storage file to a minimum. Dangling bindings are removed during compression. Dangling bindings are left in the namespace after the object the binding is associated with is deleted from the namespace. The `-p` command-line option must be specified when specifying the `-c` command-line option.

`-d`

Directs the server process for the CORBA Name Service to delete orphan contexts when the server process starts. An orphan context is a context that is not bound to any other

context. It may never have been bound or it may have been bound to a context and the binding was destroyed either explicitly or as a side-effect of a rebind. The `-p` command-line option must be specified when specifying the `-d` command-line option.

`-f` *`filename`*
: Specifies a file into which the server process for the CORBA Name Service writes the Interoperable Object Reference (IOR) of the root of the namespace.

`-M` *`maxiterators`*
: Defines the maximum number of binding iterators that can be outstanding at any one time.

Binding iterators are created when a client application uses the `CosNaming::NamingContext::list()` method. The client application should use the `CosNaming::BindingIterator::destroy()` method to delete a binding iterator when the client application is done using the binding iterator.

If a client application does not specifically delete binding iterators, the server process for the CORBA Name Service deletes the binding iterators when the number reaches the value specified in the `-M` command-line option. Once the maximum number of binding iterators is reached, any attempt to create a new binding iterator causes the server process for the CORBA Name Service to destroy a binding iterator currently in use by the client application.

Binding iterators are deleted using a least-recently-used algorithm. The default value is 20. A value of 0 indicates that there is no maximum number of binding iterators (meaning binding interators are never destroyed by the server process for the CORBA Name Service and the associated memory is not released). If a value of 0 is specified, the client application must explicitly use the `CosNaming::BindingIterator::destroy()` method to delete outstanding binding iterators.

`-p [`*`persistentstoragefilename`*`]`
: Directs the server process for the CORBA Name Service to save a copy of the current namespace to persistent storage using the specified file. If a filename is not specified, the value of the `CNS_PERSIST_FILE` environment variable is used. If the `CNS_PERSIST_FILE` environment variable is not set, the following files are used:

Windows

`%APPDIR%\cnspersist.dat`

UNIX

`$APPDIR/cnspersist.dat`

The persistent storage file is read when the server process for the CORBA Name Service starts. The persistent storage file is added to as changes are made to the namespace. If you

want to create a new namespace, the existing persistent storage file must be deleted or a new one must be created on the server process for the CORBA Name Service.

cnsbind

Synopsis

Binds application objects and naming context objects into the namespace.

Note: The `cnsbind` command interacts with the CosNaming interfaces. The server process for the CORBA Name Service must be running to use this command.

Syntax

```
cnsbind
    [-C]
    [-f root_context_filename]
    [-h]
    [-N]
    [-o ior_filename]
    [-r]
    [-T TObjAddr]
    bind_name
```

Description

The `cnsbind` command binds new application and naming context objects into the namespace using the CORBA CosNaming interfaces. This command facilitates the creation of a federated namespace. If an exception is returned when the `cnsbind` command is invoked, the command exits and an appropriate message is displayed.

The command-line options for the `cnsbind` command are as follows:

`-C`
: Specifies that the `cnsbind` command creates a context using the *`bind_name`* for the name and the *`ior_filename`* specified for the `-o` command-line option. The `-C` command-line option is used to federate a naming context object from one namespace into the specified namespace.

`-f` *`root_context_filename`*
: Specifies the file containing the IOR of the server process for the CORBA Name Service with which the command interacts to modify the contents of the namespace. If this command-line option is not specified, the command uses the `Tobj_Bootstrap::resolve_initial_references()` method with the NameService environmental object to locate the server process for the CORBA Name Service in the specified BEA Tuxedo domain. The host and port in the IOR must match the value of `TOBJADDR`. This command-line option overrides the setting for the `TOBJADDR`

environment variable. If the command-line option is not specified, the `TOBJADDR` environment variable is used.

`-h`
: Prints the command syntax.

`-N`
: Creates a new context and binds the new context into the namespace using the specified name. The `-o` command-line option is not needed with the `-N` command-line option because the `cnsbind` command is creating a new context. If the `-o` command-line option is used with the `-N` command-line option, the information from the `-o` command-line option is ignored.

`-o` *`ior_filename`*
: Specifies a file that contains the IOR of the object to be bound into the namespace specified via the `-f` command-line option. If the `-C` command-line option is specified, an object of type `ncontext` is created otherwise a object of type `nobject` is created.

`-r`
: Creates a binding for an application or naming context object even if the name already has a binding. The default behavior of the `cnsbind` command without the `-r` command-line option is to raise the `AlreadyBound` exception in the case where a binding for the specified object already exists. If an `AlreadyBound` or any other exception is returned when the `cnsbind` command is invoked, the command exits and an `"Error, already bound"` message is displayed.

`-T` *`TObjAddr`*
: Specifies the host and port for a BEA Tuxedo domain. Before connecting to a server process for the CORBA Name Service, the `cnsbind` command must log into the BEA Tuxedo domain in which the server process is running. This command-line option overrides the setting for the `TOBJADDR` environment variable. If the command-line option is not specified, the value of the `TOBJADDR` environment variable is used. If the command-line option is not specified and `TOBJADDR` is not set, the program will run as a native client and load the TGIOP protocol.

  The valid format for the *`TObjAddr`* specification is `//`*`hostname`*`:`*`port_number`*.

`bind_name`
: Specifies the name to be bound to the application object or name context object added to the namespace relative to either the root naming context retrieved from the `Tobj_Bootstrap::resolve_initial_references` method, or the naming context identified by the stringified IOR obtained from the `-f` command-line option. The *`bind_name`* string should conform to the name string form specified in the Object Management Group (OMG) Interoperable Name Service (INS) specification.

Examples

The following example illustrates binding an application object:

```
cnsbind -o ./app_obj_ior.txt MyContext/AppObject1
```

The following example illustrates binding a naming context object:

```
cnsbind -N MyContext/CtxObject1
```

The following example illustrates binding a federation point to another namespace:

```
cnsbind -C -o ./remote_ior.txt MyContext/RemoteNSCtx1
```

cnsls

Synopsis

Displays the contents of the namespace.

Note: The `cnsls` command interacts with the CosNaming interfaces. The server process for the CORBA Name Service must be running to use this command.

Syntax

```
cnsls
    [-f root_context_filename]
    [-h]
    [-s]
    [-R]
    [-T TobjAddr]
    [resolve_name]
```

Description

The `cnsls` command displays the contents of the namespace using the CORBA CosNaming interfaces. If non-printing characters are used as part of a `NameComponent` data structure, the behavior of the `cnsls` command is undefined. If an exception is returned when the `cnsls` command is invoked, the command exits and an appropriate message is displayed.

The command-line options for the `cnsls` command are as follows:

`-f` *`root_context_filename`*

Specifies the file containing the IOR of the server process for the CORBA Name Service with which the command interacts to modify the contents of the namespace. If this command-line option is not specified, the command uses the `Tobj_Bootstrap::resolve_initial_references()` method with the NameService environmental object to locate the server process for the CORBA Name Service in the specified BEA Tuxedo domain. The host and port in the IOR must match the value of *`TObjAddr`*. This command-line option overrides the setting for the `TOBJADDR` environment variable. If the command-line option is not specified, the value of the `TOBJADDR` environment variable is used.

`-h`

Prints the command syntax.

`-s`

Displays the stringified IOR for the namespace name specified in *`resolve_name`* command-line option.

`-R`

Recursively displays namespace bindings beginning at *`resolve_name`*. This command-line option may cause the `cnsls` command to cross federation boundaries with no indication when such a boundary is cross. Also, if cycles exist in the namespace information, this command-line option can cause the `cnsls` command to enter a loop.

`-T` *`TObjAddr`*

Specifies the host and port for a BEA Tuxedo domain. Before connecting to a server process for the CORBA Name Service, the `cnsls` command must log into the BEA Tuxedo domain in which the server process is running. This command-line option overrides the setting for the `TOBJADDR` environment variable. If the command-line option is not specified, the `TOBJADDR` environment variable is used.

`resolve_name`

Specifies the name to resolve in the name service relative to either the root naming context retrieved via the `Tobj_Bootstrap::resolve_initial_references()` method or the naming context identified by the stringified IOR obtained from the `-f` command-line option. The *`resolve_name`* string should conform to the name string form specified in the OMG INS specification. The backslash (\) character is used to delimit name components and the period (`.`) character separates the `id` and `kind` fields.

If this command-line option is not specified, the root context is resolved.

Example

```
cnsls -R MyContext.kind/AnotherContext
[context] MyContext.kind/AnotherContext
    [object] Obj1
    [object] Obj2
    [context] Ctx1
            [object] AnotherObject
```

cnsunbind

Synopsis

Removes bindings from the namespace.

Syntax

```
cnsunbind
    [-D]
    [-f root_context_filename]
    [-h]
    [-T TObjAddr]
    bind_name
```

Description

The `cnsubind` command removes bindings from the namespace. If an exception is returned when the `cnsunbind` command is invoked, the command exits and an appropriate message is displayed.

The `cnsunbind` command-line options are as follows:

`-D`
: Destroys the naming context bound to the *`bind_name`* after removing the binding. Specifying the `-D` command-line option when deleting a context prevents the context from being orphaned if it is not part of another binding. This command-line option should be used with care because it can cause dangling bindings (for example, if the binding was bound to multiple naming context objects at the same time).

`-f` *`root_context_filename`*
: Specifies the file containing the IOR of the server process for the CORBA Name Service with which the command interacts to modify the contents of the namespace. If this command-line option is not specified, the command uses the `Tobj_Bootstrap::resolve_initial_references()` method with the NameService environmental object to locate the server process for the specified BEA Tuxedo domain.

`-h`
: Prints the command syntax.

`-T` *`TObjAddr`*
: Specifies the host and port for a BEA Tuxedo domain. Before connecting to a server process for the CORBA Name Service, the `cnsbind` command must log into the BEA Tuxedo domain in which the server process is running. This command-line option

overrides the setting for the TOBJADDR environment variable. If the command-line option is not specified, the TOBJADDR environment variable is used.

bind_name
: Specifies the name of the binding to be removed from the namespace relative to either the root naming context retrieved via the `Tobj_Bootstrap::resolve_initial_references()` method or the naming context identified by the stringified IOR obtained from the `-f` command-line option. The *bind_name* string should conform to the name string form specified in the OMG INS specification.

Examples

The following example illustrates removing a binding from the namespace:

```
cnsunbind MyContext/CtxObject1
```

The following example illustrates removing a binding from the namespace and destroying the object to which it was bound:

```
cnsunbind -D MyContext/CtxObject1
```

Capabilities and Limitations of the CORBA Name Service

The CORBA Name Service has the following capabilities and limitations:

- A NULL character must only be used to terminate the `id` and `kind` strings (empty strings are considered valid).
- Wide characters are not supported.
- The CORBA Name Service imposes no limit on the length of the strings in a name component.
- The CORBA Name Service imposes no maximum on the number of components in a name. Zero length names are illegal.
- The CORBA Name Service imposes no limit on the number of bindings in a context.
- The CORBA Name Service imposes no limit on the number of bindings (implementation-wide).
- The CORBA Name Service imposes no limit on the number of contexts.
- The CORBA Name Service deletes orphaned naming contexts and dangling bindings when starting the server process for the CORBA Name Service.

- The CORBA Name Service deletes orphaned naming contexts when starting the server process for the CORBA Name Service.
- The CORBA Name Service offers the option of cleaning up binding iterator objects based on a least-recently-used algorithm. For more information, see "Managing Binding Iterators" on page 3-8.
- The CORBA Name Service does not throw the `CannotProceed` exception.

Getting the Initial Reference to the NameService Environmental Object

A `NameService` environmental object is available for connecting to the root of the namespace. When using the `NameService` environmental object, the Object Request Broker (ORB) locates the root of the namespace. Use the Bootstrap object or the CORBA Interoperable Naming Service (INS) bootstrapping mechanism to get an initial reference to the NameService environmental object. Use the BEA proprietary mechanism if you are using the BEA client ORB. Use the CORBA INS mechanism is you are using a client ORB from another vendor.

For more information on connecting to the namespace, see "Step 3: Connect to the BEA Tuxedo Namespace." For more information about bootstrapping the BEA Tuxedo domain see Chapter 4, "CORBA Bootstrapping Programming Reference," in the *CORBA Programming Reference* in the BEA Tuxedo online documentation.

The CosNaming Data Structures Used by the CORBA Name Service

The CORBA Name Service uses the following CosNaming data structures:

- `CosNaming::BindingList`
- `CosNaming::BindingType`
- `CosNaming::Istring`
- `CosNaming::Name`
- `CosNaming::NameComponent`

The NamingContext Object

The `NamingContext` object is used to contain and manipulate a list of names that are bound to Object Request Broker (ORB) objects or to other `NamingContext` objects. BEA Tuxedo CORBA client applications use this interface to resolve or list all the names within that context. BEA Tuxedo CORBA server applications use this object to bind names to application objects or naming context objects. Listing 2-1 shows the OMG IDL for the `NamingContext` object.

Listing 2-1 OMG IDL for the NamingContext Object

```
module CosNaming {
  interface NamingContext {
    void bind(in Name, in Object obj)
       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind(in Name, in Object obj)
       raises(NotFound, CannotProceed, InvalidName);
    void bind_context(in Name n, in NamingContext nc)
       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void rebind_context(in Name n, in NamingContext nc)
       raises(NotFound, CannotProceed, InvalidName);
    Object resolve(in Name n)
       raises(NotFound, CannotProceed, InvalidName);
    void unbind(in Name n)
       raises(NotFound, CannotProceed, InvalidName);
    NamingContext new_context
    NamingContext bind_new_context(in Name n)
       raises(NotFound, CannotProceed, InvalidName, AlreadyBound);
    void destroy()
       raises(NotEmpty);
    void list(in unsigned long how_many,
              out BindingList bl,
              out BindingIterator bi);
    }
}
```

CosNaming::NamingContext::bind()

Synopsis

Attempts to bind the specified object to the specified name by resolving the context associated with the first `NameComponent` data structure and then binding the object to the new context.

C++ Mapping

```
void bind(in Name n, in Object obj);
```

Parameters

`n`
: A `Name` data structure, initialized with the desired name of the object.

`obj`
: The object to bind to the supplied name.

Exceptions

`AlreadyBound`
: The `Name` on a `bind()` or a `bind_context()` method has already been bound to another object within the naming context.

`InvalidName`
: The specified `Name` has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The `Name` or one of its components, could not be found.

Description

Naming contexts bound with `bind` do not participate in name resolution when compound names are passed to be resolved.

Return Value

None.

CosNaming::NamingContext::bind_context()

Synopsis

This method is similar to the `bind()` method, except that the supplied `Name` is associated with a `NamingContext` object.

C++ Mapping

```
void bind_context(in Name n, in NamingContext nc);
```

Parameters

n
: A `Name` data structure initialized with the desired name for the naming context. The first `NameComponent` data structure in the sequence must resolve to a naming context.

nc
: The `NamingContext` object to be bound to the supplied name.

Exceptions

`AlreadyBound`
: The `Name` on a `bind()` or a `bind_context()` method has already been bound to another object within the naming context.

`InvalidName`
: The specified `Name` has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The `Name` or one of its components, could not be found.

`BAD_PARAM`
: Indicates the call attempted to bind a `NULL` context.

Description

Naming contexts bound with `bind_context()` participate in name resolution when compound names are passed to be resolved.

Return Value

None.

CosNaming::NamingContext::bind_new_context()

Synopsis

Creates a new context and binds it to the specified `Name` within this context.

C++ Mapping

```
NamingContext bind_new_context(in Name n);
```

Parameter

n

A `Name` data structure, initialized with the desired name for the newly created `NamingContext` object.

Exceptions

`AlreadyBound`

The `Name` on a `bind()` or a `bind_context()` method has already been bound to another object within the naming context.

`InvalidName`

The specified `Name` has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`

The `Name` or one of its components could not be found.

Description

This method combines the `CosNaming::NamingContext::new_context()` and `CosNaming::NamingContext::bind_context()` methods into a single method.

Return Value

Returns a reference to a new `NamingContext` object.

CosNaming::NamingContext::destroy()

Synopsis

Deletes a `NamingContext` object. Any subsequent attempt to invoke methods on the `NamingContext` object raises a `CORBA::NO_IMPLEMENT` exception.

C++ Mapping

```
void destroy();
```

Parameter

None.

Exceptions

`NotEmpty`
If the `NamingContext` object contains bindings, the method raises `NotEmpty`.

Description

Before using this method, all name objects that have been bound to the `NamingContext` object should be unbound using the
`CosNaming::NamingContext::unbind()` method.

Return Value

None.

CosNaming::NamingContext::list()

Synopsis

Returns all of the bindings contained by this naming context.

C++ Mapping

```
void list(in unsigned_long how_many,
    out BindingList bl,
    out BindingIterator bi);
```

Parameters

how_many
: The maximum number of bindings to be returned in the list.

bl
: A list of returned bindings where each element is a binding containing a `Name` representing a single `NameComponent` object. Each `Name` is a simple name, that is, a name sequence of length 1. The number of bindings in the list does not exceed the value of *how_many*.

bi
: A reference to a `BindingIterator` object for use in traversing the rest of the bindings.

Exceptions

`InvalidName`
: The specified `Name` has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The `Name` or one of its components could not be found.

Description

This method returns a sequence of name bindings. If more name bindings exist than can fit in the *bl* list, a `BindingIterator` object is returned. The `BindingIterator` object can be used to get the next set of bindings. The `BindingList` (C++) object can return less than the requested number of bindings if it is at the end of the list. If *bi* returns a `NULL` reference, then *bl* contains all of the bindings.

Return Value

None.

CosNaming::NamingContext::new_context()

Synopsis

Creates a new naming context. The newly created context is initially not bound to any `Name`.

C++ Mapping

```
NamingContext new_context();
```

Parameter

None.

Exceptions

None.

Description

Use the `CosNaming::NamingContext::bind_context()` method to bind the new naming context to a `Name`.

Return Value

Returns a reference to a new naming context.

CosNaming::NamingContext::rebind()

Synopsis

This method is similar to the `bind()` method. The difference is that the `rebind` method does not raise the `AlreadyBound` exception. If the specified `Name` has already been bound to another object, that binding is replaced by the new binding.

C++ Mapping

```
void rebind(in Name n, in Object obj);
```

Parameters

n
: A `Name` data structure, initialized with the desired name for the object.

obj
: The object to be named.

Exceptions

`InvalidName`
: The specified `Name` data structure has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The `Name` or one of its components, could not be found. If this exception is raised because the binding already exists or the binding is of the wrong type, the `rest_of_name` member of the exception has a length of 1.

Description

Naming contexts bound with the `rebind()`method do not participate in name resolution when compound names are passed to be resolved.

Return Value

None.

CosNaming::NamingContext::rebind_context()

Synopsis

This method is similar to the `bind_context()` method. The difference is that the `rebind_context` method does not raise the `AlreadyBound` exception. If the specified `Name` has already been bound to another object, that binding is replaced by the new binding.

C++ Mapping

```
void rebind_context(in Name n, in NamingContext nc);
```

Parameters

n
: A `Name` data structure, initialized with the desired name for the object.

nc
: The `NamingContext` object to be rebound.

Exceptions

`InvalidName`
: The specified `Name` data structure has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The component of a name does not identify a binding or the type of binding is incorrect for the operation being performed. If this exception is raised because a binding already exists or it is of the wrong type, the `rest_of_name` member of the exception has a length of 1.

Description

Naming contexts bound with the `rebind_context` method do not participate in name resolution when compound names are passed to be resolved.

Return Value

None.

CosNaming::NamingContext::resolve()

Synopsis

Attempts to resolve the specified `Name`.

C++ Mapping

```
Object resolve(in Name n);
```

Parameters

n

A `Name` data structure, initialized with the desired name for the object.

Exceptions

`InvalidName`

The specified `Name` data structure has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`

The component of a name does not identify a binding or the type of binding is incorrect for the operation being performed.

Description

The specified `Name` must exactly match the name used to bind the object. The CORBA Name Service does not return the type of the object. Client applications are responsible for narrowing the object to the appropriate type.

Return Value

Returns the object reference for the specified `Name`.

CosNaming::NamingContext::unbind()

Synopsis

Performs the inverse operation of the `bind()` method, removing the binding associated with the specified `Name`.

C++ Mapping

```
void unbind(in Name n);
```

Parameters

n
: A `Name` data structure, initialized with the desired name for the object.

Exceptions

`InvalidName`
: The specified `Name` data structure has zero name components or one of the first name components did not resolve to a naming context.

`NotFound`
: The component of a name does not identify a binding or the type of binding is incorrect for the operation being performed.

Description

This method removes the binding between a name and an object. It does not delete the object. Use the `CosNaming::NamingContext::unbind()` method and then the `CosNaming::NamingContext::destroy()` method to delete the object.

Return Value

None.

The NamingContextExt Object

The `NamingContextExt` object provides methods to use URLs and stringified names in the CORBA Name Service. The `NamingContextExt` object is derived from the `NamingContext` object. Note that the root of the CORBA Name Service is a `NamingContextExt` object (which means the root is also a `NamingContext` object). No special operation is needed to obtain a reference to a `NamingContextExt` object. Listing 2-2 shows the OMG IDL for the `NamingContextExt` object.

Listing 2-2 OMG IDL for the NamingContextExt Object

```
module CosNaming {
       interface NamingContextExt : NamingContext {
              typedef string StringName;
              typedef string Address;
              typedef string URLString;

              StringName to_string(in Name n)
                     raises(InvalidName);
              Name to_name(in StringName sn)
                     raises(InvalidName);


              exception InvalidAddress {};

              URLString to_url(in Address addr, in StringName sn)
                     raises(InvalidAddress, InvalidName);
              Object resolve_str(in StringName n)
                     raises(NotFound,
                            CannotProceed,
                            InvalidName,
                            AlreadyBound);
       }
}
```

CosNaming::NamingContextExt::resolve_str()

Synopsis

Takes a stringified name, converts it to a `Name,` and resolves it.

Syntax

```
object resolve_str(in StringName n);
```

Parameter

n

The stringified name to be resolved.

Exceptions

`InvalidName`
The name is invalid. A name of length zero is invalid.

`NotFound`
The component of the name does not identify a binding or the type of the binding is incorrect for the operation being performed.

Description

This is a convenience method that performs a resolve in the same manner as the `CosNaming:NamingContext::resolve()` method. The method accepts a stringified name as an argument instead of a `Name` object. The method returns errors if the stringified name is invalid or if the method cannot bind it.

Return Value

A reference to the bound name.

CosNaming::NamingContextExt::to_name()

Synopsis

Takes a stringified name and returns a `Name` object.

Syntax

```
Name to_name(in StringName sn);
```

Parameter

sn
: The stringified name to be resolved to a `Name` object.

Exceptions

`InvalidName`
: The name is invalid. A name of length zero is invalid.

Description

This method accepts a stringified name and returns a `Name` object. The method returns errors if the name is invalid.

Return Value

Returns a `Name` object.

CosNaming::NamingContextExt::to_string()

Synopsis

Accepts a `Name` object and returns a stringified name.

Syntax

```
StringName to_string(in Name n);
```

Parameter

n
: The `Name` object to be converted to stringified name

Exceptions

`InvalidName`
: The name is invalid. A name of length zero is invalid.

Description

This method accepts a `Name` object and returns a stringified name. It returns errors if the name is invalid.

Return Value

Returns a stringified name.

CosNaming::NamingContextExt::to_URL()

Synopsis

Combines a URL and a stringified name and returns a URL string.

Syntax

```
CosNaming::NamingContextExt::to_URL()

URLString to_URL(in Address addr, in StringName sn);
```

Parameter

addr

A URL. If this parameter is not defined, the local host name is used with the IIOP protocol.

sn

The stringified name to be combined with the URL.

Exceptions

`InvalidAddress`

The URL is invalid.

`InvalidName`

The name is invalid. A name of length zero is invalid.

Return Value

Returns a URL string that combines the URL and the stringified name.

The BindingIterator Object

The `BindingIterator` object allows a client application to walk through the unbounded collection of bindings returned by the `list` method of a `NamingContext` object. Using the `BindingIterator` object, a client application can control the number of bindings obtained with each call. If a naming context is modified between calls to the methods of a `BindingIterator` object, the behavior of further calls to the `next_one()` method or the `next_n()` method is implementation specific.

If a client application creates `BindingIterator` objects but never calls the `destroy` method, the client application can run out of resources. The CORBA Name Service is free to destroy

binding iterators at any time and without warning to the client application. Client applications should be written to expect the `OBJECT_NOT_EXIST` exception from calls to a `BindingIterator` object and to handle this exception gracefully.

Listing 2-3 shows the OMG IDL for the `BindingIterator` object.

Listing 2-3 OMG IDL for BindingIterator Object

```
module CosNaming {
       interface BindingIterator {
              boolean next_one(out Binding b);
              boolean next_n(in unsigned long how_many,
                             out BindingList b);
              void destroy();
       };
}
```

CosNaming::BindingIterator::destroy()

Synopsis

Destroys the `BindingIterator` object and releases the memory associated with the object. Failure to call this method results in increased memory usage.

C++ Mapping

```
void destroy();
```

Parameter

None.

Exceptions

None.

Description

If a client application invokes any operation on a `BindingIterator` object after calling the `destroy` method, the operation raises an `OBJECT_NOT_EXIST` exception.

Return Value

None.

CosNaming::BindingIterator::next_n()

Synopsis

Returns a `BindingList` data structure containing the number of requested bindings from the list. The number of bindings returned may be less than the requested amount if the list is exhausted.

C++ Mapping

```
boolean next_n(in unsigned_long how_many, out BindingList bl);
```

Parameter

how_many
: The maximum number of bindings to return.

bl
: A `BindingList` data structure containing no more than the requested number of bindings.

Exceptions

BAD_PARAM
: Raised if the *how_many* parameter has a value of zero.

Return Value

`CORBA::FALSE` is returned when the list has been exhausted. Otherwise, `CORBA::TRUE` is returned.

CosNaming::BindingIterator::next_one()

Synopsis

Returns the next `Binding` object in the list.

C++ Mapping

```
boolean next_one(out Binding b);
```

Parameter

b

The next `Binding` object from the list.

Exceptions

None.

Return Value

`CORBA::FALSE` is returned when the list has been exhausted. Otherwise, `CORBA::TRUE` is returned.

Exceptions Raised by the CORBA Name Service

This section describes the exceptions raised by the CORBA Name Service.

AlreadyBound

Syntax

```
exception AlreadyBound{};
```

Parameter

None.

Description

This exception is raised when an object is already bound to the supplied name. Only one object can be bound to a name in a context.

CannotProceed

Syntax

```
exception CannotProceed{};
```

Parameters

`NamingContext` *`cxt`*

The context that the operation may be able to retry from.

`Name` *`rest_of_name`*

The remainder of the non working name.

Description

This exception is raised when an unexpected exception is encountered and the method cannot proceed in a meaningful way.

InvalidAddress

Syntax

```
exception InvalidAddress{};
```

Parameter

None.

Description

This exception is raised if a URL is invalid.

InvalidName

Syntax

```
exception InvalidName{};
```

Parameter

None.

Description

This exception is raised if a `Name` is invalid. A name length of zero is invalid.

NotEmpty

Syntax

```
exception NotEmpty{};
```

Parameter

None.

Description

This exception is raised when the `destroy()` method is used on a `NamingContext` object that contains bindings. A `NamingContext` object must be empty before it is destroyed.

NotFound

Syntax

```
exception NotFound{NotFoundReason why; Name rest_of_name;};
```

Parameters

why

The context that the operation may be able to retry from.

rest_of_name

The remainder of the non-working name.

Description

This exception is raised when a component of the name does not identify a binding, or if the type of binding is incorrect for the operation being performed. The *why* parameter explains the reason for the error. The *rest_of_name* parameter identifies the cause of the error. The following causes can appear:

- `missing_node`—the first name component in the *rest_of_name* parameter is a binding that is not bound under that name within its parent context.
- `not_context`—the first name component in the *rest_of_name* parameter is a binding with a type of `nobject` when the type of `ncontext` was required.
- `not_object`—the first name component in the *rest_of_name* parameter is a binding with a type of `ncontext` when the type of `nobject` was required.

CHAPTER 3

Managing a BEA Tuxedo Namespace

This topic includes the following sections:

- Installing the CORBA Name Service
- Starting the Server Process for the CORBA Name Service
- Making the Namespace Persistent
- Compressing the Persistent Storage File
- Removing Orphan NamingContext Objects
- Federating the Namespace
- Managing Binding Iterators

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.

Installing the CORBA Name Service

You install the CORBA Name Service when you install BEA Tuxedo. For complete information about installing BEA Tuxedo, see *Installing the BEA Tuxedo System*.

Starting the Server Process for the CORBA Name Service

To start the server process for the CORBA Name Service, you need to define the server process in the `UBBCONFIG` file for your BEA Tuxedo CORBA application. Use the `cns` command to start the server process for the CORBA Name Service. List the `cns` command-line options after the `CLOPT` parameter in the `UBBCONFIG` file. Note there can be only one CORBA Name Service server process running per BEA Tuxedo domain. Listing 3-1 is an example of the `UBBCONFIG` entry for the server process for the CORBA Name Service.

Listing 3-1 UBBCONFIG File Entry for the CORBA Name Service

```
...
#
#Server process for BEA Tuxedo CORBA Name Service
#
       cns
              SRVGRP = SYS_GRP
              SRVID = 6
              RESTART = N
       CLOPT = "-A -- -f C:\cnsroot.dat -M 0"
```

For a complete description of the `cns` command and its options, see Chapter 2, "CORBA Name Service Reference." For information about creating a configuration file, see *Setting Up a BEA Tuxedo Application* in the BEA Tuxedo online documentation.

Once the server process for the CORBA Name Service is started, you can use the commands listed in Table 3-1 to display the contents of the namespace and manage objects in the namespace. For a complete description of the commands and their options, see Chapter 2, "CORBA Name Service Reference."

Table 3-1 Commands for Managing a BEA Tuxedo Namespace

Command	Description
`cns`	Starts the server process for the BEA Tuxedo namespace.
`cnsbind`	Binds application objects and naming context objects to the BEA Tuxedo namespace.
`cnsls`	Displays the contents of a BEA Tuxedo namespace.
`cnsunbind`	Removes bindings from a BEA Tuxedo namespace.

Making the Namespace Persistent

The CORBA Name Service keeps two copies of the information in a namespace. One copy is kept in-memory. Access to this copy is fast and optimized for name resolution. The other copy is optionally saved to persistent storage allowing the state and structure of the namespace to be saved and restored.

The primary goal of making a namespace persistent is to keep a current representation of the in-memory naming information maintained by the currently running instance of the namespace. By maintaining a persistent copy of the namespace, the CORBA Name Service can recreate current naming information in case the server process of the CORBA Name Service is terminated. A new instance of the server process for the CORBA Name Service can be configured to read the persistent storage file to recreate the last recorded naming information.

To create a persistence copy of the namespace and store the copy to a file, specify the `-p` option of the `cns` command when starting the server process for the CORBA Name Service. The CORBA Name Service creates a persistent storage file with the specified location and name.

If the persistent storage file specified by the `-p` option already exists, the file is opened and processed. A backup of the persistent storage file is always made prior to the startup of the server process for the CORBA Name Service. The name of the backup copy of the persistent storage file is *`filename`*`.BAK`. If you want to reuse the name of the persistent storage file, you must delete or move the existing file and then restart the server process for the CORBA Name Service.

If the persistent storage file is successfully created, an entry for the file is written to the `ULOG` file. The entry indicates the directory location and name of the file, whether or not the file was newly created, and the mechanism used to determine the name of the file (for example, specified,

environmental, or default). If an error occurs when creating the persistent storage file, an entry is written to the ULOG file indicating the type of error that occurred.

Since the server process for the CORBA Name Service recreates the structure of the namespace from the persistent storage file at startup, the startup time is directly proportional to the size of the persistent storage file.Very large persistent storage files (on the order of hundreds of megabytes) can result in the server process for the CORBA Name Service taking several seconds or even minutes to recreate the namespace at startup.

Compressing the Persistent Storage File

The persistent storage file contains information about all operations affecting the in-memory copy of the namespace. Over time, the persistent storage file can contain more information than is necessary to recreate the structure and state of the current namespace. In fact, the persistent storage file can grow quite large even though the structure of the namespace stays the same size.

The CORBA Name Service allows you to compress the persistent storage file to remove unneeded information. The `-c` option of the `cns` command controls compression of the persistent storage file. The compression option processes the current information to produce a new compressed persistent storage file.

When the server process for the CORBA Name Service is started, the compression operation performs the following:

1. Processes the in-memory structure of the namespace.
2. Overwrites the existing persistent storage file.
3. Deletes all bind and rebind entries which were removed from the namespace by unbind, rebind, or destroy operations.
4. Removes all dangling bindings. Dangling bindings are bindings left in the namespace after the object the binding is associated with is deleted from the namespace. Dangling bindings occur when a `CosNaming::NamingContext::destroy()` method is performed on a naming context without the naming context being unbound from its parent context.

The `-c` option can only be used if the `-p` option of the `cns` command is specified. For a complete description of the `-c` option of the `cns` command, see Chapter 2, “CORBA Name Service Reference.”

Removing Orphan NamingContext Objects

An orphan context is a context that is not bound to any other context. The context may have never been bound or it may have been bound and the binding was destroyed either explicitly or as the result of a rebind. In the CORBA Name Service, orphan `NamingContext` objects are created in one of the following ways:

- Using the `CosNaming::NamingContext::new_context` method to create a new `NamingContext` object but never binding the new `NamingContext` object to the namespace.
- Using the `CosNaming::NamingContext::rebind()` or `CosNaming::NamingContext::unbind()` methods to unbind the `NamingContext` object from their last parent `NamingContext` object but never destroying the `NamingContext` object.

Client applications and other namespaces federated to the `NamingContext` object can perform operations on orphan `NamingContext` objects as long as they maintain the object reference to the orphan `NamingContext` object.

The current implementation of the namespace maintains the orphan `NamingContext` objects in a special `LostandFoundContext` object.

Use the `-d` option of the `cns` command to delete orphan `NamingContext` objects from the namespace. The `-d` option unbinds and destroys all `NamingContext` objects identified as orphaned.

The `-d` option can only be used if the `-p` option of the `cns` command is specified. For a complete description of the `-d` option of the `cns` command, see Chapter 2, “CORBA Name Service Reference.”

Federating the Namespace

The CORBA Name Service supports the concept of a federated namespace which means elements of a logical namespace may reside on multiple, disparate, and remote machines. In a federated namespace, a `NamingContext` object can be bound to one namespace using an object reference to a `NamingContext` object maintained by another namespace. The CORBA Name Service supports federation with implementations of the CORBA Name Service running on other machines as well as third-party name services. In order for the CORBA Name Service to federate with a third-party name service, the third-party name service must support the naming formats specified in the Object Management Group (OMG) Interoperable Name Service (INS) specification.

The following topics explain how to support inbound and outbound federation as well as federation with third-party name services.

Inbound Federation

Inbound federation is the ability to bind a `NamingContext` object which exists in a local BEA Tuxedo namespace into a namespace on a remote name service. Once the namespaces are federated, the remote name service can perform operations on `NamingContext` objects in a the BEA Tuxedo namespace. Inbound federation with a third-party name service is done using the Internet Inter-Orb Protocol (IIOP). Therefore, the IIOP Listener/Handler for the CORBA Name Service must be configured to support unoffical IIOP connections.

To enable the unofficial connections on the IIOP Listener/Handler, use the `-C` parameter of the ISL command. The `-C` parameter determines how the IIOP Listener/Handler will behave when unofficial connections are made to it. To use inbound federation, specify the `warn` or `none` values for the `-C` parameter. A value of `warn` causes the IIOP Listener/Handler to log a message to the user log exception when an unofficial connection is detected; no exception will be raised. A value of `none` causes the IIOP Listener/Handler to ignore unofficial connections. For more information about the ISL command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Listing 3-2 shows an example of the `UBBCONFIG` entry for the IIOP Listener/Handler that supports inbound federation.

Listing 3-2 UBBCONFIG File Entry for an IIOP Listener/Handler That Supports Inbound Federation

```
#
# Entry to start IIOP Listener/Handler
#  that supports inbound federation

ISL
              SRVGRP = SYS_GRP
              SRVID  = 5
              MIN    = 1
              MAX    = 1
              CLOPT  = "-A -- -n //blotto:2470 -C none"
```

Outbound Federation

Outbound federation is the ability to bind a `NamingContext` object which exists in a remote name service into the namespace of a CORBA Name Service. Operations can then be performed on this `NamingContext` object using the CORBA Name Service. Outbound federation with a third-party name service is done using IIOP. Therefore, the IIOP Listener/Handler for the CORBA Name Service must be configured to support outbound federation.

To enable outbound federation on the IIOP Listener/Handler, use the `-O` parameter of the ISL command. The `-O` parameter causes the IIOP Listener to allow outbound IIOP invocations to objects located in server applications not connected to a IIOP Handler. For more information about the ISL command, see the *BEA Tuxedo Command Reference* in the BEA Tuxedo online documentation.

Listing 3-3 shows an example of the `UBBCONFIG` entry for the IIOP Listener/Handler that supports outbound federation.

Listing 3-3 UBBCONFIG File Entry for an IIOP Listener/Handler That Supports Outbound Federation

```
#
# Entry for IIOP Listener/Handler
# that supports outbound federation
#
ISL
              SRVGRP = SYS_GRP
              SRVID  = 5
              MIN    = 1
              MAX    = 1
              CLOPT  = "-A -- -n //blotto:2470 -O"
```

Federation Across BEA Tuxedo Domains

Federation across multiple CORBA Name Service server processes running in different BEA Tuxedo domains requires the use of Domain Gateways to allow for inter-domain communication. For more information about configuring a domain gateway, see the “Configuring Multiple Domains (BEA Tuxedo System)” section in the Administration topic.

Managing Binding Iterators

The OMG INS specification allows the collection of outstanding binding iterators. Since binding iterators require explicit destruction by client applications, there is the chance that binding iterators will not be deleted properly.

To minimize the possibility that the CORBA Name Service will run out of resources due to a large number of unused binding iterators, use the `-M` option of the `cns` command to set the maximum number of binding iterators in the name service. Once the limit has been reached, requests for new binding iterators may result in the destruction of outstanding binding iterators. The CORBA Name Service uses a least-recently-used algorithm to select which binding iterators are deleted.

If the server process for the CORBA Name Service is started with the `-M` option, the CORBA Name Service may destroy a binding iterator that is currently being used by a BEA Tuxedo CORBA application so all BEA Tuxedo applications need to be able to handle the `CORBA::OBJECT_NOT_EXIST` system exception when invoking on binding iterators.

Using the CORBA Name Service in Secure BEA Tuxedo Applications

When using the `cnsls`, `cnsbind`, and `cnsunbind` commands in a secure BEA Tuxedo CORBA application, you need to obtain the PrincipalAuthenticator object for the BEA Tuxedo domain and log on to the domain with the appropriate security information.

In order for a secure logon to occur, the BEA Tuxedo domain must be configured with a security level of `TOBJ_SYSAUTH` or `TOBJ_APPAUTH`. The username for the `cnsls`, `cnsbind`, and `cnsunbind` commands is `cnsutils`. You need to use the `tpusradd` command to create a user named `cnsutils`. Depending on the Security level specified for the BEA Tuxedo domain, the user password and/or the domain password may be defined in the `UBBCONFIG` file in the `USER_AUTH` and `APP_PW` environment variables. If these environment variables are not set and the BEA Tuxedo domain has a security level of `TOBJ_SYSAUTH` or `TOBJ_APPAUTH`, the `cnsls`, `cnsbind`, and `cnsunbind` commands will prompt for a password.

For more information about security levels and defining users in the BEA Tuxedo security environment, see *Using Security in CORBA Applications* in the BEA Tuxedo online documentation.

CHAPTER 4

Using the CORBA Name Service Sample Application

This topic includes the following sections:

- How the Name Service Sample Application Works
- Building and Running the Name Service Sample Application

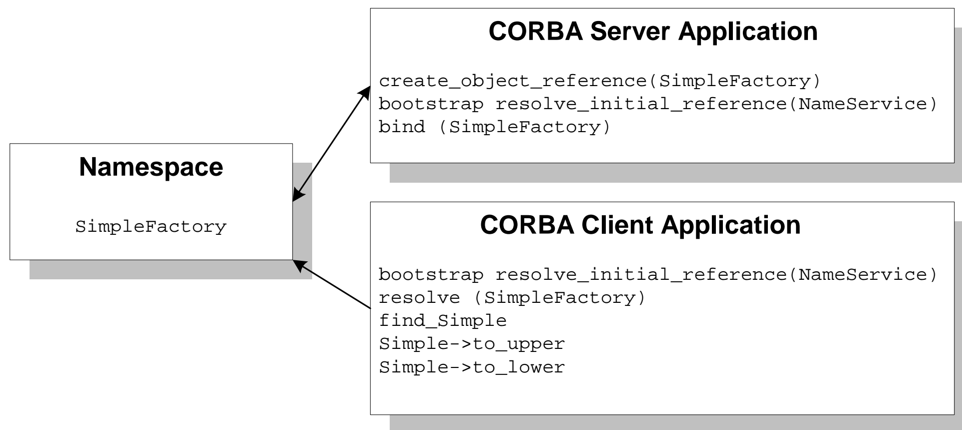
Note: The BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB were deprecated in Tuxedo 8.1 and are no longer supported. All BEA Tuxedo CORBA Java client and BEA Tuxedo CORBA Java client ORB text references, associated code samples, should only be used to help implement/run third party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

How the Name Service Sample Application Works

The CORBA Name Service sample application is a modification of the Simpapp sample application. This sample application provides a CORBA C++ client and server. The Name Service sample application uses a namespace to store the `SimpleFactory` object. The server application creates the `SimpleFactory` object and binds the object to the namespace. The client application connects to the namespace, resolves the name of the `SimpleFactory` object, and then invokes methods on the `SimpleFactory`. Figure 4-1 illustrates how the Name Service sample application works.

Figure 4-1 The Name Service Sample Application



The Name Service sample application implements the CORBA interfaces listed in Table 4-1:

Table 4-1 CORBA Interfaces for the Name Service 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 4-1 shows the `simple.idl` file that defines the CORBA interfaces in the Name Service sample application.

Listing 4-1 OMG IDL Code for the Name Service 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();
};
```

Building and Running the Name Service Sample Application

To build and run the Name Service sample application, complete the following steps:

1. Copy the files for the Name Service sample application into a work directory.
2. Change the permissions on the files in the work directory.
3. Verify the locations defined in environment variables.
4. Execute the `runme` command.

Step 1: Copy the Files for the Name Service Sample Application into a Work Directory

Copy the files for the Name Service sample application into a work directory on your local machine. Running the sample application in a work directory allows you to identify the files that are created when the sample is executed. The following sections detail the directory location and sources files for the the Name Service sample application.

CORBA C++ Client and Server Version of the Name Service Sample Application

The files for the Name Service sample application are located in the following directories:

Windows

drive:*tuxdir*\samples\corba\cnssimpapp

UNIX

/*usr/local/tuxdir*/samples/corba/cnssimpapp

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

Table 4-2 Files Included in the Name Service Sample Application

File	Description
`simple.idl`	The OMG IDL code that declares the `Simple` and `SimpleFactory` interfaces.
`simples.cpp`	The C++ source code for the CORBA server application in the Name Service sample application.
`simplec.cpp`	The C++ source code for the CORBA client application in the Name Service sample application.
`simple_i.cpp`	The C++ source code that implements the `Simple` and `SimpleFactory` methods.
`simple_i.h`	The C++ header file that defines the implementation of the `Simple` and `SimpleFactory` methods.
`Readme.txt`	Provides information about building and running the C++ client and server of the Name Service sample application.
`runme.cmd`	The Windows command file that builds and runs the Name Service sample application.
`runme.ksh`	The UNIX Korn shell script that builds and executes the Name Service sample application.

Table 4-2 Files Included in the Name Service Sample Application (Continued)

File	Description
makefile.mk	The makefile for the Name Service sample application on UNIX operating systems. This file is used to build the Name Service sample application manually. See the Readme.txt file for additional information. The location of the executable UNIX make command must be defined in the PATH environment variable.
makefile.nt	The makefile for the Name Service sample application on the Windows operating system. This makefile can be used directly by the Visual C++ nmake command. This file is used to manually build the Name Service sample application. See the Readme.txt file for more information. The location of the executable Windows nmake command must be defined in the PATH environment variable.

Step 2: Change the Protection Attribute on the Files for the Name Service Sample Application

The files for the sample application are installed with a permission level of read only. Before you can edit or build the files in the Name Service sample application, you must change the protection attribute of the files you copied into your work directory, as follows:

Windows

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

UNIX

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

On UNIX platforms, you also need to change the permission of `runme.ksh` to allow execute permission, as follows:

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

Step 3: Verify the Settings of the Environment Variables

Before running the Name Service sample application, you need to verify that certain environment variables are defined to correct locations. In most cases, these environment variables are set as part of the installation procedure. Some environment variables are set when you execute the `runme` command. You need to check the environment variables to ensure they reflect correct information.

Table 4-3 lists the environment variables required to run the Name Service sample application.

Table 4-3 Required Environment Variables for the Name Service Sample Application

Environment Variable	Description

`APPDIR`	Execution of the `runme` command sets this environment variable to the absolute path name of the current directory. Execute the `runme` command from the directory to which you copied the sample application files. For example:
	Windows
	`APPDIR=C:\workdirectory\cnssimpapp`
	UNIX
	`APPDIR=/usr/workdirectory/cnssimpapp`

`RESULTSDIR`	Execution of the `runme` command sets this environment variable to the `results` directory, subordinate to the location defined by the `APPDIR` environment variable.
	Windows
	`RESULTSDIR=%APPDIR%\results`
	UNIX
	`RESULTSDIR=$APPDIR\results`

`TUXCONFIG`	Execution of the `runme` command sets this environment variable to the directory path and filename of the configuration file.
	Windows
	`TUXCONFIG=%RESULTSDIR%\tuxconfig`
	UNIX
	`TUXCONFIG=$RESULTSDIR/tuxconfig`

To verify that the information for the environment variables defined during installation is correct, complete 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 settings of the environment variables.

UNIX

```
ksh prompt> printenv TUXDIR
```

To change the settings, complete the following steps:

Windows

1. On the Environment page in the System Properties window, click the environment variable you want to change or enter the name of the environment variable in the `Variable` field.
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
```

Step 4: Execute the runme Command

The `runme` command completes the following steps end-to-end:

1. Sets the system environment variables.
2. Loads the `UBBCONFIG` file.
3. Compiles the code for the client application.
4. Compiles the code for the server application.

5. Starts the server application using the `tmboot` command.
6. Starts the client application.
7. Stops the server application using the `tmshutdown` command.

Note: You can also run the Name Service sample application manually. The steps for manually running the Name Service sample application are described in the `Readme.txt` file.

To build and run the Name Service sample application, enter the `runme` command, as follows:

Windows

```
prompt> cd workdirectory
prompt> runme
```

UNIX

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

When the Name Service sample application runs successfully from start to finish, this series of messages is printed:

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

Table 4-4 lists the files in the work directory generated by the `runme` command.

Table 4-4 C++ Files Generated by the runme Command

File	Description
simple_c.cpp	Generated by the idl command, this file contains the client stubs for the SimpleFactory and Simple interfaces.
simple_c.h	Generated by the idl command, this file contains the client definitions of the SimpleFactory and Simple interfaces.
simple_s.cpp	Generated by the idl command, this file contains the server skeletons for the SimpleFactory and Simple interfaces.
simple_s.h	Generated by the idl command, this file contains the server definition for the SimpleFactory and Simple interfaces.
.adm/.keybd	A file that contains the security encryption key database. The subdirectory is created by the tmloadcf command in the runme command.
results	A directory created by the runme command, subordinate to the location defined by the APPDIR environment variable.

Table 4-5 lists files in the results directory generated by the runme command.

Table 4-5 Files in the results Directory Generated by the runme Command

File	Description
input	Contains the input that the runme command provides to the Java client application.
output	Contains the output produced when the runme command executes the Java client application.

Table 4-5 Files in the results Directory Generated by the runme Command (Continued)

File	Description
expected_output	Contains the output that is expected when the Java client application is executed by the runme command. The data in the output file is compared to the data in the expected_output file to determine whether or not the test passed or failed.
log	Contains the output generated by the runme command. If the runme command fails, check this file for errors.
setenv.cmd	Contains the commands to set the environment variables needed to build and run the Java Name Service sample application on the Windows operating system platform.
setenv.ksh	Contains the commands to set the environment variables needed to build and run the Java Name Service sample application on UNIX operating system platforms.
stderr	Output from commands generated by the tmboot command, which is executed by the runme command. If the -noredirect JavaServer option is specified in the UBBCONFIG file, the System.err.println method sends the output to the stderr file instead of to the ULOG file.
stdout	Output generated by the tmboot command, which is executed by the runme command. If the -noredirect JavaServer option is specified in the UBBCONFIG file, the System.out.println method sends the output to the stdout file instead of to the ULOG file.
tmsysevt.dat	Contains filtering and notification rules used by the TMSYSEVT (system event reporting) process. This file is generated by the tmboot command in the runme command.
tuxconfig	A binary version of the UBBCONFIG file.

Table 4-5 Files in the results Directory Generated by the runme Command (Continued)

File	Description
`ubb`	The `UBBCONFIG` file for the Java Name Service sample application.
`ULOG.`*`date`*	A log file that contains messages generated by the `tmboot` command.

CHAPTER 5

Developing an Application That Uses the CORBA Name Service

This topic includes the following sections:

- Development Steps
- Step 1: Obtain the OMG IDL for the CosNaming Interfaces
- Step 2: Include the Declarations and Prototypes for the CosNaming Interfaces
- Step 3: Connect to the BEA Tuxedo Namespace
- Step 4: Bind an Object to the BEA Tuxedo Namespace
- Step 5: Use a Name to Locate an Object in the BEA Tuxedo Namespace

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.

Development Steps

Table 5-1 outlines the process for developing BEA Tuxedo CORBA applications that use the CORBA Name Service.

Table 5-1 Development Process

Step	Description
1	Obtain the OMG IDL for the CosNaming interfaces.
2	Include the declarations and prototypes for the CosNaming interfaces.
3	Connect to the BEA Tuxedo namespace.
4	Bind an object to the BEA Tuxedo namespace.
5	Use a name to locate an object in the BEA Tuxedo namespace.

Before performing the steps in this topic, you need to start the server process for the CORBA Name Service. For more information, see "Starting the Server Process for the CORBA Name Service" on page 3-2.

After performing the development steps in this topic, use the `buildobjclient` and `buildobjserver` commands to compile server and client applications that use the CORBA Name Service. For more information about the `buildobjclient` and `buildobjserver` commands, see the *BEA Tuxedo Command Reference*.

Step 1: Obtain the OMG IDL for the CosNaming Interfaces

A BEA Tuxedo CORBA application accesses the CORBA Name Service using the interfaces defined in `CosNaming.idl`. This Object Management Group (OMG) Interface Definition Language (IDL) file defines the interfaces, COSnaming data structures, and exceptions used by the CORBA Name Service. The `CosNaming.idl` file is located in the following directory locations:

Windows

```
drive:\%TUXDIR%\include\CosNaming.idl
```

UNIX

```
/usr/local/$TUXDIR/include/CosNaming.idl
```

Listing 5-1 shows the OMG IDL for CosNaming.idl. The same OMG IDL file is used by both CORBA C++ applications.

Listing 5-1 CosNaming.idl

```
#ifndef _COSNAMING_IDL_
#define _COSNAMING_IDL_

module CosNaming {

#pragma prefix "omg.org/CosNaming"
       typedef string Istring;

       struct NameComponent {
              Istring id;
              Istring kind;
       };

       typedef sequence<NameComponent> Name;

       enum BindingType { nobject, ncontext };

       struct Binding {
              Name        binding_name;
              BindingType binding_type;
       };

       typedef sequence <Binding> BindingList;

       interface BindingIterator;

       interface NamingContext {
              enum NotFoundReason { missing_node,
                                    not_context,
                                    not_object };

              exception NotFound {
                     NotFoundReason  why;
                     Name            rest_of_name;
              };

              exception CannotProceed {
                     NamingContext   cxt;
                     Name            rest_of_name;
```

```
};

exception InvalidName{};

exception AlreadyBound {};

exception NotEmpty{};

void    bind(in Name n, in Object obj)
        raises(NotFound,
              CannotProceed,
              InvalidName,
              AlreadyBound);

void   rebind(in Name n, in Object obj)
       raises(NotFound,
              CannotProceed,
              InvalidName);

void    bind_context(in Name n, in NamingContext nc)
       raises(NotFound,
              CannotProceed,
              InvalidName,
              AlreadyBound);

void    rebind_context(in Name n, in NamingContext nc)
       raises(NotFound,
              CannotProceed,
              InvalidName);

Object  resolve (in Name n)
       raises(NotFound,
              CannotProceed,
              InvalidName);

void    unbind(in Name n)
       raises(NotFound,
              CannotProceed,
              InvalidName);
```

```
        NamingContext   new_context();
        NamingContext   bind_new_context(in Name n)
                raises(NotFound,
                        AlreadyBound,
                        CannotProceed,
                        InvalidName);

        void    destroy() raises(NotEmpty);
        void    list(in unsigned long    how_many,
                     out BindingList     bl,
                     out BindingIterator bi);
};

interface BindingIterator {
        boolean next_one(out Binding b);
        boolean next_n(in unsigned long how_many,
                       out BindingList bl);
        void    destroy();
};

interface NamingContextExt:NamingContext {
        typedef string StringName;
        typedef string Address;
        typedef string URLString;

StringName  to_string(in Name n) raises(InvalidName);
Name        to_name(in StringName sn)
                    raises(InvalidName);
exception InvalidAddress {};

URLString   to_url(in Address addr, in StringName sn)
                    raises(InvalidAddress, InvalidName);

Object      resolve_str(in StringName n)
                    raises(NotFound,
                            CannotProceed,
                            InvalidName,
                            AlreadyBound
              );
```

```
        };
};
#pragma ID CosNaming "IDL:omg.org/CosNaming:1.0"
#endif // _COSNAMING_IDL_
```

Step 2: Include the Declarations and Prototypes for the CosNaming Interfaces

The declarations and prototypes for the CosNaming interfaces are provided as part of the software kit for the CORBA Name Service.

- For CORBA C++ client applications, include the declarations and prototypes for the naming interfaces by adding this statement to your BEA Tuxedo CORBA client application:

  ```
  #include "CosNaming_c.h"
  ```

  The include files for a BEA Tuxedo CORBA C++ client application are located in the `$TUXDIR/include` directory on UNIX systems and the `%TUXDIR%\include` directory on Windows systems.

- If you are using a third-party Object Request Broker (ORB), you need to include or import the CosNaming interfaces in your client source stub programs before compiling.

Step 3: Connect to the BEA Tuxedo Namespace

The Bootstrap object supports a `NameService` environmental object for connecting to the root of the namespace. When using the NameService environmental object, the Object Request Broker (ORB) locates the root of the namespace. The object reference can then be narrowed to `CosNaming::NamingContext` or `CosNamingContextExt`. You need to connect to the BEA Tuxedo namespace before binding objects into the namespace and resolving names in the namespace.

Use the Bootstrap object or the CORBA Interoperable Naming Service (INS) bootstrapping mechanism to get an initial reference to the NameService environmental object. Use the BEA proprietary mechanism if you are using the BEA client ORB. 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 Chapter 4, “CORBA Bootstrapping Programming Reference,” in the *CORBA Programming Reference* in the BEA Tuxedo online documentation.

Listing 5-2 illustrates C++ code that establishes communication with a BEA Tuxedo namespace.

Listing 5-2 C++ Example of Connecting to a Namespace

```
...
Tobj_Bootstrap * bootstrap = new Tobj_Bootstrap (v_orb.in(), "");
CORBA::Object_var var_nameservice_oref=
       bootstrap.resolve_initial_references("NameService");
root = CosNaming::NamingContext::_narrow (obj);
...
```

A stringified object reference for the root of the namespace can also be used to connect to a namespace in a BEA Tuxedo domain. In order to use a stringified object reference, the `-f` command-line option must be specified when starting the server process for the CORBA Name Service. The `-f` command-line option writes the stringified object reference to the `CNS_ROOT_FILE` environment variable or to one of the following locations:

Windows

`%APPDIR%\cnsroot.dat`

UNIX

`$APPDIR/cnsroot.dat`

The stringified object reference for the root of the namespace does not change when the server process for the CORBA Name Service is started and stopped because stringified object reference is associated with a particular host machine rather than a particular server process. A stringified object reference that has been retrieved to communicate with one BEA Tuxedo namespace cannot be used to communicate with another BEA Tuxedo namespace.

Listing 5-3 includes C++ code that establishes communication with a BEA Tuxedo namespace using a stringified object reference.

Listing 5-3 C++ Example of Using a Stringified Object Reference

```
...
Tobj_Bootstrap * bootstrap;
bootstrap = new Tobj_Bootstrap (v_orb.in(), "");
CORBA::Object_var obj = GetRefFromFile ("cnsroot.dat", v_orb);
```

```
root = CosNaming::NamingContext::_narrow (obj);
...
```

If you choose to use a stringified object reference in a BEA Tuxedo CORBA application that also employs security and transactions, please note the following restrictions:

1. The BEA Tuxedo CORBA application must create a Bootstrap object and connect to the IIOP Listener/Handler before using the stringified object reference to connect to a BEA Tuxedo namespace. By calling the Bootstrap object first, the BEA Tuxedo application establishes an official connection to the IIOP Listener/Handler.

   If a BEA Tuxedo application does not first create a Bootstrap object, transactions and security cannot be used with any object retrieved from the namespace. Transactions and security require the use of an official connection.

2. If more than one IIOP Listener/Handler is defined in the `UBBCONFIG` file, the BEA Tuxedo CORBA application must use the first IIOP Listener/Handler defined in the `UBBCONFIG` file by the `TOBJADDR` environment variable.

   The CORBA Name Service creates the stringified object reference for the root of the namespace, using the default IIOP Listener/Handler's host and port. The first IIOP Listener/Handler defined in a `UBBCONFIG` file is considered the default IIOPListener/Handler. Using the default IIOP Listener/Handler causes all object references retrieved by the CORBA Name Service to be official connections. Transactions and security require the use of official connections.

Step 4: Bind an Object to the BEA Tuxedo Namespace

There are two ways to bind an object to the BEA Tuxedo namespace:

- The `cnsbind` command
- The `bind()` method of the `CosNaming::NamingContext` object

The `cnsbind` command can be used to bind application objects or naming context objects to the BEA Tuxedo namespace. The server process for the CORBA Name Service must be started before using the `cnsbind` command. For a complete description of the `cnsbind` command, see Chapter 2, "CORBA Name Service Reference."

Listing 5-4 show the C++ code implementations of the `bind()` method of the `CosNaming::NamingContext` object. The code examples accept two parameters, representing the `id` and `kind` fields for a `Name`. These parameters initialize a `Name` for the `SimpleFactory` object and bind the `SimpleFactory` object to the namespace.

Listing 5-4 C++ Example of Binding a Name to the BEA Tuxedo Namespace

```
...
//Establish the Name used to identify the SimpleFactory object
//in the namespace.

CosNaming::Name_var         factory_name = new CosNaming::Name(1);
       factory_name->length(1);
       factory_name[(CORBA::ULong) 0].id =
                                   (const char * "simple_factory";
       factory_name[(CORBA::ULong) 0].kind =
                                   (const char *) "";
//Create an object reference for the SimpleFactory object

s_v_factory_refer = TP::create_object_reference(
                                   _tc_SimpleFactory->id(),
                                   "simple_factory",
                                   CORBA::NVList::_nil()
);
//Get the NameService object reference. See Listing 4-2.
//Place the object reference for SimpleFactory in the namespace

root->bind(factory_name, s_v_fact_ref);
...
```

Step 5: Use a Name to Locate an Object in the BEA Tuxedo Namespace

Use the `resolve()` method of the `CosNaming::NamingContext` object to locate an object in a namespace in a BEA Tuxedo domain. Listing 5-5 shows the C++ code that accepts two parameters, representing the `id` and `kind` fields for a `Name`. The code example then binds to a naming context, resolves the name, and obtains an object reference for the specified object.

Listing 5-5 C++ Example of Locating a Name in the BEA Tuxedo Namespace

```
...
//Establish the Name used to identify the SimpleFactory object
//in the namespace.
CosNaming::Name_var         factory_name = new CosNaming::Name(1);
       factory_name->length(1);
       factory_name[(CORBA::ULong) 0].id =
                                   (const char * "simple_factory";
       factory_name[(CORBA::ULong) 0].kind =
                                   (const char *) "";

//Locate the SimpleFactory object in the namespace
CORBA::Object_var v_simple_factory_oref =
              root->resolve( *factory_name);
SimpleFactory_var v_simple_factory_ref =
              SimpleFactory::_narrow(v_simple_factory_oref.in());

// Use the reference obtained from the BEA Tuxedo CORBA Name Service // to
find the Simple object
Simple_var v_simple = v_simple_factory_ref->find_simple();
...
```