MICRO FOCUS

Orbix 3.3.17

OrbixNames Programmer's and Administrator's Guide

Micro Focus
The Lawn
22-30 Old Bath Road
Newbury, Berkshire RG14 1QN
UK

http://www.microfocus.com

2021-03-17

Contents

Part III OrbixNames Java Programmer's Guide

Part IV OrbixNames Administrator's Guide

Part V OrbixNames Programmer's Reference

Part VI Appendices

Preface

OrbixNames is a Micro Focus implementation of the CORBA Naming Service. This service allows you to associate abstract names with CORBA objects and to locate objects using those names.

Audience

This guide is intended for use by application programmers who wish to familiarize themselves with the Naming Service, and OrbixNames in particular. Before reading this guide, you should be familiar with either the C++ or the Java programming language and Orbix application programming.

Organization of this Guide

This guide is divided into the following parts:

Part I "Introduction"

This part introduces the CORBA Naming Service and describes the features of the Naming Service specification.

Part II "OrbixNames C++ Programmer's Guide"

Part II describes how C++ programmers can use OrbixNames to take advantage of the CORBA Naming Service in their applications. It also describes OrbixNames extensions to this service that facilitate the implementation of load balancing in CORBA servers.

Part III "OrbixNames Java Programmer's Guide"

Part III describes how Java programmers can use OrbixNames to take advantage of the CORBA Naming Service in their applications. It also describes OrbixNames extensions to this service that facilitate the implementation of load balancing in CORBA servers.

Part IV "OrbixNames Administrator's Guide"

Part IV describes the OrbixNames command-line utilities and graphical browser. These allow administrators to access the CORBA Naming Service without writing applications.

Part V "OrbixNames Programmer's Reference"

Part V provides a complete reference for the programming interface to OrbixNames, defined in the CORBA Interface Definition Language (IDL).

Part VI "Appendices"

Part VI describes the configuration options available for OrbixNames.

Document Conventions

This guide uses the following typographical conventions:

`Constant width` — Constant width in normal text represents portions of code and literal names of items such as classes, functions, variables, and data structures. For example, text might refer to the `CORBA::Object` class.

Constant width paragraphs represent code examples or information a system displays on screen. For example:

```
#include <stdio.h>
```

Italic — Italic words in normal text represent emphasis and new terms.

Italic words or characters in code and commands represent variable values you must supply, such as arguments to commands or path names for your particular system. For example:

```
% cd /users/your_name
```

This guide may use the following keying conventions:

< >	Some command examples use angle brackets to represent variable values you must supply. This is an older convention.
... . . .	Horizontal or vertical ellipses in format and syntax descriptions indicate that material has been eliminated to simplify the discussion.
[ ]	Brackets enclose optional items in format and syntax descriptions.
{ }	Braces enclose a list from which you must choose an item in format and syntax descriptions.
|	A vertical bar separates items in a list of choices enclosed in { } (braces) in format and syntax descriptions.

Contacting Micro Focus

Our Web site gives up-to-date details of contact numbers and addresses.

Further Information and Product Support

Additional technical information or advice is available from several sources.

The product support pages contain a considerable amount of additional information, such as:

- The WebSync service, where you can download fixes and documentation updates.
- The Knowledge Base, a large collection of product tips and workarounds.
- Examples and Utilities, including demos and additional product documentation.

To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

Note:
Some information may be available only to customers who have maintenance agreements.

If you obtained this product directly from Micro Focus, contact us as described on the Micro Focus Web site, http://www.microfocus.com. If you obtained the product from another source, such as an authorized distributor, contact them for help first. If they are unable to help, contact us.

Information We Need

However you contact us, please try to include the information below, if you have it. The more information you can give, the better Micro Focus SupportLine can help you. But if you don't know all the answers, or you think some are irrelevant to your problem, please give whatever information you have.

- The name and version number of all products that you think might be causing a problem.
- Your computer make and model.
- Your operating system version number and details of any networking software you are using.
- The amount of memory in your computer.
- The relevant page reference or section in the documentation.
- Your serial number. To find out these numbers, look in the subject line and body of your Electronic Product Delivery Notice email that you received from Micro Focus.

Contact information

Our Web site gives up-to-date details of contact numbers and addresses.

Additional technical information or advice is available from several sources.

The product support pages contain considerable additional information, including the WebSync service, where you can download fixes and documentation updates. To connect, enter http://www.microfocus.com in your browser to go to the Micro Focus home page.

If you are a Micro Focus SupportLine customer, please see your SupportLine Handbook for contact information. You can download it from our Web site or order it in printed form from your sales representative. Support from Micro Focus may be available only to customers who have maintenance agreements.

You may want to check these URLs in particular:

- http://www.microfocus.com/products/corba/orbix/orbix-3.aspx (trial software download and Micro Focus Community files)
- https://supportline.microfocus.com/productdoc.aspx. (documentation updates and PDFs)

To subscribe to Micro Focus electronic newsletters, use the online form at:

http://www.microfocus.com/Resources/Newsletters/infocus/newsletter-subscription.asp

Part I

Introduction

In this part

This part contains the following:

Introduction to the CORBA Naming Service

OrbixNames is a Micro Focus implementation of the CORBA Naming Service, a service that allows you to associate abstract names with CORBA objects in your applications. This chapter describes the features of the CORBA Naming Service.

The Naming Service is a standard service for CORBA applications, defined in the Object Management Group's (OMG) CORBAservices specification. The Naming Service allows you to associate abstract names with CORBA objects and allows clients to find those objects by looking up the corresponding names. This service is both very simple and very useful.

A server that holds a CORBA object *binds* a name to the object by contacting the Naming Service. To obtain a reference to the object, a client requests the Naming Service to look up the object associated with a specified name. This is known as *resolving* the object name. The Naming Service provides interfaces defined in IDL that allow servers to bind names to objects and clients to resolve those names.

Most CORBA applications make some use of the Naming Service. Locating a particular object is a common requirement in distributed systems and the Naming Service provides a simple, standard way to do this.

The Interface to the Naming Service

The Naming Service maintains a database of names and the objects associated with them. An association between a name and an object is called a *binding*. The IDL interfaces to the Naming Service provide operations to access the database of bindings. For example, you can create new bindings, resolve names, and delete existing bindings.

OrbixNames is implemented as a normal Orbix server. This server contains objects which support the standard IDL interfaces to the Naming Service. These interfaces are defined in the IDL module `CosNaming`:

```
// IDL
module CosNaming {
    // Naming Service IDL definitions.
    ...
};
```

Part V of this guide provides a full reference for the definitions in this module. The remainder of this chapter provides a brief overview of the most commonly used definitions.

Format of Names in the Naming Service

In the CORBA Naming Service, names can be associated with two types of object: a *naming context* or an application object. A naming context is an object in the Naming Service within which you can resolve the names of other objects.

Naming contexts are organized into a naming graph, which may form a naming hierarchy much like that of a filing system. Using this analogy, a name bound to a naming context would correspond to a directory and a name bound to an application object would correspond to a file.

The full name of an object, including all the associated naming contexts, is known as a *compound name*. The first component of a compound name gives the name of a naming context, in which the second component is accessed. This process continues until the last component of the compound name has been reached.

The notion of a compound name is common in filing systems. For example, in UNIX, compound names take the form `/aaa/bbb/ccc`; in Windows they take the form `C:\aaa\bbb\ccc`. A compound name in the Naming Service takes a more abstract form: an IDL sequence of name components.

Name components are not simple strings. Instead, a name component is defined as an IDL structure, of type `CosNaming::NameComponent`, that holds two strings:

```
// IDL
// In module CosNaming.
typedef string Istring;

struct NameComponent {
    Istring id;
    Istring kind;
};
```

A name is a sequence of these structures:

```
typedef sequence<NameComponent> Name;
```

The `id` member of a `NameComponent` is a simple identifier for the object; the `kind` member is a secondary way to differentiate objects and is intended to be used by the application layer. For example, you could use the `kind` member to distinguish the type of the object being referred to. The semantics you choose for this member are not interpreted by OrbixNames.

Both the `id` and `kind` members of a `NameComponent` are used in name resolution. Two names that differ only in the `kind` member of one `NameComponent` are considered to be different names.

IDL Interfaces to the Naming Service

The IDL module `CosNaming` contains two interfaces that allow your applications to access the Naming Service:

`NamingContext`	Provides the operations that allow you to access the main features of the Naming Service, such as binding and resolving names.
`BindingIterator`	Allows you to read each element in a list of bindings. Such a list may be returned by operations of the `NamingContext` interface.

The remainder of this chapter describes how you use the `NamingContext` interface to do simple Naming Service operations, such as binding names to your application objects and resolving those names in your clients.

Using the Naming Service

The first step in using the Naming Service is to get a reference to the *root naming context*. The root naming context is an object, of type `CosNaming::NamingContext`, which acts as an entry point to all the bindings in the Naming Service.

This section describes some of the operations you can call on the root naming context, or other naming contexts created by you, to do basic Naming Service tasks.

Associating a Name with an Object

The operation `CosNaming::NamingContext::bind()` allows you to bind a name to an object in your application. This operation is defined as:

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

To use this operation, you first create a `CosNaming::Name` structure containing the name you want to bind to your object. You then pass this structure and the corresponding object reference as parameters to `bind()`.

Using Names to Find Objects

Given an abstract name for an object, you can retrieve a reference to the object by calling `CosNaming::NamingContext::resolve()`. This operation is defined as:

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

When you call `resolve()`, the Naming Service retrieves the object reference associated with the specified `CosNaming::Name` value and returns it to your application.

Associating a Compound Name with an Object

Figure 1 shows an example of a simple compound name.

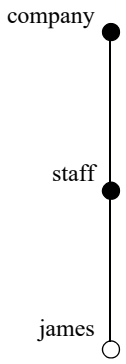

Figure 1 Example of a Compound Name

In this figure, a name with identifier `company` (and no kind value) is bound to a naming context in the Naming Service. This naming context contains one binding: between the name `staff` and another naming context. The `staff` naming context contains a binding between the name `james` and an application object.

If you want to associate a compound name with an object, you must first create the naming contexts that will allow you to build the compound name. For example, to create the compound name shown in Figure 1:

1. Get a reference to the root naming context.
2. Use the root naming context to create a new naming context and bind the name `company` to it. To do this, call the operation `CosNaming::NamingContext::bind_new_context()`, passing the name `company` as a parameter. This operation returns a reference to the newly created naming context.
3. Call `CosNaming::NamingContext::bind_new_context()` on the `company` naming context object, passing the name `staff` as a parameter. This returns a reference to the new `staff` naming context.
4. Call `CosNaming::NamingContext::bind()` on the `staff` naming context, to bind the name `james` to your application object.

The operation `CosNaming::NamingContext::bind_new_context()` is defined as:

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

To create a new naming context and bind a name to it, create a `CosNaming::Name` structure for the context name and pass it to `bind_new_context()`. If the call is successful, the operation returns a reference to your newly created naming context.

Removing Bindings from the Naming Service

If you want to remove the association between a name and an object in the Naming Service, call the operation `CosNaming::NamingContext::unbind()`. This operation is defined as:

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

This operation takes a single parameter that indicates the name to be removed from the Naming Service.

The name passed as a parameter to `unbind()` may be associated with a naming context or an application object. If you unbind the name of a context and your applications have no further use for that context, you should delete the corresponding naming context object. To do this, call `CosNaming::NamingContext::destroy()` on a reference to the naming context. This operation is defined as:

```
void destroy ()
    raises (NotEmpty);
```

Before calling `destroy()` on a naming context object, remove any bindings contained in the context.

Convention for String Format of Names

To make it easier to describe examples, this guide uses a string representation of Naming Service names. This convention is specific to OrbixNames and is illustrated by the following example:

```
documents-dir.reports-dir.april97-txt
```

In this example, the ID value of the first name component is `documents` and the kind value is `dir`. The next component has ID `reports` and kind `dir`, followed by a component with ID `april97` and kind `txt`. This string format is used throughout the rest of this guide and is understood by the OrbixNames utilities described in the chapter "Using the OrbixNames Utilities".

Note

If the dash '–' character is omitted from a name component, the kind field is a zero length string. The forward slash character '/' may be used to escape the characters '–' (dash), '.' (period), and '/' (forward slash).

Part II

OrbixNames C++ Programmer's Guide

In this part

This part contains the following:

C++ Programming with OrbixNames

This chapter describes how you can use OrbixNames to make objects available in CORBA servers and to locate those objects in clients. The examples in this chapter use a C++ programming interface to the Naming Service introduced in the chapter "Introduction to the CORBA Naming Service".

OrbixNames implements the CORBA Naming Service. To develop applications that access the Naming Service, you must use two components of OrbixNames:

- The *OrbixNames IDL files* contain the IDL definitions for the interfaces to the CORBA Naming Service and the load balancing features of OrbixNames.
- The *OrbixNames server* is a normal Orbix server, provided by Micro Focus, that implements the functionality of the CORBA Naming Service.

When you write a CORBA program that uses the Naming Service, this program contacts the OrbixNames server using the OrbixNames IDL definitions. In this way, any CORBA client or server that uses the Naming Service simply acts as a client to the OrbixNames server. The examples in this chapter show how to develop, compile, and run such programs.

Developing an OrbixNames Application

Consider a software engineering company that maintains an administrative database of personnel records which includes details of names, login names, addresses, salaries, and holiday entitlements. These records are used for various administrative purposes, and it is convenient to use the Naming Service to locate an employee record by name. Figure 2 shows part of a naming context graph designed for this purpose.

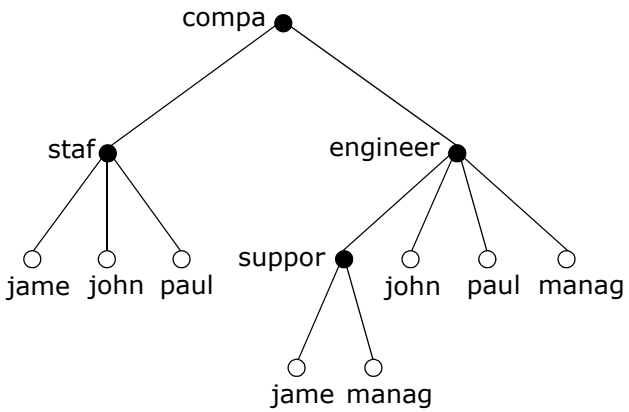

Figure 2 A Naming Context Graph

The nodes `company`, `staff`, `engineering`, and `support` represent naming contexts. A name such as `company.staff.paula-person` names an application object. The same object may have more than one name; for example, each person is listed in the generic `company.staff` context and is also listed in a particular division such as `company.engineering` or `company.sales`.

In addition, it is convenient to use abstract names so that, for example, the engineering manager can be found by looking up the name `company.engineering.manager`.

Allowing different paths to the same object facilitates the many uses that might be made of the Naming Service. For example, a payroll system might be interested only in the `company.staff` context; the engineering manager might want the holiday records for all of the employees with entries in the `company.engineering` context to be written to a spreadsheet, and so on.

The remainder of this section shows some sample code based on the naming context graph in Figure 2. The full source code for this example is available in the directory `demo/naming/staff` of your OrbixNames installation.

Making Initial Contact with the Naming Service

Whether you are writing a client or server application, the first step in communicating with the Naming Service is to obtain a reference to the root naming context. There are two ways for an application to do this:

- The recommended way is to use the CORBA Initialization Service. This approach is fully CORBA compliant. To use the Initialization Service, pass the string `NameService` to the following C++ function call on the ORB:

  ```
  // C++
  // In class CORBA::ORB.
  Object_ptr resolve_initial_references(
                          const char* identifier)
  ```

  The result must be narrowed using the function `CosNaming::NamingContext::_narrow()` to obtain a reference to the naming context.

  The call to `resolve_initial_references()` succeeds if an OrbixNames server is running on the local host or the locator is appropriately configured as described in "Compiling and Running an Application" on page 18.

  The name of the OrbixNames server as registered in the Implementation Repository is assumed to be `NS` by default. To contact an OrbixNames server registered with a different name, the configuration entry `IT_NAMES_SERVER` must identify that name, as described in "Configuring OrbixNames" on page 19.

- The second approach is to read the root naming context IOR from a shared file. To do this, use the `-I` switch to specify a file name when running the OrbixNames server, `NS`:

  ```
  ns -I /sharedIORs/ns.ior
  ```

When you run the server in this way, it stores the root naming context IOR in the specified file. You can use this file later to get the initial naming context:

```
// C++
#include <Naming.hh>
...

char *rootIOR;
CORBA::Object_var objVar;
CORBA::ORB_var orbVar;

// Read the contents of file /sharedIORs/ns.ior
// into the string rootIOR.
...

try {
    orbVar =
        CORBA::ORB_init (argc, argv, "Orbix");
    objVar = orbVar->string_to_object (rootIOR);
}
...
```

The resulting object reference must subsequently be narrowed using the following call:

`CosNaming::NamingContext::_narrow()`.

Once you get a reference to the root naming context, you can look up names in contexts held by the corresponding OrbixNames server. This allows you to obtain a reference to a particular context or to an application object.

Binding Names to Objects

The following sample server code shows how to build the `company` and `company.staff` naming contexts shown in Figure 2 on page 11. It then shows how to bind the name `company.staff.john-person` to the object referenced by the variable `johnVar` (which supports the IDL interface `Person` implemented by class `PersonImpl`).

```
// C++
// An Orbix server.
#include <Naming.hh>
...

int main () {
    Person_var johnVar = new PersonImpl
                                    ("John", "Engineer");
    CORBA::ORB_var orbVar;
    CORBA::Object_var objVar;
    CosNaming::NamingContext_var rootContext,
                    companyContext, staffContext;
    CosNaming::Name_var name;
    ...

    try {
        orbVar =
            CORBA::ORB_init (argc, argv, "Orbix");

        // Find the initial naming context:
```

```
1                           objVar = orbVar->
                             resolve_initial_references("NameService");
                        if (rootContext=CosNaming::
                                    NamingContext::_narrow(objVar)) {
                            // A CosNaming::Name is simply a sequence
                            // of structs.
2                           name = new CosNaming::Name(1);
                            name->length(1);
                            name[0].id =CORBA::string_dup("company");
                            name[0].kind = CORBA::string_dup("company");

                            // (In one step) create a new context, and
                            // bind it relative to the initial
                            // context:
3                            companyContext =
                                   rootContext->bind_new_context(name);

4                         name[0].id = CORBA::string_dup("staff");
                          name[0].kind = CORBA::string_dup("staff");

                            // (In one step) create a new context, and
                            // bind it relative to the company
                            // context:
5                           staffContext =
                            companyContext->bind_new_context(name);

6                          name[0].id = CORBA::string_dup("john");
                           name[0].kind=CORBA::string_dup("person");

                            // Bind name to object johnVar in context
                            // company.staff:
7                            staffContext->bind(name,johnVar);
                        } else { ... }
                            // Deal with failure to _narrow().
                    } // catch clauses not shown here.
                    ...
                }
```

This code is explained as follows:

1. The server calls `CORBA::ORB::resolve_initial_references()` to get a reference to the root naming context.
2. The server creates a `CosNaming::Name` structure that contains a single component with ID `company` and `company` kind value.
3. A call to `bind_new_context()` on the root context binds the newly created name to a new context object. The new context object is directly within the scope of the root naming context.
4. The server modifies the `CosNaming::Name` structure, assigning ID `staff` and an empty kind value to the single name component.
5. The server calls `bind_new_context()` on a reference to the `company` context object created in step 3. The Naming Service creates a new context object and binds the name `company.staff` to it.
6. The server again modifies the `CosNaming::Name` structure, assigning ID `john` and kind `person` to the single name component.

7 A call to `bind()` on the `company.staff` naming context associates the name `company.staff.john-person` with the application object `johnVar`.

The server code builds up a naming graph by creating individual naming contexts and then binding a name to the application object within the scope of those contexts.

Resolving Object Names in Clients

For a client, a typical use of the Naming Service is to find the initial naming context and then to resolve a name to obtain an object reference. The following code sample illustrates this. It finds the object named `company.engineering.manager-person` and then prints the manager's name.

The following IDL definition is assumed:

```
// IDL
interface Person {
    readonly attribute name;
    ...
};
```

The client is written as:

```
// C++
// An Orbix client.
#include <Naming.hh>
...
int main (int argc, char** argv) {
    CosNaming::NamingContext_var rootContext;
    CosNaming::Name_var name;
    Person_var personVar;
    CORBA::Object_var objVar;
    CORBA::ORB_var orbVar;

    try {
        orbVar =
            CORBA::ORB_init (argc, argv, "Orbix");

        // Find the initial naming context:
1           objVar = orbVar->
            resolve_initial_references("NameService");
        if (rootContext = CosNaming::
                  NamingContext::_narrow(objVar)) {

2          name = new CosNaming::Name(3);
           name->length(3);
           name[0].id = CORBA::string_dup("company");
           name[0].kind = CORBA::string_dup("");
           name[1].id = CORBA::string_dup
                                    ("engineering");
           name[1].kind = CORBA::string_dup("");
           name[2].id = CORBA::string_dup("manager");
           name[2].kind = CORBA::string_dup
                                              ("person");

3          objVar = rootContext->resolve(name);
4          if (personVar = Person::_narrow(objVar)) {
                  cout << personVar->name()
                         << " is the engineering manager."
```

```
                        << endl;
                } else { ... }
                    // Deal with failure to _narrow().
            } else { ... }
                    // Deal with failure to _narrow().

        } // catch clauses not shown here.
        ...
    }
```

This code is explained as follows:

1. The client calls `CORBA::ORB::resolve_initial_references()` to get a reference to the root naming context.
2. The client creates a `CosNaming::Name` structure that contains three name components. The client assigns this structure to represent the compound name `company.engineering.manager-person`.
3. A call to `resolve()` on the root naming context returns the object associated with the name `company.engineering.manager-person`. The client resolves the entire compound name with a single call to the Naming Service.
4. The object returned in step 3 is an application object that implements the IDL interface `Person`. The client now narrows the returned object to type `Person`.

Iterating through Context Bindings

The following code sample shows a simple example of using the `BindingIterator` interface to list the bindings in a context. This code lists the bindings in the context `company.staff`:

```
// C++
CosNaming::NamingContext_var rootContext, staffContext;
CosNaming::BindingList_var bList;
CosNaming::BindingIterator_var bIter;
CosNaming::Name_var name;
CORBA::Object_var objVar;
CORBA::ORB_var orbVar;

try {
    orbVar =
        CORBA::ORB_init (argc, argv, "Orbix");

    // Find the initial naming context:
1       objVar = orbVar->
            resolve_initial_references("NameService");
    rootContext =
        CosNaming::NamingContext::_narrow(objVar);
    if (!CORBA::is_nil (rootContext)) {
2        name = new CosNaming::Name(2);
        name->length(2);
        name[0].id = CORBA::string_dup("company");
        name[0].kind = CORBA::string_dup("");
        name[1].id = CORBA::string_dup("staff");
        name[1].kind = CORBA::string_dup("");
3        objVar = rootContext->resolve(name);
```

```
                staffContext = CosNaming::
                    NamingContext::_narrow(objVar);

                if (!CORBA::is_nil (staffContext)) {
                    const CORBA::ULong batchSize = 10;

4               staffContext->list(batchSize,bList,bIter);
                    CORBA::ULong i;
5                    for (i = 0; i < bList.length(); i++) {
                       cout << bList[i].binding_name[0].id
                               << "-";
                       cout << bList[i].binding_name[0].kind
                               << endl;
                    }

                    // If more than batchSize bindings in
                    // context, obtain them using next_n().
6               if ( !CORBA::is_nil(bIter) ) {
                       while(bIter->next_n(batchSize, bList) {
                           for (i=0; i < bList.length(); i++) {
                               cout << bList[i].
                                       binding_name[0].id << "-"
                               cout << bList[i].
                                          binding_name[0].kind
                                    << endl;
                           }
                    }
                } else { ... }
                       // Deal with failure to _narrow().
            } else { ... }
                       // Deal with failure to _narrow().
        } // catch clauses not shown.
```

The information retrieved by this code may be useful to either a client or a server. The functionality of this code is:

1. The application calls `CORBA::ORB::resolve_initial_references()` to get a reference to the root naming context.
2. It then creates a `CosNaming::Name` structure that contains two name components. The client assigns this structure to represent the compound name `company.staff`, which is bound to a naming context.
3. The application calls `resolve()` on the root naming context to obtain a reference to the `company.staff` context object.
4. A call to `list()` on this context object returns a list of at most ten bindings contained in this context.
5. The application examines each element in the list of bindings returned in step 4.
6. If more than ten bindings are available in context `company.staff`, the `CosNaming::BindingIterator` object `bIter` contains all the bindings not returned in step 4. The application calls the operation `next_n()` to retrieve a list of these additional bindings.

For more information about operation `CosNaming::NamingContext::list()`, refer to "CosNaming::NamingContext::list()" on page 125. For more information about the interface `CosNaming::BindingIterator`, refer to "CosNaming::BindingIterator" on page 119.

Finding Unreachable Context Objects

Applications can create naming contexts with no associated name binding. If such an application exits without destroying these contexts, the context objects remain in the Naming Service but are unreachable and cannot be deleted. For example, an application could do this by calling the operation `CosNaming::NamingContext::unbind()` to unbind a context name, without calling `CosNaming::NamingContext::destroy()` to destroy the corresponding context object.

On start-up, OrbixNames automatically creates a naming context to handle this problem. This context is named `lost+found`. If you create a context without binding a name to it, or unbind a context name without destroying the context object, OrbixNames gives the context a special name within the `lost+found` context. The format of this name is as follows:

`NC_`*number time*

The `number` value is a random number assigned by OrbixNames. The `time` value indicates the date and time at which the name was created in the `lost+found` context. The combination of the `number` and `time` values uniquely identifies the naming context in `lost+found`.

Of course, this naming format makes it almost impossible to determine which context in `lost+found` came from which application. However, this is not important because the `lost+found` context simply allows you to ensure that the Bindings Repository does not become cluttered with unreachable context objects. For example, you might want to destroy all contexts in `lost+found` created before a certain date. This is quite straightforward. First, list the contents of `lost+found` using the OrbixNames `lsns` utility and then delete the appropriate contexts using the OrbixNames `rmns` utility. These utilities are described in the chapter "Using the OrbixNames Utilities".

For example, the following command deletes the context object associated with the name `"NC_9Thu Dec 10 11-09-02 GMT+00-00 1998"` in the `lost+found` context:

```
rmns -x lost+found.NC_9Thu Dec 10 11-09-02 GMT+00-00 1998
```

Before you delete a context in `lost+found`, ensure that the context is no longer required by your applications. For example, if an application uses `CosNaming::NamingContext::new_context()` to create a context that it intends to name later, the context is stored temporarily in `lost+found` until the application binds a name to it. You should take care to avoid deleting such contexts. Deleting contexts created before a given date is one way to achieve this.

The `lost+found` context is most useful during application testing, because leaving unreachable contexts in the Naming Service is bad application behavior. When coding your applications, try to ensure that they avoid doing this.

Compiling and Running an Application

This section describes how to build an application that uses OrbixNames, the configuration variables that are required, how to register an OrbixNames server in the Implementation Repository, and the options that are available on the server executable.

The following steps are required to build an application that uses OrbixNames:

1. Generate stub code for the OrbixNames server by passing the OrbixNames IDL file, `NamingService.idl`, through your IDL compiler. Link your application with the client stub code. For example, you can run the Orbix IDL compiler as follows:
   ```
   idl NamingService.idl
   ```
2. This generates three files: `NamingService.hh`, `NamingServiceC.cc`, and `NamingServiceS.cc`. Include the header file `NamingService.hh` in your application code and link your application with the object code for `NamingServiceC.cc`. Discard `NamingServiceS.cc`.
3. If your application uses the load balancing features of OrbixNames, described in the chapter "Load Balancing with OrbixNames Using C++", you must also pass the other OrbixNames IDL file, `LoadBalancing.idl`, through your IDL compiler, for example:
   ```
   idl LoadBalancing.idl
   ```
4. Again, this generates three files: `LoadBalancing.hh`, `LoadBalancingC.cc`, and `LoadBalancingS.cc`. Include the header file `LoadBalancing.hh` in your application code and link your application with the object code for `LoadBalancingC.cc`. Discard `LoadBalancingS.cc`.
5. Register the OrbixNames server in the Implementation Repository as described in "Registering the OrbixNames Server" on page 20.
6. Configure the Orbix locator to make the OrbixNames server known to `CORBA::ORB::resolve_initial_references()`. Assuming that the OrbixNames server is registered in the Implementation Repository with the name `NS` on host `alpha`, this can be achieved by adding the following line to the `Orbix.hosts` or `orbix.hst` file:
   ```
   NS:alpha:
   ```

Configuring OrbixNames

When you install OrbixNames, the configuration file `orbixnames3.cfg` is added to your system, in the OrbixNames `config` directory. This file contains the configuration variables that relate to OrbixNames and it is included in the Orbix configuration file `iona.cfg`, as described in the ***Orbix Administrator's Guide C++ Edition***.

On UNIX, you can set the OrbixNames configuration variables in the `orbixnames3.cfg` configuration file using the Orbix Configuration Explorer described in the ***Orbix Administrator's Guide C++ Edition***. They may also be set as environment variables. On Windows these values are set in either the configuration file or the system registry.

When setting the values of these variables in the file `orbixnames3.cfg`, define each variable in the `OrbixNames` scope, that is `OrbixNames.IT_NAMES_SERVER`, `OrbixNames.IT_NS_HOSTNAME`, `OrbixNames.IT_NAMES_PATH`, and so on.

For a comprehensive description of OrbixNames and common configuration variables, refer to the appendix "Configuration Variables".

Registering the OrbixNames Server

As a normal Orbix server, the OrbixNames server must be registered with the Orbix Implementation Repository.

As usual, the registration can be performed using either the Graphical Server Manager utility or the `putit` utility. The OrbixNames server can also be registered using the `registerns12` utility.

Once registered with the Implementation Repository, the server can be activated by the Orbix daemon or launched manually.

You can terminate the OrbixNames server in the same way as any Orbix server; that is, by using the `killit` utility or the Graphical Server Manager utility.

With `registerns12`, the name service is set up to be started automatically by the Orbix daemon.

```
registerns12
```

For `putit`, a typical usage is to specify the server name and command line which Orbix daemon will use to start the server on demand. For the OrbixNames server, the command line is the full path to the `ns` command. If any arguments to `ns` are required, the entire command must be put in quotes.

```
putit NS /orbix/bin/ns
putit NS "/orbix/bin/ns -I /orbix/names.ior"
```

The `putit` utility has a `-persistent` flag, which means that the server must be manually launched with the `ns` command and will not be automatically started by the daemon.

```
putit NS -persistent
```

Options to the OrbixNames Server

The OrbixNames server executable is named `ns`; it takes the following options:

```
ns [-v] [-r <repository path>] \
    [-I <ns ior file>] [-l] [-h <hashtable size>] \
    [-p <thread pool size>] [-e <cache size>] [-j]
    [-semisecure] [-secure]
```

The options are

`-v`	Outputs version information. Specifying `-v` does not cause the OrbixNames server to run.
`-r`	Specifies the directory to be used as the Bindings Repository. This overrides the value of `IT_NAMES_PATH`, as set in `Orbix.cfg` (or the system registry on Windows).
`-I <ns ior file>`	Specifies a file where the server will store the root context IOR as it starts up.
`-l`	Starts the OrbixNames server in load balancing mode. If you wish to use object groups, you must start the server with this option.

-h <hash table size>	In OrbixNames, each naming context has an associated hash table. A naming context uses this table to store references to bindings the context contains. The `-h` switch allows you to specify the size of this hash table. The default hash table size is `23`. If you expect your naming contexts to contain more than this number of bindings, increase the hash table size to reduce the number of times the hash table resizes. If you expect less than this number, decrease the hash table size to improve performance.
-p <thread pool size>	The OrbixNames server is a multithreaded application. The `-p` switch sets the size of the thread pool used to handle incoming requests. The default value is `10`.
-e <cache size>	The OrbixNames server caches naming contexts in memory to improve performance. The `-e` switch specifies how many contexts should be cached. The default value is `10`.
-j	The OrbixNames server is a Java application. On platforms other than Solaris, you can instruct the server to pass command-line switches directly to the Java interpreter. To do this, use the `-j` switch to the OrbixNames server. For example, to increase the virtual memory used by the interpreter when running OrbixNames, start the server as follows: `ns -j -mx9000000`
-semisecure	The default OrbixNames server possesses no security. This switch forces the server to accept both secure (SSL) and insecure (non-SSL) connections. You will be prompted for a password that should correspond to the SSL certificates referenced in the OrbixNames section of the `orbixssl.cfg` configuration file.
-secure	The default OrbixNames server possesses no security. This switch forces the server to accept Secure Sockets Layer (SSL) connections only. You will be prompted for a password that should correspond to the SSL certificates referenced in the OrbixNames section of the `orbixssl.cfg` configuration file.

Running OrbixNames in a Secure System

OrbixSSL enables you to create Orbix applications that communicate using Secure Sockets Layer (SSL) security. If you run secure applications that use OrbixNames, the OrbixNames server must also communicate using SSL.

When running OrbixNames with OrbixSSL, you must:

1. Configure SSL support in OrbixNames.
2. Write the OrbixNames Interoperable Object Reference (IOR) to a file.
3. Configure clients to read the OrbixNames IOR from a file.
4. Run the OrbixNames server.

5 If required, run the OrbixNames utilities.

This section briefly describes each of these steps. Refer to the OrbixSSL documentation for more information about OrbixSSL and SSL security.

Configuring SSL Support in OrbixNames

As described in the OrbixSSL documentation, the OrbixSSL configuration file, `orbixssl.cfg`, controls how a program uses SSL. To configure the use of SSL in OrbixNames, you must add several configuration values to `orbixssl.cfg`.

Adding SSL Security to OrbixNames

First, you must instruct OrbixNames to use SSL. To do this, add the following text to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_SECURITY_POLICY = "SECURE";
    };
};
```

The configuration variable `OrbixNames.IT_SECURITY_POLICY` can take one of the following values:

`SECURE`	The OrbixNames server accepts only secure communications.
`INSECURE`	The OrbixNames server accepts only insecure communications.
`SEMI_SECURE`	The OrbixNames server accepts both secure and insecure communications.

If you do not set this variable in the configuration file, OrbixNames does not use SSL security. If you set the value to `SECURE`, you must then configure SSL *authentication*.

Configuring SSL Authentication in OrbixNames

SSL authentication allows one SSL program to verify the identity of another. Each authenticated program has an associated *certificate* and a *private ke*y that it uses to prove its identity. Each certificate is signed by a *Certification Authority* (CA) that guarantees that the certificate is valid. By default, only OrbixSSL server programs are authenticated.

To ensure that the OrbixNames server can prove its identity during authentication, you must specify the location of the OrbixNames certificate and private key files in the OrbixSSL configuration file. By default, OrbixNames uses the certificate file `orbix_names` and the private key file `orbix_names.jpk`, both located in the OrbixSSL `certificates/services` directory.

To configure OrbixNames to use these files, add the following settings to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_CERTIFICATE_FILE = "OrbixSSL directory/
            certs/services/orbix_names";
        IT_PRIVATEKEY_FILE = "OrbixSSL directory/
            certs/services/orbix_names.jpk"
```

```
        };
    };
```

Replace the *OrbixSSL directory* value with the actual directory in which OrbixSSL is installed. In a fully secure system, where you do not use the OrbixSSL demonstration certificates, you must change these settings to associate your chosen certificate and private key with OrbixNames.

Adding Client Authentication to OrbixNames

If required, OrbixNames can authenticate programs that connect to it. In this case, the communicating program must have an associated certificate and the certificate must be signed by a trusted CA.

If you want to enable client authentication by OrbixNames, add the following setting to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_AUTHENTICATE_CLIENTS = "TRUE";
    };
};
```

To specify the file that contains the list of trusted CAs, add the following:

```
OrbixNames {
    Server {
        IT_CA_LIST_FILE = "OrbixSSL directory/
            /ca_lists/demo_ca_list_1";
    };
};
```

In a fully secure system, change this setting to your actual certificate list file.

Configuring the SSL Port for the OrbixNames Server

When the OrbixNames server is SSL-enabled, it requires an additional port on which it listens for incoming secure communications. To set this port value, add the following variable to the OrbixNames configuration file:

```
OrbixNames {
    IT_SSL_IIOP_LISTEN_PORT = "portnumber";
};
```

Replace the *portnumber* value with any available port number.

Writing the OrbixNames IOR to a File

Before running the OrbixNames server with OrbixSSL, you must instruct the server to publish its IOR to a file. This IOR includes the SSL tag component which is necessary when making a secure connection. To publish the IOR, use the `-I` switch as follows:

```
ns -I filename
```

This causes the server to write its IOR to the file specified in *filename*.

Configuring Clients to Read the OrbixNames IOR

After the OrbixNames server writes its IOR to a file, you must configure your clients to read this IOR when making contact with the CORBA Naming Service.

For Orbix clients, add the following setting to the OrbixNames configuration file:

```
Common {
    Services {
        NameService = "IOR";
    };
};
```

In this case, *IOR* is the OrbixNames IOR copied from file.

When the client calls `resolve_initial_references()` to obtain a reference to the OrbixNames server, these settings ensure that it uses the correct IOR. The only way that clients can contact a secure OrbixNames server is by using `resolve_initial_references()` in this manner.

Running the OrbixNames Server

To use security with OrbixNames, the OrbixNames server can be started either manually via the `ns` command script or automatically by the daemon after registration with the Orbix Implementation Repository. See "Registering the OrbixNames Server" for more information.

In all cases, either the `-secure` or `-semisecure` switch must be specified. For example:

```
ns -secure
registerns12 -semisecure
putit NS "/orbix/bin/ns -secure"
```

The `-secure` and `-semisecure` switches override the security setting specified by the `OrbixNames.Server.IT_SECURITY_POLICY` variable in `orbixssl.cfg`. If no switch is specified, the OrbixNames server runs in insecure mode.

To gain access to its private key, OrbixNames must supply the pass phrase that was used to encrypt the key. When the server is started, an attempt is made to retrieve the pass phrase from the KDM. If it is not available from the KDM, the user is prompted for the pass phrase.

If you use the OrbixSSL demonstration certificates and private keys, enter the pass phrase `demopassword`. Otherwise, enter the correct pass phrase for the private key specified in the `OrbixNames.Server.IT_PRIVATEKEY_FILE` configuration value in `orbixssl.cfg`.

Running the OrbixNames Utilities

Using a secure OrbixNames server, you can run only the C++ OrbixNames utilities, for example `lsns`. You cannot run the Java utilities. For example, `lsnsj` cannot use SSL security.

If the OrbixNames server uses client authentication, the utilities must be able to supply a certificate and gain access to a private key. During installation, each utility is configured to use the `orbix` demonstration certificate from the OrbixSSL `certificates/services` directory. The ***OrbixSSL Programmer's and Administrator's Guide C++ Edition*** describes how to replace this certificate and update the utilities with a new private key pass phrase.

Federation of Name Spaces

The collection of all valid names recognized by the Naming Service is called a *name space*. A name space is not necessarily located on a single OrbixNames server, because a context in one OrbixNames server can be bound to a context in another OrbixNames server on the same host or on a different host. The name space provided by a Naming Service is the association or *federation* of the name spaces of each individual OrbixNames server that comprises the Naming Service.

Figure 3 shows a Naming Service federation that comprises two OrbixNames servers running on different hosts. In this example, names relating to the company's engineering and PR divisions are served by one server, and names relating to the company's marketing division are served by a separate server. A request to resolve a name starts in one OrbixNames server, but may continue in another server's database. Clients do not have to be aware that more than one server is involved in the resolution of a name, and they do not need to know which server interprets which part of a compound name.

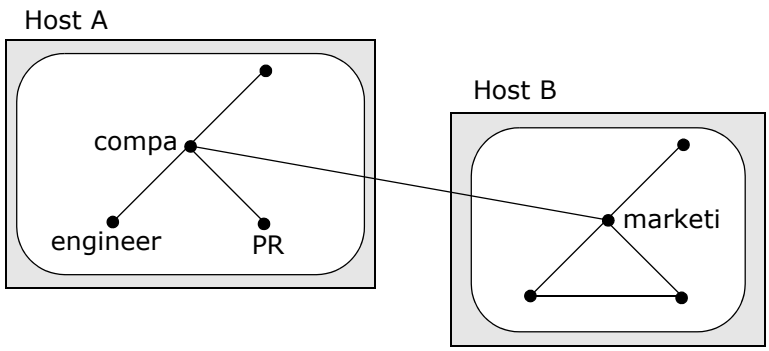

Figure 3 Naming Graph Spanning Two OrbixNames Servers

The following code sample shows how to create the naming context `company` on host `A` and the naming context `marketing`, which is a sub-context of `company`, on host `B`:

```
// C++
#include <Naming.hh>
...
int main (int argc, char** argv) {
    const char* hostA = "A";
    const char* hostB = "B";
    char* ior;
    CORBA::Object_var objVar;
    CosNaming::NamingContext_var hostAContext,
                    hostBContext, companyContext,
```

```
                    marketingContext;
    CosNaming::Name_var name;
    CORBA::ORB_var orbVar;

    try {
        orbVar =
            CORBA::ORB_init (argc, argv, "Orbix");

1       // Read IOR for root context on host B
        // from a file into the string ior.
        // (Not shown.)
        ...
        objVar = orbVar->string_to_object (ior);

        hostBContext =
            CosNaming::NamingContext::_narrow
            (objVar);

2       name = new CosNaming::Name(1);
        name->length(1);
        name[0].id = CORBA::string_dup("marketing");
        name[0].kind = CORBA::string_dup("");
3       marketingContext =
            hostBContext->bind_new_context (name);

4       // Read IOR for root context on host A
        // from a file into the string ior.
        // (Not shown.)
        ...
        objVar = orbVar->string_to_object (ior);

        hostAContext =
            CosNaming::NamingContext::_narrow
            (objVar);

5       name[0].id = CORBA::string_dup("company");
        name[0].kind = CORBA::string_dup("");

6           companyContext =
            hostAContext->bind_new_context (name);

7       name[0].id = CORBA::string_dup("marketing");
        name[0].kind = CORBA::string_dup("");

8       companyContext->bind_context (
            name, marketingContext);
        ...
    } // catch clauses not shown here.
    ...
}
```

This code is explained as follows:

1 The application assumes that the IORs for the root naming contexts on hosts `A` and `B` have been written to files, as described in "Making Initial Contact with the Naming Service" on page 12. The application then obtains a reference to the root naming context associated with the OrbixNames server on host `B`.

2. The application creates a name structure with a single element. This structure represents the name of the `marketing` context on host `B`.
3. A call to `bind_new_context()` creates a new context on host `B` and binds the name `marketing` to it.
4. The application gets a reference to the root naming context associated with the OrbixNames server on host `A`.
5. The application modifies the name structure to contain the name of the `company` context.
6. A call to `bind_new_context()` creates a new context on host `A` and binds the name `company` to it.
7. The application modifies the name structure to contain the name of the `marketing` context, which is a sub-context of `company` on host `A`.
8. The operation `bind_context()`, called on the `company` context, binds the name `company-marketing` to the object reference associated with the `marketing` context on host `B`. If a client contacts the OrbixNames server on host `A` and resolves a name in the `company-marketing` context, the server on host `B` completes the name resolution.

You can also create a federated name space using the OrbixNames utilities. These utilities are described in detail in the chapter "Using the OrbixNames Utilities". To achieve the same result as the code above, first use the `putnewncns` command to create the `company` naming context on host `A` and the `marketing` naming context on host `B`:

```
putnewncns -h A company
putnewncns -h B marketing
```

Next, instruct OrbixNames to copy the object reference for the `marketing` context object to the file `marketing.ior`:

```
catns -h B marketing > marketing.ior
```

Finally, associate the name of this context with the object reference of the `marketing` context on host `B`:

```
putncns -h A company.marketing -f marketing.ior
```

Load Balancing with OrbixNames Using C++

Load balancing is a crucial requirement for many distributed applications. This chapter describes the powerful, but easy-to-use OrbixNames approach to load balancing in CORBA applications.

The Need for Load Balancing

The role of the CORBA Naming Service is critical in large-scale distributed applications. The Naming Service acts as a central repository of objects, which clients use to locate server applications. Administrators can relocate or upgrade server applications by modifying the contents of the Naming Service. This requires no coding modifications on the client side.

Figure 4 shows a typical OrbixNames environment:

- The `Bank` server binds an object `obj1`, to a name `name1`, in the Naming Service.
- `Clients 1...N` resolve this name by obtaining a proxy for `obj1`.
- `Clients 1...N` then invoke `obj1` directly.

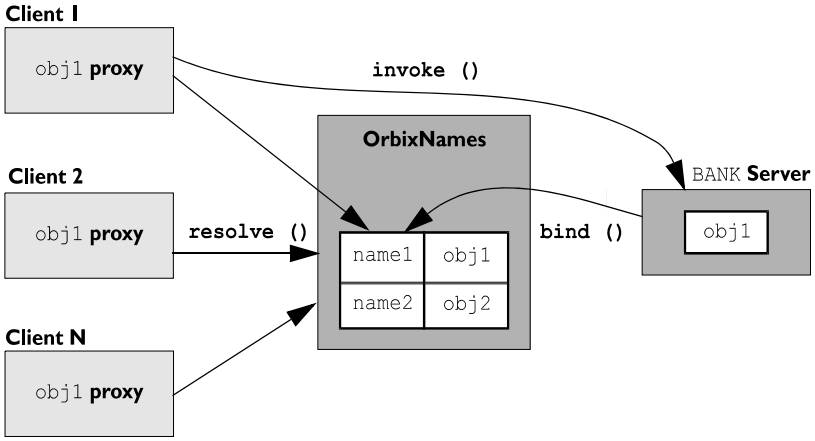

Figure 4 Example of Typical OrbixNames Usage

As the number of deployed clients increases, the load on an individual server may become excessive. To redress this problem, server load balancing through replication may be required.

In the example shown in Figure 4, replication involves creating a new server `Bank_replica`, which contains an object `obj1_replica`. This is an object offering an identical service to `obj1`. The new server registers the replica object in the Naming Service under the name `name1_replica`. Clients can choose to resolve either `name1` or `name1_replica`, to access either `obj1` or `obj1_replica` respectively. This approach is simple and practical, but requires a significant amount of application-specific coding.

Code changes on the client side are especially problematic. For example, if the clients are installed extensively in an enterprise,

each installation will need to be upgraded when clients are modified to select different replica objects. Similarly, if two servers are insufficient, another server `Bank_replica_2` will be required, necessitating further code modifications.

This simple approach to replication does not scale very well because, unlike upgrading or relocating servers, it involves code changes on the client side. However, the Naming Service is a useful candidate for handling server replication and OrbixNames provides a solution to the scalability problem.

Introduction to Load Balancing in OrbixNames

The CORBA Naming Service defines a repository of names that map to objects. A name maps to one object only. OrbixNames extends the CORBA Naming Service model to allow a name to map to a group of objects. An *object group* is a collection of objects that can increase or decrease in size dynamically. For example, `{obj1, obj1_replica, obj1_replica_2}` would constitute an object group.

Each object group has a selection algorithm. This algorithm is applied when a client resolves the name associated with the object group. Two algorithms are supported: round-robin selection and random selection.

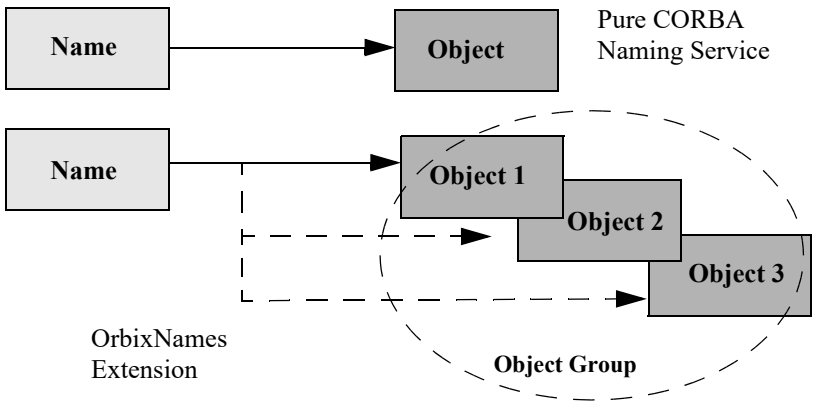

Figure 5 Associating a Name with an Object Group

OrbixNames supports object groups by introducing new IDL interfaces to the Naming Service. These interfaces enable you to create object groups, add objects to and remove objects from groups, and to find out which objects are members of a particular group. If you want to take advantage of object groups, you can use these interfaces in your servers to create and manipulate groups. Your client code can remain unchanged.

To enable load balancing in OrbixNames, the `-l` flag must be used. For example,

```
putit NS "ns -l"
```

Figure 5 illustrates the concept of binding a name to multiple objects using an object group.

The Interface to Object Groups in OrbixNames

The IDL module `LoadBalancing`, defined in the IDL file `LoadBalancing.idl`, provides access to the load balancing features of OrbixNames:

```
module LoadBalancing {
  exception no_such_member{};
  exception duplicate_member{};
  exception duplicate_group{};
  exception no_such_group{};
  typedef string memberId;
  typedef sequence<memberId> memberIdList;
  typedef string groupId;
  typedef sequence<groupId> groupList;

  struct member {
    Object obj;
    memberId id;
  };

  interface ObjectGroup;
  interface RoundRobinObjectGroup;
  interface RandomObjectGroup;

  interface ObjectGroupFactory {
    RoundRobinObjectGroup createRoundRobin(in
groupId id)
                raises (duplicate_group);
    RandomObjectGroup createRandom(in groupId id)
                raises (duplicate_group);
    ObjectGroup findGroup(in groupId id)
                raises (no_such_group);
    groupList rr_groups();
    groupList random_groups();
  };
  interface ObjectGroup {
    readonly attribute string id;

    Object pick();
    void addMember(in member mem) raises
(duplicate_member);
    void removeMember(in memberId id)
                raises (no_such_member);
    Object getMember(in memberId id)
                raises (no_such_member);
    memberIdList members();
    void destroy();
  };
  interface RandomObjectGroup : ObjectGroup {};
  interface RoundRobinObjectGroup : ObjectGroup
{};
};
```

Part IV of this guide provides a complete reference for these definitions.

Using Object Groups in OrbixNames

Because object groups are designed to be transparent to clients, you generally use the `LoadBalancing` module when writing servers. There are four common tasks for which servers use this module:

- Creating a new object group and adding objects to it.
- Adding objects to an existing object group.
- Removing objects from an object group.
- Removing an object group.

The remainder of this section describes how to do each of these operations.

Creating a New Object Group

To create a new object group and add objects to it:

1. Get a reference to a naming context, for example the root naming context.
2. On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.
3. On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::createRandom()` or `LoadBalancing::ObjectGroupFactory::createRoundRobin()` to create an object group that uses the selection algorithm you want. Each of these operations returns a reference to an object that inherits interface `LoadBalancing::ObjectGroup`.
4. Use the operation `LoadBalancing::ObjectGroup::addMember()` to add your application objects to the newly created object group.
5. Use the operation `CosNaming::NamingContext::bind()` to bind a name to the `LoadBalancing::ObjectGroup` object in the usual way.

When creating the object group in step 3, you must specify a *group identifier*. This identifier is a string value unique to that object group.

Similarly, when adding a member to the object group, you must provide a reference to the object and a corresponding *member identifier*. This identifier is a string value that must be unique within the object group.

In both cases, you decide the format of the identifier string. OrbixNames does not interpret these identifiers.

Adding Objects to an Existing Object Group

Before adding objects to an existing object group, you must get a reference to the corresponding `LoadBalancing::ObjectGroup` object. You can do this using the group identifier or the name bound to the object group. This section uses the group identifier.

To add objects to an existing object group:

1. Get a reference to a naming context, for example the root naming context.
2. On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

3 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

4 Use the operation `LoadBalancing::ObjectGroup::addMember()` to add your application objects to the object group.

Removing Objects from an Object Group

Removing an object from a group is quite straightforward if you know the object group identifier and the member identifier for the object:

1 Get a reference to a naming context, for example the root naming context.

2 On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

3 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

4 On the object group, call the operation `LoadBalancing::ObjectGroup::removeMember()` to remove the required object from the group. You must specify the member identifier for the object as a parameter to this operation.

If you already have a reference to the `LoadBalancing::ObjectGroup` object associated with the object group, steps 1 to 3 are unnecessary.

Removing an Object Group

If you do not have a reference to the object group you want to remove, do the following:

1 Get a reference to the root naming context.

2 Use the root naming context to unbind the name associated with the object group, by calling `CosNaming::NamingContext::unbind()` in the usual way.

3 On the root naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

4 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

5 On the object group, call the operation `LoadBalancing::ObjectGroup::destroy()` to remove the group from the Naming Service.

If you already have a reference to the target `LoadBalancing::ObjectGroup` object, steps 3 and 4 are unnecessary.

Finding an Object Group without the Group Identifier

The procedures described in the previous sections assume that your application gets a reference to an object group using the group identifier. You can also get a reference to an object group if you know the name bound to the group in the Naming Service. To do this, call the operation CosNaming::NamingContext::resolve_object_group().

Example of Load Balancing with Object Groups

This section uses sample code to show how you can take advantage of object groups in your CORBA applications. The example described here is a very simple stock market system. In this example, a CORBA object has access to all current stock prices. Clients request stock prices from this CORBA object and display those prices to the user of the application.

In any realistic stock market application, there are potentially many stock prices available and many clients that require price updates without delay. Given such a high processing load, a single CORBA object may not be able to satisfy client requirements. A simple solution to this problem is to replicate the CORBA object, invisibly to the client, using object groups.

Sample code for the application described in this section is available in the `load_balancing` demonstration directory of your OrbixNames installation. This sample code may differ slightly from the code described in this section.

Defining the IDL for the Application

The architecture for the stock market system is shown in Figure 6 on page 34. Two servers process client requests for stock price information. The server `stockmarketserver1` creates two CORBA objects for this purpose. Server `stockmarketserver2` creates an additional CORBA object which, from a client perspective, provides exactly the same service as the objects in `stockmarketserver1`.

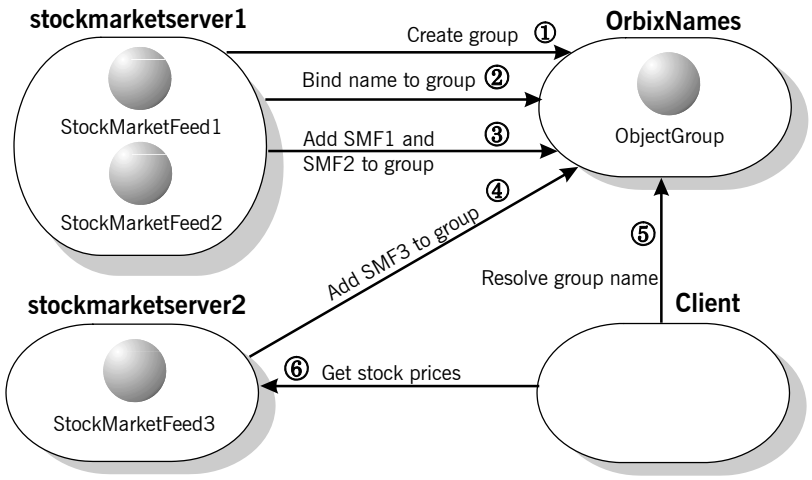

Figure 6 Architecture of the Stock Market Example

The IDL for this application requires only a single interface definition. This interface, called `StockMarketFeed`, is implemented by each of the three CORBA objects.

Interface `StockMarketFeed` is defined in the module `ObjectGroupDemo`:

```
// IDL
module  ObjectGroupDemo {
  interface StockMarketFeed {
    enum feedFailureDetails {
      service_interruption,
stock_feed_terminated};

    exception stock_unavailable {};
    exception stock_feed_failure {
      feedFailureDetails reason;
    };

    long read_stock (in string stock_name)
      raises (stock_unavailable,
stock_feed_failure);
  };
};
```

The interface `StockMarketFeed` includes a single operation, `read_stock()`, which returns the current price of the stock associated with a specified stock name. A name is a string identifier unique to each stock. This operation can raise the following exceptions:

`stock_unavailable`	This exception is raised by `read_stock()` to indicate that the specified stock name is not valid.
`stock_feed_failure`	A `stock_feed_failure` indicates that an error occurred in communications between the server and the source of stock prices.

Creating an Object Group and Adding Objects

After you define your IDL, the next step in developing an application is to implement your interfaces. Using object groups has no effect on how you do this, therefore this section assumes that you have defined a C++ class, `StockMarketFeedImpl`, which implements the interface `StockMarketFeed`.

When you have implemented your IDL interfaces, you must develop a server program that contains and manages your implementation objects. In our application, we have two servers. The first, `stockmarketserver1`, creates two `StockMarketFeed` implementation objects, creates an object group in the Naming Service, and adds the implementation objects to this group. The second server, `stockmarketserver2`, creates an additional `StockMarketFeed` implementation object and adds this to the existing object group.

The source code for the `main()` routine of `stockmarketserver1` is:

```
// C++
#include <stdlib.h>
#include <iostream.h>
```

```
#include "NamingService.hh"
#include "StockMarketFeedImpl.h"
#include "common.h"

int main () {
  CosNaming::NamingContext_var root_context_var;
  LoadBalancing::ObjectGroupFactory_var
ogfactory_var;
  LoadBalancing::ObjectGroup_var object_group_var;
  ObjectGroupDemo::StockMarketFeed_var
stock_market_feed1;
  ObjectGroupDemo::StockMarketFeed_var
stock_market_feed2;
  CORBA::Object_var object_var;

  CORBA::ORB_ptr orb_p;
  CORBA::BOA_ptr boa_p;
  CORBA::ORB_var orb_var;
  CORBA::BOA_var boa_var;

  // Initialize the ORB and BOA.
  orb_var = CORBA::ORB_init (argc, argv, "Orbix");
  boa_var = orb_var->BOA_init (argc, argv,
"Orbix_BOA");
  orb_p = orb_var;
  boa_p = boa_var;

  // Initialize the server name. (Not shown here.)
  ...

  // Create implementation objects.
1  stock_market_feed1 = new StockMarketFeedImpl ();
  stock_market_feed2 = new StockMarketFeedImpl ();

  try  {
    // Get root context.
2    root_context_var = get_root_context ();
    if (CORBA::is_nil (root_context_var))
      return 1;

    // Get object group factory from root context.
3    object_var = root_context_var->OBfactory ();
    ogfactory_var =
    LoadBalancing::ObjectGroupFactory::_narrow
(object_var);

    if (CORBA::is_nil
((LoadBalancing::ObjectGroupFactory_ptr)
      ogfactory_var)) {
      cerr << "Failed to get object group
factory."
           << endl;
      return 1;
    }

    // Create a group and bind a name to it.
    LoadBalancing::groupId_var
sms_group_identifier =
      CORBA::string_dup ("StockMarketServices");
    CORBA::String_var sms_object_group_name =
```

```
        CORBA::string_dup
("stockmarketgroupserver");
    if (!(object_group_var =
4           create_group (ogfactory_var, sms_group_identifier,
      sms_object_group_name, root_context_var)))
      return 1;

    // Add two stock market feed objects to the
group.
5    if (!add_object_to_group (stock_market_feed1,
      "StockMarketFeed1", object_group_var)) {
      cerr << "Failed to add object to group." <<
endl;
      return 1;
    }

    // Add two stock market feed objects to the
group.
    if (!add_object_to_group (stock_market_feed2,
      "StockMarketFeed2", object_group_var)) {
      cerr << "Failed to add object to group." <<
endl;
      return 1;
    }

    // Handle client requests.
6    boa_var->impl_is_ready ("stockmarketserver1");
  }
  catch (CORBA::SystemException &se) {
    cerr << "Unexpected exception:" << endl;
    cerr << &se;
    return 1;
  }
  catch (...) {
    cerr << "Unknown exception." << endl;
    return 1;
  }

  return 0;
}
```

The functionality of this code is as follows:

1. The server creates two implementation objects of type `StockMarketFeedImpl`.
2. The function `get_root_context()` returns a reference to the root naming context in the Naming Service. See "Getting the Root Naming Context" for the implementation of this function.
3. The server calls the operation `OBfactory()` on the root naming context. This operation is implemented by the Naming Service and returns a factory object, of type `LoadBalancing::ObjectGroupFactory`, which the server can use to create object groups.
4. The server calls the function `create_group()`. This function uses the object group factory to create a new group with the specified identifier. It then binds a specified Naming Service name to this group. The implementation of `create_group()` is shown in "Creating an Object Group" on page 40.

5. The function `add_object_to_group()` adds the `StockMarketFeedImpl` objects to the object group created in step 4. The implementation of this function is shown in "Adding an Object to an Object Group" on page 42.
6. Finally, the server prepares to receive client requests by calling `CORBA::BOA::impl_is_ready()` as usual.

Getting the Root Naming Context

The programs in this chapter use the following simple function to get a reference to the root naming context:

```
// C++
#include <stdlib.h>
#include <iostream.h>
#include "NamingService.hh"

CosNaming::NamingContext_ptr get_root_context () {
  CORBA::Object_var object_var;
  CosNaming::NamingContext_ptr root_context_p;
  CORBA::ORB_var orb_var;

  try {
        orb_var =
           CORBA::ORB_init (argc, argv, "Orbix");
    object_var =
      orb_var->resolve_initial_references
("NameService");

    root_context_p =
      CosNaming::NamingContext::_narrow
(object_var);
  }
  catch (CORBA::SystemException &se) {
    cerr << "Unexpected system exception:" <<
endl;
    cerr << &se;
    return CosNaming::NamingContext::_nil ();
  }
  catch (...) {
    cerr << "Unknown exception." << endl;
    return CosNaming::NamingContext::_nil ();
  }

  if (CORBA::is_nil (root_context_p)) {
    cerr << "Narrow to root context failed." <<
endl;
    return CosNaming::NamingContext::_nil ();
  }

  return root_context_p;
}
```

Creating an Object Group

In this example, the server calls the function `create_group()` to create an object group and bind a Naming Service name to it. You can implement this function as follows:

```
// C++
#include <stdlib.h>
#include <iostream.h>
#include "NamingService.hh"
#include "StockMarketFeedImpl.h"
...

LoadBalancing::ObjectGroup_ptr create_group (
  LoadBalancing::ObjectGroupFactory_ptr factory_p,
  LoadBalancing::groupId_var id,
  CORBA::String_var name,
  CosNaming::NamingContext_ptr context_p) {
  LoadBalancing::ObjectGroup_ptr group_p;
  try {
1   group_p = factory_p->createRoundRobin (id);

2     if (!bind_name_to_group (name, group_p, context_p))
        return 0;
  }
  catch (LoadBalancing::duplicate_group& dg) {
    cout << "Group already exists." << endl;

    try  {
      group_p = factory_p->findGroup (id);
    }
    catch (LoadBalancing::no_such_group& nsg) {
      cerr << "Failed to find group." << endl;
      return 0;
    }
  }

  return group_p;
}
```

The function `create_group()` takes four parameters: a reference to the object group factory, a string value used to identify the new group, a string value used to create the name associated with all objects in the group, and a reference to the naming context in which this name should be bound.

The function `create_group()` makes two important calls:

1. It calls the operation `createRoundRobin()` on the object group factory in the Naming Service. This operation returns a new object group in which objects are selected on a round-robin basis.
2. Function `create_group()` then calls `bind_name_to_group()`, a local function that binds a Naming Service name to the newly created group.

Binding a Name to an Object Group

The function `create_group()` calls the function `bind_name_to_group()` to bind a name to the object group. When a client resolves this name, it receives a reference to one of the group's member objects, selected by the Naming Service in accordance with the group selection algorithm. The client does not know that the name is actually bound to a group of objects.

You can code `bind_name_to_group()` as follows:

```
// C++
int bind_name_to_group (
  const char *name_str,
  CORBA::Object_ptr object_p,
  CosNaming::NamingContext_ptr context_p) {
  CosNaming::Name_var group_name = new
CosNaming::Name (2);
  group_name->length (2);

  // Bind name in context LoadBalancingDemo.
  // Assume this context already exists.
  group_name[0].id = CORBA::string_dup
("LoadBalancingDemo");
  group_name[0].kind = CORBA::string_dup ("");
  group_name[1].id = CORBA::string_dup (name_str);
  group_name[1].kind = CORBA::string_dup ("");

  try  {
    context_p->bind (group_name, object_p);
  }
  catch (CosNaming::NamingContext::NotFound) {
    cerr << "NotFound exception." << endl;
    return 0;
  }
  catch (CosNaming::NamingContext::CannotProceed)
{
    cerr << "CannotProceed exception." << endl;
    return 0;
  }
  catch (CosNaming::NamingContext::InvalidName) {
    cerr << "InvalidName exception." << endl;
    return 0;
  }
  catch (CosNaming::NamingContext::AlreadyBound) {
    cerr << "AlreadyBound exception." << endl;
    return 0;
  }
  catch (CORBA::SystemException &se){
    cerr << "Unexpected exception:" << endl;
    cerr << &se << endl;
    return 0;
  }
  return 1;
}
```

The functionality of `bind_name_to_group()` is quite straightforward. This function simply calls `bind()` on a naming context to associate a Naming Service name with an object. In this case, the object's true type is `LoadBalancing::ObjectGroup`, so the name is associated with an object group.

In this example, the object group name is bound in the context `LoadBalancingDemo`. The code assumes that this naming context already exists. For example, you could create this context in the initialization code for `stockmarketserver1`. Alternatively, you could use the OrbixNames `putnewncns` or `putnewncnsj` utilities, described in the chapter "Using the OrbixNames Utilities".

Adding an Object to an Object Group

After creating the object group, `stockmarketserver1` adds its `StockMarketFeed` implementation objects to the group. To do this, the server calls the function `add_object_to_group()`:

```
// C++
#include <stdlib.h>
#include <iostream.h>
#include "NamingService.hh"
#include "StockMarketFeedImpl.h"

int add_object_to_group (
  ObjectGroupDemo::StockMarketFeed_ptr object_p,
  const char* id,
  LoadBalancing::ObjectGroup_ptr objectGroup_p) {

  LoadBalancing::member memberDetails;

  try {
1   memberDetails.obj =
      ObjectGroupDemo::StockMarketFeed::_duplicate
(object_p);
    memberDetails.id = CORBA::string_dup (id);
2    objectGroup_p->addMember (memberDetails);
  }
3  catch (LoadBalancing::duplicate_member& dm)   {
    cerr << "Member with id " << memberDetails.id
      << " already exists." << endl;
    return 0;
  }
  catch (CORBA::SystemException& se) {
    cerr << "Unexpected exception:" << endl;
    cerr << &se << endl;
    return 0;
  }
  return 1;
}
```

The function `add_object_to_group()` takes three parameters: the object to be added to the object group, a string that uniquely identifies the object within the group, and a reference to the object group itself. The member identifier has no effect on the naming of the object within the Naming Service. To obtain a reference to the object, a client resolves the name bound to the object group.

The functionality of `add_object_to_group()` is as follows:

1. The server creates an IDL `struct` of type `LoadBalancing::member` which contains two items: a reference to the `StockMarketFeedImpl` object, and a string that identifies the object within the group.

2 The server adds the new member to the object group in the Naming Service by calling the operation `addMember()` on the corresponding `LoadBalancing::ObjectGroup` object.

3 If the string identifier of the new member clashes with an existing member identifier, the operation `addMember()` throws an exception of type `LoadBalancing::duplicate_member` to indicate this. In this case `addMember()` does not update the contents of the object group in the Naming Service.

Creating Replicated Objects

In this example, the server `stockmarketserver2` replicates the behavior of `stockmarketserver1`. To do this, it creates a new `StockMarketFeed` implementation object that provides the same service to clients as the object in `stockmarketserver1`. It then adds this object to the existing object group, which is associated with the group identifier `StockMarketServices` and the name `LoadBalancingDemo-stockmarketgroupserver` in the Naming Service.

The source code for the `main()` routine of `stockmarketserver2` is:

```
// C++
#include <stdlib.h>
#include <iostream.h>
#include "NamingService.hh"
#include "StockMarketFeedImpl.h"
#include "common.h"

int main () {
  CosNaming::NamingContext_var root_context_var;
  LoadBalancing::ObjectGroup_var group_var;
  CORBA::Object_var object_var;
  CORBA::String_var group_id;
  ObjectGroupDemo::StockMarketFeed_var
feed_object;

  CORBA::ORB_ptr orb_p;
  CORBA::BOA_ptr boa_p;
  CORBA::ORB_var orb_var;
  CORBA::BOA_var boa_var;

  // Initialize the ORB and BOA.
  orb_var = CORBA::ORB_init (argc, argv, "Orbix");
  boa_var = orb_var->BOA_init (argc, argv,
"Orbix_BOA");
  orb_p = orb_var;
  boa_p = boa_var;

  // Initialize the server name. (Not shown here.)
  ...

  group_id = CORBA::string_dup
("ObjectDemoGroup");
  feed_object = new StockMarketFeedImpl ();

  try  {
1   group_var = find_group (group_id);

    if (CORBA::is_nil (group_var)) {
```

```
            cerr << "Failed to get object group." <<
        endl;
            return 1;
          }
          // Add stock market feed object to the group.
2         if (!add_object_to_group (
            feed_object, "StockMarketFeed3", group_var))
        {
            cerr << "Failed to add object to group." <<
        endl;
            return 1;
          }

          // Handle client requests.
3         boa_var->impl_is_ready ("stockmarketserver2");
        }
        catch (CORBA::SystemException &se) {
          cerr << "Unexpected exception:" << endl;
          cerr << &se;
          return 1;
        }
        catch (...) {
          cerr << "Unknown exception." << endl;
          return 1;
        }

        return 0;
      }
```

The functionality of this code is as follows:

1. The server calls the function `find_group()`, which contacts the Naming Service to get a reference to the required object group. This function is described in detail in "Finding an Existing Object Group" on page 44.
2. The server calls `add_object_to_group()` to make the object a member of the existing object group.
3. The server prepares to receive client requests by calling `CORBA::BOA::impl_is_ready()` as usual.

Finding an Existing Object Group

The most important partof `stockmarketserver2` is the function `find_group()`, which retrieves a reference to an existing object group. One way to do this is as follows:

```
// C++
#include <stdlib.h>
#include <iostream.h>
#include "NamingService.hh"
#include "StockMarketFeedImpl.h"
...

LoadBalancing::ObjectGroup_ptr find_group (
  CORBA::String_var group_id) {

  CosNaming::NamingContext_var root_context_var;
  LoadBalancing::ObjectGroupFactory_var
factory_var;
  LoadBalancing::ObjectGroup_var group_var;
```

```
    CORBA::Object_var object_var;

    try  {
      // Get root context.
1        if (!(root_context_var = get_root_context ()))
        return LoadBalancing::ObjectGroup::_nil ();

      // Get object group factory from root context.
2       object_var = root_context_var->OBfactory ();

      factory_var =
      LoadBalancing::ObjectGroupFactory::_narrow
  (object_var);

      if (CORBA::is_nil
  ((LoadBalancing::ObjectGroupFactory_ptr)
      factory_var)) {
      cerr << "Failed to get object group factory."
  << endl;
        return LoadBalancing::ObjectGroup::_nil ();
      }

3       group_var = factory_var->findGroup (group_id);
    }
    catch (LoadBalancing::no_such_group &nsg) {
      cerr << "no_such_group exception." << endl;
      return LoadBalancing::ObjectGroup::_nil ();
    }
    catch (CORBA::SystemException &se) {
      cerr << "Unexpected exception:" << endl;
      cerr << &se;
      return LoadBalancing::ObjectGroup::_nil ();
    }

    return LoadBalancing::ObjectGroup::_duplicate
  (group_var);
  }
```

The functionality of this code is as follows:

1. A call to `get_root_context()` returns a reference to the root naming context.
2. The server calls `OBfactory()` on the root naming context to get a reference to an object group factory.
3. The server calls the operation `findGroup()` on the object group factory. The operation `findGroup()` is defined on the interface `LoadBalancing::ObjectGroupFactory`. Given a group identifier, this operation returns a reference to the corresponding `LoadBalancing::ObjectGroup` object.

Accessing the Objects from a Client

All objects in an object group provide the same service to clients. A client that resolves a name in the Naming Service does not know if the name is bound to an object group or a single object. The client receives a reference to one object only. A client program resolves an object group name in exactly the same way as it resolves a name bound to just one object.

For example, the `main()` routine of the stock market example client could look like this:

```
// C++
#include <iostream.h>
#include <stdlib.h>
#include "ObjectGroupDemo.hh"
#include "NamingService.hh"

int main () {
  CosNaming::NamingContext_var root_context_var;
  ObjectGroupDemo::StockMarketFeed_var feed_var;
  CORBA::Object_var object_var;
  CosNaming::Name_var name;

  // Create name to be resolved.
  name = new CosNaming::Name(2);
  name->length (2);
  name[0].id = CORBA::string_dup
("LoadBalancingDemo");
  name[0].kind = CORBA::string_dup ("");
  name[1].id = CORBA::string_dup
("stockmarketgroupserver");
  name[1].kind = CORBA::string_dup ("");

  try {
    // Get root context.
    root_context_var = get_root_context ();

    // Resolve name.
    object_var = root_context_var->resolve (name);

    if (CORBA::is_nil (object_var)) {
      cerr << "Failed to resolve name." << endl;
      return 1;
    }

    feed_var =
ObjectGroupDemo::StockMarketFeed::_narrow
(object_var);

    // Use stock market feed object. (Not shown.)
    ...
  }

catch (CosNaming::NamingContext::NotFound) {
    cerr << "NotFound exception." << endl;
    return 1;
  }
  catch (CosNaming::NamingContext::CannotProceed)
{
    cerr << "CannotProceed exception." << endl;
    return 1;
  }


  catch (CosNaming::NamingContext::InvalidName) {
    cerr << "InvalidName exception." << endl;
    return 1;
  }
  catch (CORBA::SystemException &se){
```

```
    cerr << "Unexpected exception:" << endl;
    cerr << &se;
    return 1;
  }

  return 0;
}
```

Part III

OrbixNames Java Programmer's Guide

In this part

This part contains the following:

Java Programming with OrbixNames

This chapter describes how you can use OrbixNames to make objects available in CORBA servers and to locate those objects in clients. The examples in this chapter use a Java programming interface to the Naming Service introduced in the chapter "Introduction to the CORBA Naming Service".

OrbixNames implements the CORBA Naming Service. To develop applications that access the Naming Service, you must use two components of OrbixNames:

- The *OrbixNames IDL files* contain the IDL definitions for the interfaces to the CORBA Naming Service and the load balancing features of OrbixNames.
- The *OrbixNames server* is a normal Orbix server, provided by Micro Focus, that implements the functionality of the CORBA Naming Service.

When you write a CORBA program that uses the Naming Service, this program contacts the OrbixNames server using the OrbixNames IDL definitions. In this way, any CORBA client or server that uses the Naming Service simply acts as a client to the OrbixNames server. The examples in this chapter show how to develop, compile, and run such programs.

Developing an OrbixNames Application

Consider a software engineering company that maintains an administrative database of personnel records which includes details of names, login names, addresses, salaries, and holiday entitlements. These records are used for various administrative purposes, and it is convenient to use the Naming Service to locate an employee record by name. Figure 7 shows part of a naming context graph designed for this purpose.

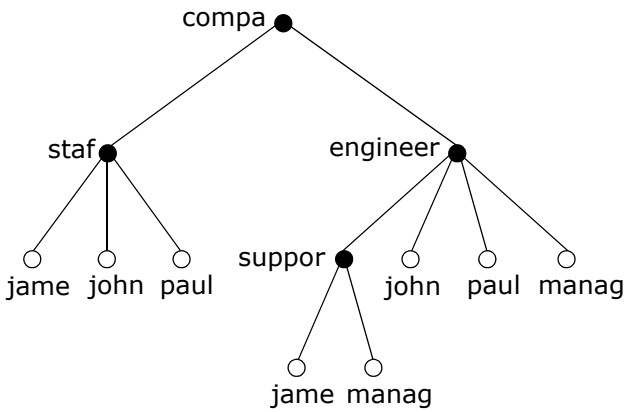

Figure 7 A Naming Context Graph

The nodes `company`, `staff`, `engineering`, and `support` represent naming contexts. A name such as `company.staff.paula-person` names an application object. The same object may have more than one name; for example, each person is listed in the generic `company.staff` context and is also listed in a particular division such as `company.engineering` or `company.sales`.

In addition, it is convenient to use abstract names so that, for example, the engineering manager can be found by looking up the name `company.engineering.manager`.

Allowing different paths to the same object facilitates the many uses that might be made of the Naming Service. For example, a payroll system might be interested only in the `company.staff` context; the engineering manager might want the holiday records for all of the employees with entries in the `company.engineering` context to be written to a spreadsheet, and so on.

The remainder of this section shows some sample code based on the naming context graph in Figure 7. The full source code for this example is available in the directory `demo/naming/staff` of your OrbixNames installation.

Making Initial Contact with the Naming Service

Whether you are writing a client or server application, the first step in communicating with the Naming Service is to obtain a reference to the root naming context. There are two ways for an application to do this:

- The recommended way is to use the CORBA Initialization Service. This approach is fully CORBA compliant. To use the Initialization Service, pass the string `NameService` to the following Java function call on the ORB:

  ```
  // Java
  // In class org.omg.CORBA.ORB
  org.omg.CORBA.Object resolve_initial_references
                              (String identifier)
  ```

  The result must be narrowed using `CosNaming.NamingContextHelper.narrow()` to obtain a reference to the naming context.

  The call to `resolve_initial_references()` succeeds if an OrbixNames server is running on the local host or the locator is appropriately configured as described in "Compiling and Running an Application" on page 59.

  The name of the OrbixNames server as registered in the Implementation Repository is assumed to be `NS` by default. To contact an OrbixNames server registered with a different name, the configuration entry `IT_NAMES_SERVER` must identify that name, as described in "Configuring OrbixNames" on page 60.

- The second approach is to read the root naming context IOR from a shared file. To do this, use the `-I` switch to specify a file name when running the OrbixNames server, `NS`:

  ```
  ns -I /sharedIORs/ns.ior
  ```

When you run the server in this way, it stores the root naming context IOR in the specified file. You can use this file later to get the initial naming context:

```
// Java
import org.omg.CORBA.ORB;
...
String rootIOR;
org.omg.CORBA.Object objRef;

// Read the contents of file /sharedIORs/ns.ior
// into the string rootIOR...
...
try {
    ORB orb = ORB.init(args, null);
    objRef = orb.string_to_object(rootIOR);
}
...
```

The resulting object reference must subsequently be narrowed using the following call:

`CosNaming.NamingContextHelper.narrow()`.

Once you get a reference to the root naming context, you can look up names in contexts held by the corresponding OrbixNames server. This allows you to obtain a reference to a particular context or to an application object.

Binding Names to Objects

The following sample server code shows how to build the `company` and `company.staff` naming contexts shown in Figure 7 on page 51. It then shows how to bind the name `company.staff.john-person` to the object referenced by the variable `johnVar` (which supports the IDL interface `Person` implemented by class `PersonImpl`).

```
// Java
// An OrbixWeb server

import org.omg.CORBA.ORB;
import org.omg.CosNaming.*
...

public class javaserver1 {

    static NamingContext rootContext = null;
    static NamingContext companyContext = null;
    static NamingContext staffContext = null;
        static org.omg.CORBA.ORB orb = null;
        public static void main (String args[]) {

        orb = ORB.init (args,null);
        ...

        // find the initial naming context
        try {
1         org.omg.CORBA.Object initNCRef =
  orb.resolve_initial_references ("NameService");
             rootContext = NamingContextHelper.narrow
                                  (initNCRef);
        }
        catch() {}
```

```
            // catch clause not implemented here

            PersonImplementation john = null;
            PersonImplementation colm = null;
            PersonImplementation john = null;

            try {
                john = new PersonImplementation
                                ("John","Engineer");
            }
            catch() {}
            // catch clause not implemented here

            // A NameComponent[] is an array of structs
2           NameComponent[] name = new NameComponent[1];
            name[1]  = new NameComponent
                                ("company","company");

            // Try to resolve the "company" context
            // in the root context
            try {
                rootContext.resolve (name);
            }
            catch() {}
            // catch clause not implemented here

            ...
            // If company context does not exist, then
            // create a new context.
        // Bind it relative to the initial context
            try {
3             companyContext =
                 rootContext.bind_new_context(name);
            }

        // Modify name, assign "staff"
4       name[1]  = new NameComponent ("staff","staff");
        try {
            // Create a new context, and bind it
            // relative to the initial context
5             staffContext =
                 companyContext.bind_new_context(name);
        }

6       name[1] = new NameComponent ("john","person");

        // Bind name to john object
        // in context company.staff
        try {
7       staffContext.bind (name, john);
        }
            ...
```

This code is explained as follows:

1. The server calls `org.omg.CORBA.Object resolve_initial_references()` to get a reference to the root naming context.

2 The server creates a `NameComponent[]` structure that contains a single component with ID `company` and `company` kind value.

3 A call to `bind_new_context()` on the root context binds the newly created name to a new context object. The new context object is directly within the scope of the root naming context.

4 The server modifies the `NameComponent[]` structure, assigning ID `staff` and a `staff` kind value to the single name component.

5 The server calls `bind_new_context()` on a reference to the `company` context object created in step 3. The Naming Service creates a new context object and binds the name `company.staff` to it.

6 The server again modifies the `NameComponent[]` structure, assigning ID `john` and kind `person` to the single name component.

7 A call to `bind()` on the `company.staff` naming context associates the name `company.staff.john-person` with the application object `john`

The server code builds up a naming graph by creating individual naming contexts and then binding a name to the application object within the scope of those contexts.

Resolving Object Names in Clients

For a client, a typical use of the Naming Service is to find the initial naming context and then to resolve a name to obtain an object reference. The following code sample illustrates this. It finds the object named `company.engineering.manager-person` and then prints the manager's name.

The following IDL definition is assumed:

```
// IDL
interface Person {
    readonly attribute name;
    ...
};
```

The client is written as:

```
// Java
// An OrbixWeb client

import org.omg.CORBA.ORB;
import IE.Iona.OrbixWeb.CosNaming.*;
...

public class javaclient1 {

    static NamingContext rootContext = null;
    static namesStaff.Person personRef = null;
    static org.omg.CORBA.ORB orb = null;

    public static void main( String[] args ) {
        ....
        NamingContext rootContext = null;

        orb = ORB.init (args,null);
```

```
                    // find initial naming context
                    try {
1                     org.omg.CORBA.Object initNCRef =
                  orb.resolve_initial_references ("NameService");
                       rootContext = NamingContextHelper.narrow
                                          (initNCRef);
                    }
                    catch() {}
                    // catch clause not implemented here

2                   NameComponent[] name = new NameComponent[3];
                    org.omg.CORBA.Object objRef = null;

                    name[0] = new NameComponent
                                        ("company","company");
                    name[1] = new NameComponent
                                 ("engineering","engineering");
                    name[2] = new NameComponent
                                        ("manager","person");

3                   objRef = rootContext.resolve (name);

4                   personRef = namesStaff.PersonHelper.narrow
                                  (objRef);
                    // Haven't dealt with failures to narrow()
                    printDetails (personRef);
                    ...
```

This code is explained as follows:

1. The client calls `org.omg.CORBA.Object resolve_initial_references()` to get a reference to the root naming context.
2. The client creates a `NameComponent[]` structure that contains three name components. The client assigns this structure to represent the compound name `company.engineering.manager-person`.
3. A call to `resolve()` on the root naming context returns the object associated with the name `company.engineering.manager-person`. The client resolves the entire compound name with a single call to the Naming Service.
4. The object returned in step 3 is an application object that implements the IDL interface `Person`.The client now narrows the returned object to type `Person`.

Iterating through Context Bindings

The following code sample shows a simple example of using the `BindingIterator` interface to list the bindings in a context. This code lists the bindings in the context `company.staff`:

```
// Java
// Client code extract
// List all the staff context:
...
BindingListHolder bList=new BindingListHolder () ;
BindingIteratorHolder biterHolder
                      = new BindingIteratorHolder ();
```

```
                BindingHolder binding = new BindingHolder ();

1               NameComponent[] name = new NameComponent[2];
                name[0] = new NameComponent
                                        ("Company", "Company");
                name[1] = new NameComponent ("Staff", "Staff");

2               objRef = rootContext.resolve (name);

                staffContext = NamingContextHelper.narrow (objRef);

3               staffContext.list (3,bList,biterHolder);

                System.out.println
                           ("\Contents of staff context:");
                System.out.println
                           ("The length of the list is "
                           + bList.value.length);
4               System.out.println
                          (bList.value[0].binding_name[0].id);
                System.out.println
                           (bList.value[1].binding_name[0].id);
                System.out.println
                           (bList.value[2].binding_name[0].id);
                System.out.println
                           ("\nPrint the remaining objects");

                // print the remaining objects
5               if (biterHolder.value != null ) {
                while ( biterHolder.value.next_one (binding))
                    System.out.println
                           (binding.value.binding_name[0].id);
                ...
```

The information retrieved by this code may be useful to either a client or a server. The functionality of this code is:

1. The application creates a `CosNaming::Name` structure that contains two name components. The client assigns this structure to represent the compound name `company.staff`, which is bound to a naming context.
2. The application calls `resolve()` on the root naming context to obtain a reference to the `company.staff` context object.
3. A call to `list()` on this context object returns a list of at most three bindings contained in this context.
4. The application begins to output each element in the list of bindings returned in step 3.
5. If more than three bindings are available in context `company.staff`, the `BindingIteratorHolder` object `biterHolder` contains all the bindings not returned in step 3. While `biterHolder.value` is not null, the application calls the operation `biterHolder.value.next_one` to retrieve a list of these additional bindings.

For more information about operation `CosNaming::NamingContext::list()`, refer to the section "CosNaming::NamingContext::list()". For more information about

the interface `CosNaming::BindingIterator`, refer to the section "CosNaming::BindingIterator".

Finding Unreachable Context Objects

Applications can create naming contexts with no associated name binding. If such an application exits without destroying these contexts, the context objects remain in the Naming Service but are unreachable and cannot be deleted. For example, an application could do this by calling the operation `CosNaming::NamingContext::unbind()` to unbind a context name, without calling `CosNaming::NamingContext::destroy()` to destroy the corresponding context object.

On start-up, OrbixNames automatically creates a naming context to handle this problem. This context is named `lost+found`. If you create a context without binding a name to it, or unbind a context name without destroying the context object, OrbixNames gives the context a special name within the `lost+found` context. The format of this name is as follows:

`NC_`*number time*

The `number` value is a random number assigned by OrbixNames. The `time` value indicates the date and time at which the name was created in the `lost+found` context. The combination of the `number` and `time` values uniquely identifies the naming context in `lost+found`.

Of course, this naming format makes it almost impossible to determine which context in `lost+found` came from which application. However, this is not important because the `lost+found` context simply allows you to ensure that the Bindings Repository does not become cluttered with unreachable context objects. For example, you might want to destroy all contexts in `lost+found` created before a certain date. This is quite straightforward. First, list the contents of `lost+found` using the OrbixNames `lsns` utility and then delete the appropriate contexts using the OrbixNames `rmns` utility. These utilities are described in the chapter "Using the OrbixNames Utilities".

For example, the following command deletes the context object associated with the name `"NC_9Thu Dec 10 11-09-02 GMT+00-00 1998"` in the `lost+found` context:

```
rmns -x lost+found.NC_9Thu Dec 10 11-09-02 GMT+00-00 1998
```

Before you delete a context in `lost+found`, ensure that the context is no longer required by your applications. For example, if an application uses `CosNaming::NamingContext::new_context()` to create a context that it intends to name later, the context is stored temporarily in `lost+found` until the application binds a name to it. You should take care to avoid deleting such contexts. Deleting contexts created before a given date is one way to achieve this.

The `lost+found` context is most useful during application testing, because leaving unreachable contexts in the Naming Service is bad application behavior. When coding your applications, try to ensure that they avoid doing this.

Compiling and Running an Application

This section describes how to build an application that uses OrbixNames, the configuration variables that are required, how to register an OrbixNames server in the Implementation Repository, and the options that are available on the server executable.

The following steps are required to build an application that uses OrbixNames:

1. Generate stub code for the OrbixNames server by passing the OrbixNames IDL file, `NamingService.idl`, through your IDL compiler. Link your application with the client stub code. For example, you can run the Orbix IDL compiler as follows:
   ```
   idl NamingService.idl
   ```
2. This generates several Java constructs that implement Java classes and interfaces to serve specific roles. You may choose to use either the TIE or the ImplBase approach. For further details, refer to the chapter *"IDL to Java Mapping"* in the ***Orbix Programmer's Guide Java Edition***.
3. If your application uses the load balancing features of OrbixNames, described in the chapter "Load Balancing with OrbixNames Using Java", you must also pass the other OrbixNames IDL file, `LoadBalancing.idl`, through your IDL compiler, for example:
   ```
   idl LoadBalancing.idl
   ```
4. Again, this generates several Java constructs for use during application implementation. Refer to *"IDL to Java Mapping"* in the ***Orbix Programmer's Guide Java Edition*** for further information.
5. Register the OrbixNames server in the Implementation Repository as described in "Registering the OrbixNames Server" on page 60.
6. Configure the Orbix locator to make the OrbixNames server known to `org.omg.CORBA.Object resolve_initial_references()`. Assuming that the OrbixNames server is registered in the Implementation Repository with the name `NS` on host `alpha`, this can be achieved by adding the following line to the `Orbix.hosts` or `orbix.hst` file:
   ```
   NS:alpha:
   ```

Compiling and Running the Demo Application

This section outlines how to build a demonstration program that uses the Naming Service. It describes what configuration variables are required, how to register a naming server in the Implementation Repository and what options are available on the naming server executable.

Building the Naming Service Demonstration Application

The Naming Service demonstration program is located in the `\demos\OrbixNames\staff` directory of your Orbix installation.

Use the following steps for running the demonstration application:

1. To build the application on UNIX platforms use `gmake`; on Windows run the `compile.bat` batch program.
2. Register the Naming Service by entering the following command:

```
putit -j NS -jdk2 --
-Xbootclasspath/p:/opt/microfocus/orbix33/lib/
OrbixNames.jar:/opt/microfocus/orbix33/lib/OrbixWeb.jar
"IE.Iona.OrbixWeb.CosNaming.NS"
```

3. Register the `Staff` server by entering the following command:

```
putit -j Staff namesStaff.javaserver1
```

4. Start the Java server by running the `javaserver1` script on Solaris or `javaserver1.bat` on Windows. This launches the Naming Service and populates it with names.
5. Start the Java client by running the `javaclient1` script on Solaris or `javaclient1.bat` on platforms. This establishes a connection with the Naming Service and resolves the names bound by the Java server.

Note

The `-Xbootclasspath` flag is used to prevent the jre from reading the CORBA Naming Service provided with the jre. The `-jdk2` flag is only required if an `ORB.properties` file has not been added to the jre.

Configuring OrbixNames

When you install OrbixNames, the configuration file `orbixnames3.cfg` is added to your system, in the OrbixNames `config` directory. This file contains the configuration variables that relate to OrbixNames and it is included in the Orbix configuration file `iona.cfg`, as described in the ***Orbix Administrator's Guide Java Edition***.

On UNIX, you can set the OrbixNames configuration variables in the `orbixnames3.cfg` configuration file using the Orbix Configuration Explorer described in the ***Orbix Administrator's Guide Java Edition***. They may also be set as environment variables. On Windows these values are set in either the configuration file or the system registry.

When setting the values of these variables in the file `orbixnames3.cfg`, define each variable in the `OrbixNames` scope, that is `OrbixNames.IT_NAMES_SERVER`, `OrbixNames.IT_NS_HOSTNAME`, `OrbixNames.IT_NAMES_PATH`, and so on.

For a comprehensive description of OrbixNames and common configuration variables, refer to the appendix "Configuration Variables".

Registering the OrbixNames Server

As a normal Orbix server, the OrbixNames server must be registered with the Orbix Implementation Repository.

As usual, the registration can be performed using either the Graphical Server Manager utility or the `putitj` utility. The OrbixNames server can also be registered using the `registerns12` utility.

Once registered with the Implementation Repository, the server can be activated by the Orbix daemon or launched manually.

You can terminate the OrbixNames server in the same way as any Orbix server; that is, by using the `killitj` utility or the Graphical Server Manager utility.

With `registerns12`, the name service is set up to be started automatically by the Orbix daemon.

```
registerns12
```

For `putitj`, a typical usage is to specify the server name and command line which Orbix daemon will use to start the server on demand. For the OrbixNames server, the command line is the full path to the `ns` command. If any arguments to `ns` are required, the entire command must be put in quotes.

```
putitj NS /orbix/bin/ns
putitj NS "/orbix/bin/ns -I /orbix/names.ior"
```

The `putitj` utility has a `-persistent` flag, which means that the server must be manually launched with the `ns` command and will not be automatically started by the daemon.

```
putitj NS -persistent
```

Options to the OrbixNames Server

The OrbixNames server executable is named `ns`; it takes the following options:

```
ns [-v] [-r <repository path>] \
    [-I <ns ior file>] [-l] [-h <hashtable size>] \
    [-p <thread pool size>] [-e <cache size>] [-j]
    [-semisecure] [-secure]
```

The options are

`-v`	Outputs version information. Specifying `-v` does not cause the OrbixNames server to run.
`-r`	Specifies the directory to be used as the Bindings Repository. This overrides the value of `IT_NAMES_PATH`, as set in `Orbix.cfg` (or the system registry on Windows).
`-I <ns ior file>`	Specifies a file where the server will store the root context IOR as it starts up.
`-l`	Starts the OrbixNames server in load balancing mode. If you wish to use object groups, you must start the server with this option.
-h <hash table size>	In OrbixNames, each naming context has an associated hash table. A naming context uses this table to store references to bindings the context contains. The `-h` switch allows you to specify the size of this hash table. The default hash table size is `23`. If you expect your naming contexts to contain more than this number of bindings, increase the hash table size to reduce the number of times the hash table resizes. If you expect less than this number, decrease the hash table size to improve performance.

-p <thread pool size>	The OrbixNames server is a multithreaded application. The `-p` switch sets the size of the thread pool used to handle incoming requests. The default value is `10`.
-e <cache size>	The OrbixNames server caches naming contexts in memory to improve performance. The `-e` switch specifies how many contexts should be cached. The default value is `10`.
-j	The OrbixNames server is a Java application. On platforms other than Solaris, you can instruct the server to pass command-line switches directly to the Java interpreter. To do this, use the `-j` switch to the OrbixNames server. For example, to increase the virtual memory used by the interpreter when running OrbixNames, start the server as follows: `ns -j -mx9000000`
-semisecure	The default OrbixNames server possesses no security. This switch forces the server to accept both secure (SSL) and insecure (non-SSL) connections. You will be prompted for a password that should correspond to the SSL certificates referenced in the OrbixNames section of the `orbixssl.cfg` configuration file.
-secure	The default OrbixNames server possesses no security. This switch forces the server to accept Secure Sockets Layer (SSL) connections only. You will be prompted for a password that should correspond to the SSL certificates referenced in the OrbixNames section of the `orbixssl.cfg` configuration file.

Running OrbixNames in a Secure System

OrbixSSL enables you to create Orbix applications that communicate using Secure Sockets Layer (SSL) security. If you run secure applications that use OrbixNames, the OrbixNames server must also communicate using SSL.

When running OrbixNames with OrbixSSL, you must:

1. Configure SSL support in OrbixNames.
2. Write the OrbixNames Interoperable Object Reference (IOR) to a file.
3. Configure clients to read the OrbixNames IOR from a file.
4. Run the OrbixNames server.
5. If required, run the OrbixNames utilities.

This section briefly describes each of these steps. Refer to the OrbixSSL documentation for more information about OrbixSSL and SSL security.

Configuring SSL Support in OrbixNames

As described in the OrbixSSL documentation, the OrbixSSL configuration file, `orbixssl.cfg`, controls how a program uses SSL.

To configure the use of SSL in OrbixNames, you must add several configuration values to `orbixssl.cfg`.

Adding SSL Security to OrbixNames

First, you must instruct OrbixNames to use SSL. To do this, add the following text to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_SECURITY_POLICY = "SECURE";
    };
};
```

The configuration variable `OrbixNames.IT_SECURITY_POLICY` can take one of the following values:

`SECURE`	The OrbixNames server accepts only secure communications.
`INSECURE`	The OrbixNames server accepts only insecure communications.
`SEMI_SECURE`	The OrbixNames server accepts both secure and insecure communications.

If you do not set this variable in the configuration file, OrbixNames does not use SSL security. If you set the value to `SECURE`, you must then configure SSL *authentication*.

Configuring SSL Authentication in OrbixNames

SSL authentication allows one SSL program to verify the identity of another. Each authenticated program has an associated *certificate* and a *private ke*y that it uses to prove its identity. Each certificate is signed by a *Certification Authority* (CA) that guarantees that the certificate is valid. By default, only OrbixSSL server programs are authenticated.

To ensure that the OrbixNames server can prove its identity during authentication, you must specify the location of the OrbixNames certificate and private key files in the OrbixSSL configuration file. By default, OrbixNames uses the certificate file `orbix_names` and the private key file `orbix_names.jpk`, both located in the OrbixSSL `certificates/services` directory.

To configure OrbixNames to use these files, add the following settings to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_CERTIFICATE_FILE = "OrbixSSL directory/
            certs/services/orbix_names";
        IT_PRIVATEKEY_FILE = "OrbixSSL directory/
            certs/services/orbix_names.jpk"
    };
};
```

Replace the *OrbixSSL directory* value with the actual directory in which OrbixSSL is installed. In a fully secure system, where you do not use the OrbixSSL demonstration certificates, you must change these settings to associate your chosen certificate and private key with OrbixNames.

Adding Client Authentication to OrbixNames

If required, OrbixNames can authenticate programs that connect to it. In this case, the communicating program must have an associated certificate and the certificate must be signed by a trusted CA.

If you want to enable client authentication by OrbixNames, add the following setting to the OrbixSSL configuration file:

```
OrbixNames {
    Server {
        IT_AUTHENTICATE_CLIENTS = "TRUE";
    };
};
```

To specify the file that contains the list of trusted CAs, add the following:

```
OrbixNames {
    Server {
        IT_CA_LIST_FILE = "OrbixSSL directory/
            /ca_lists/demo_ca_list_1";
    };
};
```

In a fully secure system, change this setting to your actual certificate list file.

Configuring the SSL Port for the OrbixNames Server

When the OrbixNames server is SSL-enabled, it requires an additional port on which it listens for incoming secure communications. To set this port value, add the following variable to the OrbixNames configuration file:

```
OrbixNames {
    IT_SSL_IIOP_LISTEN_PORT = "portnumber";
};
```

Replace the *portnumber* value with any available port number.

Writing the OrbixNames IOR to a File

Before running the OrbixNames server with OrbixSSL, you must instruct the server to publish its IOR to a file. This IOR includes the SSL tag component which is necessary when making a secure connection for a client. To publish the IOR, use the `-I` switch as follows:

```
ns -I filename
```

This causes the server to write its IOR to the file specified in *filename*.

Configuring Clients to Read the OrbixNames IOR

After the OrbixNames server writes its IOR to a file, you must configure your clients to read this IOR when making contact with the CORBA Naming Service.

For Orbix clients, add the following setting to the OrbixNames configuration file:

```
Common {
    Services {
```

```
            NameService = "IOR";
        };
    };
```

In this case, *IOR* is the OrbixNames IOR copied from file.

Running the OrbixNames Server

To use security with OrbixNames, the OrbixNames server can be started either manually via the `ns` command script or automatically by the daemon after registration with the Orbix Implementation Repository. See "Registering the OrbixNames Server" for more information.

In all cases, either the `-secure` or `-semisecure` switch must be specified. For example:

```
ns -secure
registerns12 -semisecure
putitj NS "/orbix/bin/ns -secure"
```

The `-secure` and `-semisecure` switches override the security setting specified by the `OrbixNames.Server.IT_SECURITY_POLICY` variable in `orbixssl.cfg`. If no switch is specified, the OrbixNames server runs in insecure mode.

To gain access to its private key, OrbixNames must supply the pass phrase that was used to encrypt the key. When the server is started, an attempt is made to retrieve the pass phrase from the KDM. If it is not available from the KDM, the user is prompted for the pass phrase.

If you use the OrbixSSL demonstration certificates and private keys, enter the pass phrase `demopassword`. Otherwise, enter the correct pass phrase for the private key specified in the `OrbixNames.Server.IT_PRIVATEKEY_FILE` configuration value in `orbixssl.cfg`.

Running the OrbixNames Utilities

Using a secure OrbixNames server, you can run only the C++ OrbixNames utilities, for example `lsns`. You cannot run the Java utilities. For example, `lsnsj` cannot use SSL security.

If the OrbixNames server uses client authentication, the utilities must be able to supply a certificate and gain access to a private key. During installation, each utility is configured to use the `orbix` demonstration certificate from the OrbixSSL `certificates/services` directory. The ***OrbixSSL Programmer's and Administrator's Guide Java Edition*** describes how to replace this certificate and update the utilities with a new private key pass phrase.

Federation of Name Spaces

The collection of all valid names recognized by the Naming Service is called a *name space*. A name space is not necessarily located on a single OrbixNames server, because a context in one OrbixNames server can be bound to a context in another OrbixNames server on the same host or on a different host. The name space provided by a Naming Service is the association or *federation* of the name

spaces of each individual OrbixNames server that comprises the Naming Service.

Figure 8 shows a Naming Service federation that comprises two OrbixNames servers running on different hosts. In this example, names relating to the company's engineering and PR divisions are served by one server, and names relating to the company's marketing division are served by a separate server. A request to resolve a name starts in one OrbixNames server, but may continue in another server's database. Clients do not have to be aware that more than one server is involved in the resolution of a name, and they do not need to know which server interprets which part of a compound name.

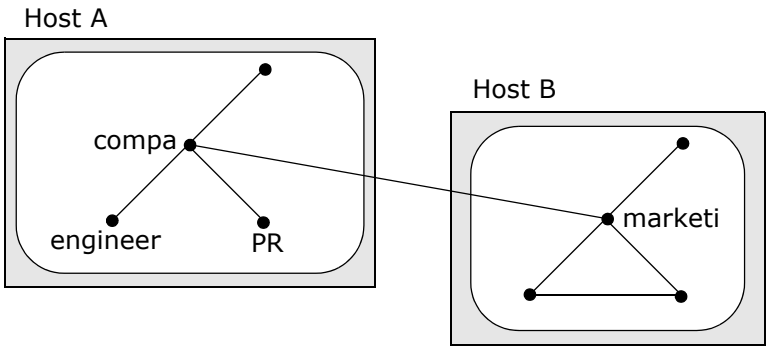

Figure 8 Naming Graph Spanning Two OrbixNames Servers

You can create a federated name space using the OrbixNames utilities. These utilities are described in detail in the chapter "Using the OrbixNames Utilities".

To implement the Figure 8 federated namespace, use the `putnewncns` command to create the `company` naming context on host `A` and the `marketing` naming context on host `B`:

```
putnewncnsj -h A company
putnewncnsj -h B marketing
```

Next, instruct OrbixNames to copy the object reference for the `marketing` context object to the file `marketing.ior`:

```
catnsj -h B marketing > marketing.ior
```

Finally, associate the name of this context with the object reference of the `marketing` context on host `B`:

```
putncns -h A company.marketing -f marketing.ior
```

Load Balancing with OrbixNames Using Java

Load balancing is a crucial requirement for many distributed applications. This chapter describes the powerful, but easy-to-use OrbixNames approach to load balancing in CORBA applications.

The Need for Load Balancing

The role of the CORBA Naming Service is critical in large-scale distributed applications. The Naming Service acts as a central repository of objects, which clients use to locate server applications. Administrators can relocate or upgrade server applications by modifying the contents of the Naming Service. This requires no coding modifications on the client side.

Figure 9 on page 67 shows a typical OrbixNames environment:

- The `Bank` server binds an object `obj1`, to a name `name1`, in the Naming Service.
- `Clients 1...N` resolve this name by obtaining a proxy for `obj1`.
- `Clients 1...N` then invoke `obj1` directly.

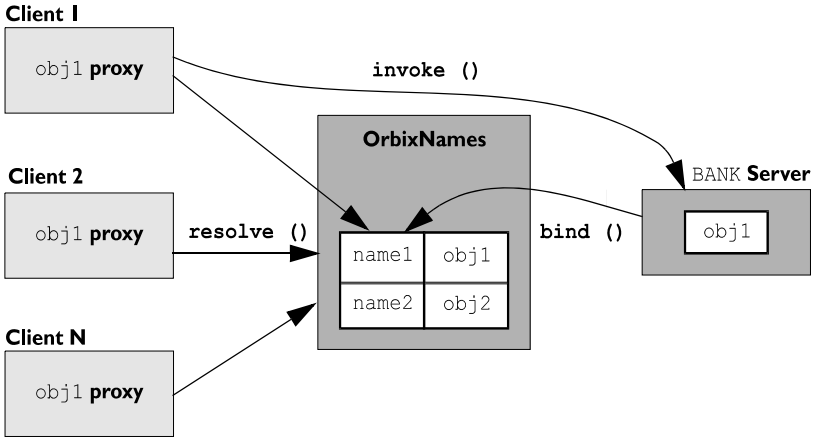

Figure 9 Example of Typical OrbixNames Usage

As the number of deployed clients increases, the load on an individual server may become excessive. To redress this problem, server load balancing through replication may be required.

In the example shown in Figure 9, replication involves creating a new server `Bank_replica`, which contains an object `obj1_replica`. This is an object offering an identical service to `obj1`. The new server registers the replica object in the Naming Service under the name `name1_replica`. Clients can choose to resolve either `name1` or `name1_replica`, to access either `obj1` or `obj1_replica` respectively. This approach is simple and practical, but requires a significant amount of application-specific coding.

Code changes on the client side are especially problematic. For example, if the clients are installed extensively in an enterprise,

each installation will need to be upgraded when clients are modified to select different replica objects. Similarly, if two servers are insufficient, another server `Bank_replica_2` will be required, necessitating further code modifications.

This simple approach to replication does not scale very well because, unlike upgrading or relocating servers, it involves code changes on the client side. However, the Naming Service is a useful candidate for handling server replication and OrbixNames provides a solution to the scalability problem.

Introduction to Load Balancing in OrbixNames

The CORBA Naming Service defines a repository of names that map to objects. A name maps to one object only. OrbixNames extends the CORBA Naming Service model to allow a name to map to a group of objects. An *object group* is a collection of objects that can increase or decrease in size dynamically. For example, `{obj1, obj1_replica, obj1_replica_2}` would constitute an object group.

Each object group has a selection algorithm. This algorithm is applied when a client resolves the name associated with the object group. Two algorithms are supported: round-robin selection and random selection.

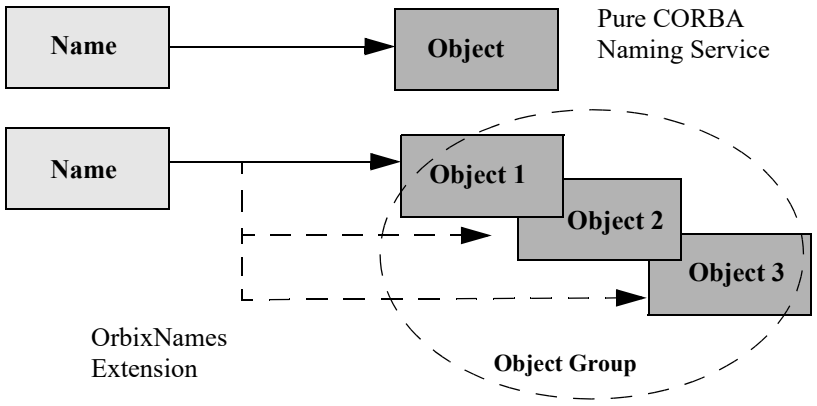

Figure 10 Associating a Name with an Object Group

OrbixNames supports object groups by introducing new IDL interfaces to the Naming Service. These interfaces enable you to create object groups, add objects to and remove objects from groups, and to find out which objects are members of a particular group. If you want to take advantage of object groups, you can use these interfaces in your servers to create and manipulate groups. Your client code can remain unchanged.

To enable load balancing in OrbixNames, the `-l` flag must be used. For example:

```
putit -j NS -jdk2 -- -Xbootclasspath /p:/
opt/microfocus/orbix33/lib/OrbixNames.jar:/opt/microfoc
us/orbix33/lib/
OrbixWeb.jar/: "IE.Iona.OrbixWeb.CosNaming.NS -l"
```

Alternatively, you can use the ns script, for example,

```
putit NS "$ORBIX_HOME/bin/ns -l"
```

Figure 10 illustrates the concept of binding a name to multiple objects using an object group.

The Interface to Object Groups in OrbixNames

The IDL module `LoadBalancing`, defined in the IDL file `LoadBalancing.idl`, provides access to the load balancing features of OrbixNames:

```
module LoadBalancing {
  exception no_such_member{};
  exception duplicate_member{};
  exception duplicate_group{};
  exception no_such_group{};
  typedef string memberId;
  typedef sequence<memberId> memberIdList;
  typedef string groupId;
  typedef sequence<groupId> groupList;

  struct member {
    Object obj;
    memberId id;
  };

  interface ObjectGroup;
  interface RoundRobinObjectGroup;
  interface RandomObjectGroup;

  interface ObjectGroupFactory {
    RoundRobinObjectGroup createRoundRobin(in
groupId id)
                raises (duplicate_group);
    RandomObjectGroup createRandom(in groupId id)
                raises (duplicate_group);
    ObjectGroup findGroup(in groupId id) raises
                (no_such_group);
    groupList rr_groups();
    groupList random_groups();
  };
  interface ObjectGroup {
    readonly attribute string id;

    Object pick();
    void addMember(in member mem) raises
(duplicate_member);
    void removeMember(in memberId id) raises
        (no_such_member);
    Object getMember(in memberId id) raises
        (no_such_member);
    memberIdList members();
    void destroy();
  };
  interface RandomObjectGroup : ObjectGroup {};
  interface RoundRobinObjectGroup : ObjectGroup
{};
};
```

Part IV of this guide provides a complete reference for these definitions.

Using Object Groups in OrbixNames

Because object groups are designed to be transparent to clients, you generally use the `LoadBalancing` module when writing servers. There are four common tasks for which servers use this module:

- Creating a new object group and adding objects to it.
- Adding objects to an existing object group.
- Removing objects from an object group.
- Removing an object group.

The remainder of this section describes how to do each of these operations.

Creating a New Object Group

To create a new object group and add objects to it:

1. Get a reference to a naming context, for example the root naming context.
2. On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.
3. On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::createRandom()` or `LoadBalancing::ObjectGroupFactory::createRoundRobin()` to create an object group that uses the selection algorithm you want. Each of these operations returns a reference to an object that inherits interface `LoadBalancing::ObjectGroup`.
4. Use the operation `LoadBalancing::ObjectGroup::addMember()` to add your application objects to the newly created object group.
5. Use the operation `CosNaming::NamingContext::bind()` to bind a name to the `LoadBalancing::ObjectGroup` object in the usual way.

When creating the object group in step 3, you must specify a *group identifier*. This identifier is a string value unique to that object group.

Similarly, when adding a member to the object group, you must provide a reference to the object and a corresponding *member identifier*. This identifier is a string value that must be unique within the object group.

In both cases, you decide the format of the identifier string. OrbixNames does not interpret these identifiers.

Adding Objects to an Existing Object Group

Before adding objects to an existing object group, you must get a reference to the corresponding `LoadBalancing::ObjectGroup` object. You can do this using the group identifier or the name bound to the object group. This section uses the group identifier.

To add objects to an existing object group:

1. Get a reference to a naming context, for example the root naming context.
2. On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

3 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

4 Use the operation `LoadBalancing::ObjectGroup::addMember()` to add your application objects to the object group.

Removing Objects from an Object Group

Removing an object from a group is quite straightforward if you know the object group identifier and the member identifier for the object:

1 Get a reference to a naming context, for example the root naming context.

2 On the naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

3 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

4 On the object group, call the operation `LoadBalancing::ObjectGroup::removeMember()` to remove the required object from the group. You must specify the member identifier for the object as a parameter to this operation.

If you already have a reference to the `LoadBalancing::ObjectGroup` object associated with the object group, steps 1 to 3 are unnecessary.

Removing an Object Group

If you do not have a reference to the object group you want to remove, do the following:

1 Get a reference to the root naming context.

2 Use the root naming context to unbind the name associated with the object group, by calling `CosNaming::NamingContext::unbind()` in the usual way.

3 On the root naming context object, call the operation `CosNaming::NamingContext::OBfactory()`. This returns a reference to a `LoadBalancing::ObjectGroupFactory` object.

4 On the object group factory, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`, passing the identifier for the group as a parameter. This operation returns a reference to the `LoadBalancing::ObjectGroup` object associated with the object group.

5 On the object group, call the operation `LoadBalancing::ObjectGroup::destroy()` to remove the group from the Naming Service.

If you already have a reference to the target `LoadBalancing::ObjectGroup` object, steps 3 and 4 are unnecessary.

Finding an Object Group without the Group Identifier

The procedures described in the previous sections assume that your application gets a reference to an object group using the group identifier. You can also get a reference to an object group if you know the name bound to the group in the Naming Service. To do this, call the operation CosNaming::NamingContext::resolve_object_group().

Example of Load Balancing with Object Groups

This section uses sample code to show how you can take advantage of object groups in your CORBA applications. The example described here is a very simple stock market system. In this example, a CORBA object has access to all current stock prices. Clients request stock prices from this CORBA object and display those prices to the user of the application.

In any realistic stock market application, there are potentially many stock prices available and many clients that require price updates without delay. Given such a high processing load, a single CORBA object may not be able to satisfy client requirements. A simple solution to this problem is to replicate the CORBA object, invisibly to the client, using object groups.

Sample code for the application described in this section is available in the `load_balancing` demonstration directory of your OrbixNames installation. This sample code may differ slightly from the code described in this section.

Defining the IDL for the Application

The architecture for the stock market system is shown in Figure 11 on page 72. Two servers process client requests for stock price information. The server `stockmarketserver1` creates two CORBA objects for this purpose. Server `stockmarketserver2` creates an additional CORBA object which, from a client perspective, provides exactly the same service as the objects in `stockmarketserver1`.

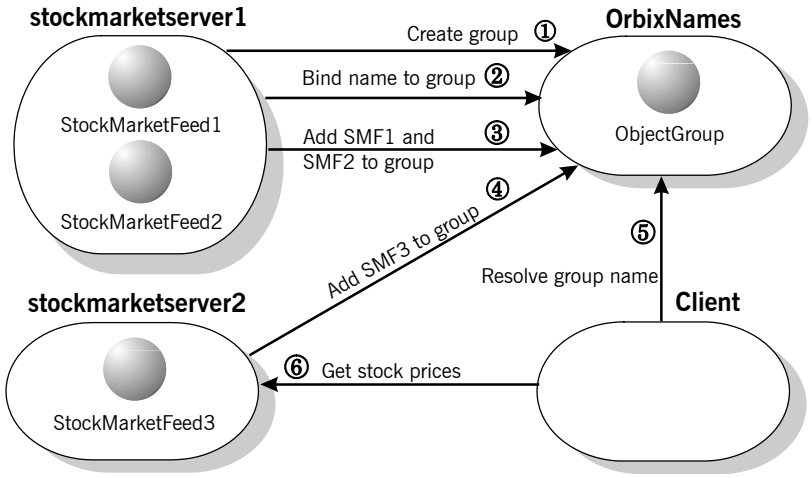

Figure 11 Architecture of the Stock Market Example

The IDL for this application requires only a single interface definition. This interface, called `StockMarketFeed`, is implemented by each of the three CORBA objects.

Interface `StockMarketFeed` is defined in the module `ObjectGroupDemo`:

```
// IDL
module  ObjectGroupDemo {
  interface StockMarketFeed {
    enum feedFailureDetails {
      service_interruption,
stock_feed_terminated};

    exception stock_unavailable {};
    exception stock_feed_failure {
      feedFailureDetails reason;
    };

    long read_stock (in string stock_name)
      raises (stock_unavailable,
stock_feed_failure);
  };
};
```

The interface `StockMarketFeed` includes a single operation, `read_stock()`, which returns the current price of the stock associated with a specified stock name. A name is a string identifier unique to each stock. This operation can raise the following exceptions:

`stock_unavailable`	This exception is raised by `read_stock()` to indicate that the specified stock name is not valid.
`stock_feed_failure`	A `stock_feed_failure` indicates that an error occurred in communications between the server and the source of stock prices.

Creating an Object Group and Adding Objects

After you define your IDL, the next step in developing an application is to implement your interfaces. Using object groups has no effect on how you do this, therefore this section assumes that you have defined a Java class, `StockMarketFeedImpl`, which implements the interface `StockMarketFeed`.

When you have implemented your IDL interfaces, you must develop a server program that contains and manages your implementation objects. In our application, we have two servers. Two `StockMarketFeed` implementation objects are created by `StockMarketServer1`, which extends the base `StockMarketServer` class. This creates an object group in the Naming Service, and adds the implementation objects to this group. The second server, `StockMarketServer2`, also extends `StockMarketServer`, creates an additional `StockMarketFeed` implementation object and adds this to the existing object group.

The key parts of the `StockMarketServer` class are:

```
// Java
// StockMarketServer.java
```

```
import org.omg.CORBA.*;
import org.omg.CosNaming.*;

import IE.Iona.OrbixWeb._OrbixWeb;
import IE.Iona.OrbixWeb.LoadBalancing.*;

import Demos.LoadBalancing.ObjectGroupDemo.*;
import
Demos.LoadBalancing.ObjectGroupDemo.StockMarketFee
dPackage.*;
...
public class StockMarketServer
{
 ...
 // Creates and registers the StockMarketFeed
 // objects that go into the round-robin load
 // balancing object group.
 private void registerStockMarketFeeds(ORB orb,
 ObjectGroup object_group, int number_of_feeds,
                int start_feed_number) throws Exception
   {
     for (int i = 0; i < number_of_feeds; i++)
     {
         // Create the stock market feed object
         // and connect to the orb
1        StockMarketFeedImpl stock_feed = new
           StockMarketFeedImpl(SMS_STOCK_MARKET_FEED_PREFIX
           + String.valueOf(start_feed_number + i));
       orb.connect(stock_feed);
 ...
 }

 // Create the Load Balancing round-robin object group
 private ObjectGroup getObjectGroup()
     throws Exception
 {
     ...
2        root_naming_context = getRootContext();
         resolved_obj =
     root_naming_context.resolve(name_components);
 ...
 }

 // Get the ObjectGroupFactory,
 // return ObjectGroupFactory
 private ObjectGroupFactory getObjectGroupFactory()
     throws Exception
   {
     // Get the Object Group Factory object
3    org.omg.CORBA.Object object =
     getRootContext().OBfactory();
     ObjectGroupFactory object_group_factory =
             ObjectGroupFactoryHelper.narrow(object);
 ...
     return object_group_factory;
 }
 ...

 // StockMarketServer constructor
```

```
    public StockMarketServer
            (ORB orb, String server_name,
            int number_of_feeds, int start_feed_number)
        throws Exception
    {
        ...
        // Create a round-robin object group
        // for load balancing
4       ObjectGroup object_group =
              createRoundRobinObjectGroup(orb,
              SMS_GROUP_IDENTIFIER, SMS_OBJECT_GROUP_NAME);

        // Creates and registers the StockMarketFeed
        // objects that go into the round-robin load
        // balancing object group.
5          registerStockMarketFeeds(orb, object_group,
              number_of_feeds, start_feed_number);
    ...
        // Handle client requests
6       _OrbixWeb.ORB(orb).impl_is_ready(server_name, 0);
    ...
    }
    ...
}
```

The functionality of this code is as follows:

1. The server creates implementation objects of type `StockMarketFeedImpl`.
2. The function `getRootContext()` returns a reference to the root naming context in the Naming Service. The implementation of this function is shown in "Getting the Root Naming Context".
3. The server calls the operation `OBfactory()` on the root naming context. This operation is implemented by the Naming Service and returns a factory object, of type `LoadBalancing.ObjectGroupFactory`, which the server can use to create object groups.
4. The server calls the function `createRoundRobinObjectGroup`(). This function uses the object group factory to create a new group with the specified identifier. It then binds a specified Naming Service name to this group. The implementation of `createRoundRobinObjectGroup`() is shown in "Creating an Object Group" on page 78.
5. The function `registerStockMarketFeeds`() adds the `StockMarketFeedImpl` objects to the object group created in step 4. The implementation of this function is shown in "Adding an Object to an Object Group" on page 80.
6. Finally, the server prepares to receive client requests by calling `_OrbixWeb.ORB(orb).impl_is_ready`.

Getting the Root Naming Context

The programs in this chapter use the following simple function to get a reference to the root naming context:

```
// Java
// StockmarketServer.java
// Gets the root context in the Naming Service
 private NamingContext getRootContext()
    throws Exception
```

```
  {
    if (m_root_naming_context == null)
    {
      org.omg.CORBA.Object naming_context_obj =
null;

      // Get the object reference.
      //
      try
      {
        displayMessage("getRootContext():
          Getting NameService object reference");
        naming_context_obj =

m_orb.resolve_initial_references("NameService");
        displayMessage("getRootContext():
            Got NameService object reference");
      }
      catch (org.omg.CORBA.ORBPackage.InvalidName
in)
      {
        throw new Exception(getServerName()
      + " - Could not retrieve NameService
reference");
      }
      catch (org.omg.CORBA.SystemException se)
      {
        throw new Exception(getServerName()
      + " - Error retrieving NameService
reference: "
      + se.getMessage());
      }
      if (naming_context_obj == null)
      {
        throw new Exception(getServerName() +
      " -
orb.resolve_initial_references(\"NameService\")
      returned a null object reference");
      }

      // Narrow the object reference.
      //
      try
      {
        displayMessage("getRootContext():
      Narrowing Object reference to
NamingContext");
        m_root_naming_context =

NamingContextHelper.narrow(naming_context_obj);
        displayMessage("getRootContext():
      Have narrowed NamingContext reference");
      }
      catch (SystemException se)
      {
        throw new Exception(getServerName() +
      " - NamingContextHelper.narrow() failed: "
      + se.getMessage());
      }
```

```
    if (m_root_naming_context == null)
    {
      throw new Exception(getServerName()
      + " - NamingContextHelper.narrow()
      returned a null object reference");
    }
  }

  return m_root_naming_context;
}
```

Creating an Object Group

In this example, the server calls the function `createRoundRobinObjectGroup()` to create an object group and bind a Naming Service name to it. You can implement this function as follows:

```
// Java
// StockMarketServer.java
...
// Create the Load Balancing round-robin object
group
  private ObjectGroup
createRoundRobinObjectGroup(ORB orb, String
group_identifier, String group_name)
    throws Exception
  {
    ObjectGroup         object_group;
    ObjectGroupFactory  object_group_factory =
getObjectGroupFactory();

    try
    {
1     object_group =
 object_group_factory.createRoundRobin(group_identifier);
2      bindNameToObjectGroup(orb, group_name, object_group);
    }
    catch (duplicate_group dg)
    {
      displayMessage("Object Group " +
group_identifier
      + " already exists, trying to find it ...");
      try
      {
        object_group =

object_group_factory.findGroup(group_identifier);
      }
      catch (no_such_group nsg)
      {
        throw new Exception(getServerName()
    + " - Couldn't find Object Group " +
group_identifier);
      }
    }
    return object_group;
  }
  ...
```

The function `createRoundRobinObjectGroup()` takes four parameters: a reference to the object group factory, a string value used to identify the new group, a string value used to create the name associated with all objects in the group, and a reference to the naming context in which this name should be bound.

The function `createRoundRobinObjectGroup()` makes two important calls:

1. It calls the operation `createRoundRobin()` on the object group factory in the Naming Service. This operation returns a new object group in which objects are selected on a round-robin basis.

2 Function `createRoundRobinObjectGroup`() then calls `bindNameToObjectGroup()`, a local function that binds a Naming Service name to the newly created group.

Binding a Name to an Object Group

The function `createRoundRobinObjectGroup`() calls the function `bindNameToObjectGroup()` to bind a name to the object group. When a client resolves this name, it receives a reference to one of the group's member objects, selected by the Naming Service in accordance with the group selection algorithm. The client does not know that the name is actually bound to a group of objects.

You can code `bindNameToObjectGroup`() as follows:

```
// Java
// StockMarketServer.java
// Binds a new ObjectGroup to a name in the
// Naming Service that the clients can refer to
and bind to
private void bindNameToObjectGroup(ORB orb,
    String object_group_name, ObjectGroup
object_group)
    throws Exception
  {
    // create a sequence of names for the resolve
    NameComponent[] name_components =
      new NameComponent[]
      {
        new
NameComponent(LOAD_BALANCING_CONTEXT_NAME, ""),
        new NameComponent(object_group_name, "")
      };

    // Get the root context in the Naming service
    displayMessage("binding name " +
LOAD_BALANCING_CONTEXT_NAME
            + "+" + object_group_name + " ...");
    getRootContext().bind(name_components,
object_group);
  }
```

The functionality of `bindNameToObjectGroup`() is quite straightforward. This function simply calls `getRootContext().bind()` on a naming context to associate a Naming Service name with an object. In this case, the object's true type is `LoadBalancing::ObjectGroup`, so the name is associated with an object group.

In this example, the object group name is bound in the context `LOAD_BALANCING_CONTEXT_NAME`. The code assumes that this naming context already exists. For example, you could create this context in the initialization code for `StockMarketServer`. Alternatively, you could use the OrbixNames `putnewncns` or `putnewncnsj` utilities, described in the chapter "Using the OrbixNames Utilities".

Adding an Object to an Object Group

After creating the object group, `StockMarketServer` adds its `StockMarketFeed` implementation objects to the group. To do this, the server calls the function `registerStockMarketFeeds()`:

```
// Java
// StockMarketServer.java
// Creates and registers the StockMarketFeed
objects
// that go into the round-robin load balancing
object group.
...
  private void registerStockMarketFeeds(ORB orb,
      ObjectGroup object_group, int
number_of_feeds,
                int start_feed_number)
    throws Exception
  {
    for (int i = 0; i < number_of_feeds; i++)
    {
      // Create the stock market feed object &
connect to the orb
1    StockMarketFeedImpl stock_feed =
     new StockMarketFeedImpl(SMS_STOCK_MARKET_FEED_PREFIX
          + String.valueOf(start_feed_number + i));
       orb.connect(stock_feed);

2    member new_member =
          new member(stock_feed, SMS_STOCK_MARKET_FEED_PREFIX
                 + String.valueOf(start_feed_number + i));

       // Add stock market feed object to this
object group
       displayMessage("adding member " +
new_member.id +
         " to object group " + object_group.id());
       try
       {
3       object_group.addMember(new_member);
       }
4    catch (duplicate_member dm)
       {
       // Remove existing duplicate and
       // then try to add our member again
     try
     {

object_group.removeMember(new_member.id);
           object_group.addMember(new_member);
     }
     catch (no_such_member nsm)
     {
           throw new Exception(getServerName() +
         " - problem adding member " +
new_member.id
         + " in object group " +
object_group.id());
     }
     catch (duplicate_member dm2)
     {
```

```
            throw new Exception(getServerName()
          + " - problem adding member " +
    new_member.id
          + " in object group " + object_group.id());
        }
      }
    }
```

The function `registerStockMarketFeeds()` takes four parameters: the server's ORB, the object group, the number of stock market feed objects added by this server, and the starting number for the first stock market feed object added. The member identifier `new_member.id` has no effect on the naming of the object within the Naming Service. To obtain a reference to the object, a client resolves the name bound to the object group.

The functionality of `registerStockMarketFeeds()` is as follows:

1. The server creates a new `StockMarketFeedImpl` object, connecting it to the ORB using `connect()`.
2. The server creates an IDL struct of type `LoadBalancing::member` which contains two items: a reference to the previously created `StockMarketFeedImpl` object, and a string that identifies the object within the group.
3. The server adds the new member to the object group in the Naming Service by calling the operation `addMember()` on the corresponding `LoadBalancing::ObjectGroup` object.
4. If the string identifier of the new member clashes with an existing member identifier, the operation `addMember()` throws an exception of type `LoadBalancing::duplicate_member` to indicate this. In this case `addMember()` does not update the contents of the object group in the Naming Service, and the catch cause checks various possible reasons for failure.

Creating Replicated Objects

In this example, `StockMarketServer1` and `StockMarketServer2` extend `StockMarketServer` and implement the creation of the required stock market feeds. To do this, they create new `StockMarketFeed` implementation objects by calling their `StockMarketServer` superclass and inheriting the Naming Service-related functions originally defined there.

```
// Java
// StockMarketServer1 - 2 server feeds
import org.omg.CORBA.ORB;

public class StockMarketServer1
  extends StockMarketServer
{
  public static void main(String args[])
  {
    try
    {
      // initialize the ORB
    org.omg.CORBA.ORB orb = ORB.init(args,null);

      // Create a new server and let it go ...
1    new StockMarketServer1(orb);
```

```
        }
        catch (Exception ex)
        {
          ex.printStackTrace();
          System.exit(1);
        }
      ...
      }

      // Stock market server 1 constructor.
      public StockMarketServer1 (ORB orb)
        throws Exception
      {
2      super(orb, "stockmarketserver1", 2, 1);
      }
    }

    // Java
    // StockMarketServer2 - 1 feed
    public class StockMarketServer2
      extends StockMarketServer
    {
      ...
3      new StockMarketServer2(orb);
      ...
      public StockMarketServer2 (ORB orb)
        throws Exception
      {
4      super(orb, "stockmarketserver2", 1, 3);
      }
    }
```

The functionality of this code is as follows:

1. Create the new `StockMarketServer1` object.
2. Constructor for the new `StockMarketServer1` object that specifies two `StockMarketFeedImpl` objects through its superclass.
3. Create the new `StockMarketServer2` object.
4. Constructor for the new `StockMarketServer2` object that specifies one `StockMarketFeedImpl` object through its superclass.

Finding an Existing Object Group

A key part of `StockMarketServer` is the function `find_group()`, which retrieves a reference to an existing object group. The function `createRoundRobinObjectGroup() accomplishes this as follows`:

```
// Java
// StockMarketServer.java
...// Creates the Load Balancing round-robin
object group
  private ObjectGroup
createRoundRobinObjectGroup(ORB orb, String
group_identifier, String group_name)
    throws Exception
  {
    ObjectGroup         object_group;
    ObjectGroupFactory  object_group_factory =

getObjectGroupFactory();
```

```
    try
    {
      object_group =

object_group_factory.createRoundRobin(group_identi
fier);
      bindNameToObjectGroup(orb, group_name,
object_group);
    }
    catch (duplicate_group dg)
    {
      displayMessage("Object Group " +
group_identifier
        + " already exists, trying to find it
...");
      try
      {
1   object_group =
        object_group_factory.findGroup(group_identifier);
      }
      catch (no_such_group nsg)
      {
        throw new Exception(getServerName()
    + " - Couldn't find Object Group " +
group_identifier);
      }
    }
    return object_group;
  } ...
```

The functionality of this code is as follows:

1 The server calls the operation `findGroup()` on the object group factory. The operation `findGroup()` is defined on the interface `LoadBalancing::ObjectGroupFactory`. Given a group identifier, this operation returns a reference to the corresponding `LoadBalancing::ObjectGroup` object.

Accessing the Objects from a Client

All objects in an object group provide the same service to clients. A client that resolves a name in the Naming Service does not know if the name is bound to an object group or a single object. The client receives a reference to one object only. A client program resolves an object group name in exactly the same way as it resolves a name bound to just one object.

For example, the stock market example client could look like this:

```
// Java
// StockMarketClient

import org.omg.CORBA.*;
import org.omg.CosNaming.*;
import IE.Iona.OrbixWeb.LoadBalancing.*;
import Demos.LoadBalancing.ObjectGroupDemo.*;
import Demos.LoadBalancing.ObjectGroupDemo.
              StockMarketFeedPackage.*;
...
public class StockMarketClient
{
```

```
  public static void main(String args[])
  {
    try
    {
      //
      // initialize the ORB
      org.omg.CORBA.ORB orb = ORB.init(args,null);

      //
      // Create a new client and let it go ...
      new StockMarketClient (orb);
    }
    catch (Exception ex)
    {
      ex.printStackTrace();
    }
  }

  ...
}

  // Reads & displays the stock prices for the
list of stocks.
  public void readStockPrices(String[]
stock_names_list)
    throws Exception
  {
    StockMarketFeed stock_market_feed;
    String stock_name;
    int stock_price = 0;
  ...
  }

  // Get a StockMarketFeed.
  private StockMarketFeed getStockMarketFeed()
    throws Exception
  {
    StockMarketFeed        stock_market_feed;
    org.omg.CORBA.Object   resolved_obj;

 // Pick the next StockMarketFeed object from the object
 // group. Each object group has a selection algorithm
 // associated with it when created. This algorithm
 // (random/round-robin) is applied when clienmts
 // resolve the name associated with object group and
 // return the object.

    resolved_obj = getObjectInObjectGroup();

    m_current_feed_id =
getIdForMember(resolved_obj);

    if (resolved_obj == null)
    {
      throw new Exception("getStockMarketFeed() -
                Resolved object is null ...");
    }

stock_market_feed =
StockMarketFeedHelper.narrow(resolved_obj);
```

```
   ...
    return stock_market_feed;
  }

    // Get the Object Group containing our
StockMarketFeeds.
  private ObjectGroup getObjectGroup()
    throws Exception
  {
    if (m_object_group == null)
    {
      NamingContext          root_naming_context;
      org.omg.CORBA.Object   resolved_obj;

      // create a sequence of names for the
resolve
      NameComponent[] name_components =
        new NameComponent[]
        {
          new
NameComponent(LOAD_BALANCING_CONTEXT_NAME, ""),
          new NameComponent(GROUP_SERVER_NAME, "")
        };

      // Get the root context in the Naming
service
      root_naming_context = getRootContext()

      resolved_obj =
root_naming_context.resolve_object_group(name_comp
onents);

      if (resolved_obj == null)
      {
        throw new Exception("getObjectGroup() -
                Resolved object is null ...");
      }

      m_object_group =
ObjectGroupHelper.narrow(resolved_obj);

    ...
    return m_object_group;
  }

  // Gets the StockMarketFeed object in the Object
Group//
  private org.omg.CORBA.Object
getObjectInObjectGroup()
    throws Exception
  {
    NamingContext root_naming_context;
    org.omg.CORBA.Object resolved_obj;

    //Create a sequence of names for the resolve//
    NameComponent[] name_components = new
NameComponent[
    {
```

```
      new
NameComponent(LOAD_BALANCING_CONTEXT_NAME, " "),
      new NameComponent(GROUP_SERVER_NAME, " ")
    };

  // Gets the root context in the Naming Service
//
  root_naming_context = getRootContext();
  resolved_obj =
root_naming_context.resolve(name_components);
  if (resolved_obj == null)
    {
    throw new Exception("getObjectInObjectGroup()
-
              Resolved object is null ...");
    }
    return resolved_obj;
  }
  // Gets the root context in the Naming Service
  private NamingContext getRootContext()
    throws Exception
  {
    if (m_root_naming_context == null)
    {
      org.omg.CORBA.Object naming_context_obj =
null;

      // Get the object reference.
      try
      {
        naming_context_obj =

m_orb.resolve_initial_references("NameService");
      }
      ...
      // Narrow the object reference.
      try
      {
        m_root_naming_context =

NamingContextHelper.narrow(naming_context_obj);
      }
    ...
  return m_root_naming_context;
  }

  // Returns the ID for a group member.
  private String
getIdForMember(org.omg.CORBA.Object member_obj)
  {
    try
    {
      String[] member_ids =
getObjectGroup().members();

      for (int i = 0; i < member_ids.length; i++)
      {
        if
(getObjectGroup().getMember(member_ids[i]).
        toString().equals(member_obj.toString()))
```

```
            {
              return member_ids[i];
            }
          }
        }
      ...
      return "Unknown";
      }

      ...
    }
```

Part IV

OrbixNames Administrator's Guide

In this part

This part contains the following:

Using the OrbixNames Utilities

OrbixNames provides a set of command line utilities that allow you to monitor and manage the Naming Service externally to your applications. This chapter describes these utilities.

The OrbixNames command line utilities allow you to manipulate the contents of the Naming Service directly. It is often useful to do this. For example, the utilities are especially convenient when testing applications that use the Naming Service.

There are two general categories of OrbixNames utilities:

- The *name management utilities* allow you to create, delete, and examine name bindings in the Names Repository.
- The *object group management utilities* allow you to create, delete, and manage the contents of object groups.

This chapter examines both types of utility in detail.

Managing Name Bindings

The name management utilities allow you to create and manipulate name bindings directly from the command line. You can use these utilities to construct and navigate a naming graph.

The name management utilities are:

Native	Functionality
`catns`	Given a name, outputs a reference to the object to which the name is bound. If the object reference is an Interoperable Object Reference (IOR), the reference is parsed and the information displayed.
`lsns`	Lists bindings in a context.
`newncns`	Creates a new unbound context. You can subsequently bind a name to the context using `putns` or `putnsj`.
`putns`	Binds a name to an object.
`putncns`	Binds a name to an unbound context created using `newncns` or `newncnsj`.
`putnewncns`	Creates a new context and binds a name to it.
`reputns`	Rebinds a name to an object.
`reputncns`	Rebinds a context, removing the original binding.
`rmns`	Removes a name binding and optionally deletes a naming context.

The remainder of this uses these utilities to build a naming graph and populate it with name bindings. The full syntax for the utilities is given in “Syntax of the Name Management Utilities” on page 96. Examples use the native name management utilities; you may generally substitute the “j” java name management utilities throughout.

Note

Many of these utilities take object references as command line arguments. These object references are expected in the string format returned from the function `CORBA::ORB::object_to_string()`. By default, this string format represents an Interoperable Object Reference (IOR). In this chapter, all object references are shown in native Orbix format for convenience. To use IORs, do not specify the `-orbixprot` option when running the utilities.

Using the Name Utilities

This section uses the OrbixNames utilities to build the naming graph used in the chapters "C++ Programming with OrbixNames" and "Java Programming with OrbixNames". Figure 12 recalls the structure of this graph.

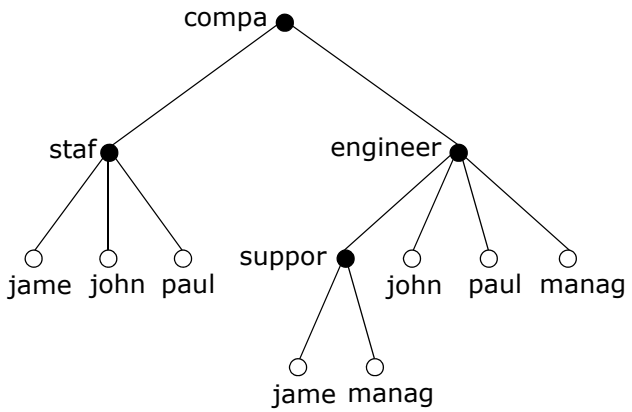

Figure 12 A Naming Context Graph

Creating Naming Contexts

The simplest way to create a naming context is to use the `putnewncns` utility. For example, the following command creates a new context bound to the name with the ID `company` and an empty kind value:

```
putnewncns -orbixprot company
```

The name is given in the format `id-kind`. The combination of ID and kind fields must unambiguously specify the name.

Further examples are:

- Create a new naming context bound to the name `company.engineering` (the context `company` must already exist).

  ```
  putnewncns "-orbixprot" company.engineering
  ```

- Create a new context bound to the name `company.engineering.support` (the context `company.engineering` must already exist).

  ```
  putnewncns "-orbixprot"
  company.engineering.support
  ```

You can also use the `newncns` utility to create an unbound context:

```
newncns "-orbixprot"
Created new UNBOUND Naming Context with object reference
:\host.iona.com:NS:NC_3::IR:CosNaming_NamingContext
```

A context created using `newncns` can be bound using the `putncns` utility. The following command binds the new context to the name `company.staff`.

```
putncns "-orbixprot" company.staff -
":\host.iona.com:NS:NC_3::IR:CosNaming_NamingContext"
```

Creating Name Bindings

To bind a name to an object, use the `putns` utility. Given the naming context graph show in Figure 12 on page 92, the examples in this section assume the following object reference strings are associated with the application objects:

```
james    :\host.iona.com:staff:0::IR:Person
john     :\host.iona.com:staff:1::IR:Person
paula    :\host.iona.com:staff:2::IR:Person
```

You can bind these objects to appropriate names within the `company.staff` naming context as follows:

```
putns company.staff.james-person -
    ":\host.iona.com:staff:0::IR:Person" -"orbixprot"

putns company.staff.john-person -
    ":\host.iona.com:staff:1::IR:Person" "-orbixprot"

putns company.staff.paula-person -
    ":\host.iona.com:staff:2::IR:Person" "-orbixprot"
```

Each of these employee records has been assigned the kind `record` in the final component of its name.

To build the naming graph further, create additional bindings based on the divisions that employees are assigned to:

```
putns company.engineering.john-person -
    ":\host.iona.com:staff:1::IR:Person" "-orbixprot"

putns company.engineering.paula-person -
    ":\host.iona.com:staff:2::IR:Person" "-orbixprot"

putns company.engineering.support.james-person -
    ":\host.iona.com:staff:0::IR:Person" "-orbixprot"
```

To allow an application to find the manager of a division easily, add the following bindings:

```
putns company.engineering.manager-person -
    ":\host.iona.com:staff:2::IR:Person" "-orbixprot"

putns company.engineering.support.manager-person -
    ":\host.iona.com:staff:0::IR:Person" "-orbixprot"
```

Note that the names `company.staff.paula-person`, `company.engineering.paula-person` and `company.engineering.manager-person` now all resolve to the same object.

The naming contexts and name bindings created by the above sequence of commands builds the complete naming graph shown in Figure 12 on page 92.

Listing Name Bindings

The utility `lsns` lists all the bindings in a naming context. The following command lists the bindings in the context `company.engineering` in the OrbixNames server on host `alpha`:

```
lsns "-h" alpha "-orbixprot" company.engineering
Contents of company.engineering
    paula (Object)
    support (Context)
    john (Object)
    manager (Object)
```

The type of the binding is also listed. A binding of type `Object` names an object; a binding of type `Context` names a naming context, that is a node in the naming graph that participates in name resolution.

By default, only the ID of each name is listed by `lsns`. However, `lsns` supports a `-k` switch that allows you see both the ID and kind in the listing:

```
lsns "-h" "host" "-k" "-orbixprot"
company.engineering
Contents of company.engineering
    paula-person (Object)
    support- (Context)
    john-person (Object)
    manager-person (Object)
```

Regardless of whether the `-k` switch is specified, `lsns` can always accept a command line argument in the `id-kind` format.

Finding Object References by Name

The `catns` utility outputs the object reference for the application object or context object to which a name is bound. For example:

```
catns "-orbixprot" company.engineering
:\host.iona.com:NS:NC_1::IR:CosNaming_NamingContext
```

The names `company.staff.paula-person` and `company.engineering.manager-person` resolve to the same object:

```
catns "-orbixprot" company.staff.paula-person
:\host.iona.com:staff:2::IR:Person

catns "-orbixprot"
     company.engineering.manager-person
:\host.iona.com:staff:2::IR:Person
```

Rebinding a Name to an Object or Naming Context

The `reputns` utility changes the binding for an object name. This is analogous to the `CosNaming::NamingContext::rebind()` operation. For example, the name `company.engineering.paula-person` and the name `company.engineering.manager-person` currently resolve to the same object. To give `john` responsibility for management, you can rebind the name `manager-person` in the context `company.engineering`:

```
catns "-orbixprot" company.engineering.john-person
:\host.iona.com:staff:1::IR:Person
reputns "-orbixprot" -
    company.engineering.manager-person -
    ":\host.iona.com:staff:1::IR:Person"
```

The `reputncns` utility changes the binding for a naming context. This is analogous to the `CosNaming::NamingContext::rebind_context()` operation. To illustrate the use of this utility, first create a new context bound to the name `company.staff.supportStaff`:

```
putnewncns "-orbixprot" company.staff.supportStaff
```

Suppose now that the context `company.staff.suppportStaff` should contain the same information as `company.engineering.support`. Rather than maintaining two separate contexts, a better option is to rebind the name `company.staff.supportStaff` so that it points to the `company.engineering.support` context:

```
catns "-orbixprot" company.engineering.support
":\host.iona.com:NS:NC_2::IR:CosNaming_NamingContext"

reputncns "-orbixprot" company.staff.supportStaff
":\host.iona.com:NS:NC_2::IR:CosNaming_NamingContext"

lsns "-k" "-orbixprot" company.staff.supportStaff
Contents of company.staff.supportStaff
  james-person (Object)
  manager-person (Object)
```

This sequence of commands leaves the context previously named by `company.staff.supportStaff` unreachable; that is, the naming context object exists in the Naming Service, but it has no corresponding name binding. In this case, the naming context is assigned a name in the OrbixNames `lost+found` context, as described in "Finding Unreachable Context Objects" (C++) or "Finding Unreachable Context Objects" (Java).

Removing Name Bindings

The `rmns` utility removes a name binding. For example, the following commands remove the `manager` bindings:

```
rmns "-orbixprot"
company.engineering.manager-person
rmns "-orbixprot" -
  company.engineering.support.manager-person
```

Take care not to leave naming contexts unreachable. For example:

```
rmns "-orbixprot" company.engineering
```

This command unbinds the name `company.engineering` and moves the corresponding naming context object into the `lost+found` context.

Syntax of the Name Management Utilities

The following is a summary of the command syntax for the name management utilities:

```
catns [-v] [-s] [-h <host>] [-orbixprot] <name>
catnsj [-v] [-h <host>] [-orbixprot] <name>

lsns [-v] [-s] [-h <host>] [-k] [-c] [-orbixprot]
[name]
lsnsj [-v] [-h <host>] [-k] [-c] [-orbixprot]
[name]

newncns [-v] [-s] [-h <host>] [-orbixprot]
newncnsj [-v] [-h <host>] [-orbixprot]

putncns [-v] [-s] [-h <host>] [-orbixprot] \
  <name> { <context-ref> | -f <file> }
putncnsj [-v] [-h <host>] [-orbixprot] \
  <name> { <context-ref> | -f <file> }

putnewncns [-v] [-s] [-h <host>] [-orbixprot]
<name>
putnewncnsj [-v] [-h <host>] [-orbixprot] <name>

putns [-v] [-s] [-h <host>] <name> \
  { <object-ref> | -f <file> } [-orbixprot]
putnsj [-v] [-h <host>] <name> \
  { <object-ref> | -f <file> } [-orbixprot]

reputncns [-v] [-s] [-h <host>] [-orbixprot] \
  <name> { <context-ref> | -f <file> }
reputncnsj [-v] [-h <host>] [-orbixprot] \
  <name> { <context-ref> | -f <file> }

reputns [-v] [-s] [-h <host>] [-orbixprot] \
  <name> { <object-ref> | -f <file> }
reputnsj [-v] [-h <host>] [-orbixprot] \
  <name> { <object-ref> | -f <file> }

rmns [-v] [-s] [-h <host>] [-x] [-orbixprot]
<name>
rmnsj [-v] [-h <host>] [-x] [-orbixprot] <name>
```

The common options are:

`-h <host>`	Specifies the host on which the OrbixNames server is located. By default, the utilities use the Initialization Service to locate the server. The `-h` switch forces the utilities to use `_bind()` instead.
`-f <file>`	Any utilities which take an object reference or context reference as an argument can optionally specify a file, using this switch, instead of putting the object reference on the command line itself.
`-orbixprot`	Communicates with OrbixNames using the Orbix protocol. The default is the CORBA Internet Inter-ORB Protocol (IIOP).
`-s`	Required for all the native (that is, non-Java) utilities to communicate with an SSL-enabled OrbixNames server. The utility will prompt for a password. OrbixSSL must have been installed and the OrbixSSL-specific `update` utility executed. Refer to the OrbixSSL documentation for further information.
`-v`	Outputs version information. Specifying `-v` does not cause the utility to run.
`-x`	This switch only applies when removing a naming context. This switch unbinds the context and then destroys it.

Managing Object Groups

In addition to the name management utilities, OrbixNames provides utilities that allow you to manipulate object groups and their members. The object group management utilities are available as both native and Java executables with similar functionality.

These utilities are:

Native	Java	Functionality
`new_group`	`new_groupj`	Creates an object group and binds it to a name in OrbixNames.
`del_group`	`del_groupj`	Deletes an object group.
`cat_group`	`cat_groupj`	Returns the stringified object reference of an object group.
`list_members`	`list_membersj`	Lists the members of an object group.
`add_member`	`add_memberj`	Adds a member to an object group.
`del_member`	`del_memberj`	Deletes a member from an object group.
`cat_member`	`cat_memberj`	Returns the stringified object reference of a member of an object group.
`pick_member`	`pick_memberj`	Selects a member of an object group.

Using the Object Group Utilities

This section provides examples of each of the object group utilities. When using these utilities, you can identify a group by specifying the group identifier, with the `-i` switch, or the name bound to the group, with the `-n` switch.

Creating and Deleting Object Groups

To create an object group and bind a name to it, use the `new_group` utility. For example:

```
new_group marketing_file_server_group -
    company.marketing.file_server "-random"
```

This command creates an object group with group identifier `marketing_file_server_group` and binds it to the name `company.marketing.file_server`. OrbixNames uses a random selection algorithm to choose an object from this group.

To associate a round-robin selection algorithm with the group, use the `-round_robin` switch:

```
new_group engineering_file_server_group -
    company.engineering.file_server "-round_robin"
```

To list all the existing object groups, use the `list_groups` utility:

```
list_groups

Round Robin Object Group List
           engineering_file_server_group
Random Object Group List
           marketing_file_server_group
```

To delete an object group, use the `del_group` utility:

```
del_group "-i" engineering_file_server_group
```

This command deletes the object group with identifier `engineering_file_server_group`. Use the `-i` switch only if the group has no associated name. If a name is bound to the group, specify this name using the `-n` switch:

```
del_group "-n" company.marketing.file_server
```

Managing the Members of an Object Group

Each member of an object group requires a unique identifier. To add a member to a group, use `add_member`. For example:

```
add_member "-i" engineering_file_server_group -
    member_1 IOR string
```

This command adds a new member `member_1` to the object group `engineering_file_server_group`. You can also identify the object group using the group name:

```
add_member "-n" company.engineering.file_server -
    member_2 IOR string
```

Use the `list_members` utility to list members of an object group:

```
list_members -ncompany.engineering.file_server
    member_1
    member_2
```

Use the `del_member` utility to remove a member from an object group:

```
del_member -ncompany.engineering.file_server -
    member_2
```

To retrieve the object reference associated with an object group member, use the `cat_member` utility:

```
cat_member member_2 -
    -ncompany.engineering.file_server
```

The `pick_member` utility cycles through the members of an object group:

```
pick_member -ncompany.engineering.file_server
    First IOR string
pick_member -ncompany.engineering.file_server
    Second IOR string
```

Syntax of the Object Group Utilities

This section summarizes the command syntax for the object group utilities:

```
add_member [-i <object group id> | -n <object
group name>]
  <member id> <obj> [-h <host>] [-orbixprot] [-v]

cat_group [-i <object group id> | -n <object group
name>]
  [-h <host>] [-orbixprot] [-v]

cat_member [-i <object group id> | -n <object
group name>]
  <member_id>  [-h <host>] [-v]

del_group [-i <object group id> | -n <object group
name>]
  [-h <host>] [-v]

del_member -i <object group id> | -n <object group
name>]
  <member_id> [-h <host>] [-orbixprot] [-v]

list_groups [-h <host>] [-orbixprot] [-v]

list_members [-i <object group id> | -n <object
group name>]
  [-h <host>] [-orbixprot] [-v]

new_group <object group id> <object group name>
  {-random | -round_robin} [-h <host>] -orbixprot]
[-v]

pick_member  [-i <object group id> | -n <object
group name>]
  [-h <host>] [-orbixprot] [-v]
```

The common options are:

`-h <host>`	Specifies the target host on which OrbixNames is running. This switch defaults to the local host.
`-v`	Outputs version information.
`-i`	Identifies an object group by specifying the identifier.
`-n`	Identifies an object group by specifying the name bound to it.
`-orbixprot`	Communicates with the OrbixNames server using the Orbix protocol. The default protocol is CORBA Internet Inter-ORB Protocol (IIOP).

The OrbixNames Browser

The OrbixNames Browser provides a graphical interface to OrbixNames. Like the OrbixNames utilities, the browser allows you to monitor and manage the Naming Service externally to your applications.

The OrbixNames Browser provides full access to the contents of the Naming Service. Using the browser, you can manipulate the contents of the Naming Service directly. For example, you can create naming contexts, bind names to objects, create and modify object groups, and examine the existing name bindings in the Naming Service.

Starting the OrbixNames Browser

On UNIX, start the OrbixNames Browser by running the `NAmesBrowser.sh` script, located in the `bin` directory of your Orbix installation. On Windows, you can run the OrbixNames Browser from the Windows **Start** menu, or run the batch file `NamesBrowser.bat` from the `bin` folder of the Orbix installation. The main browser window appears as shown in Figure 13.

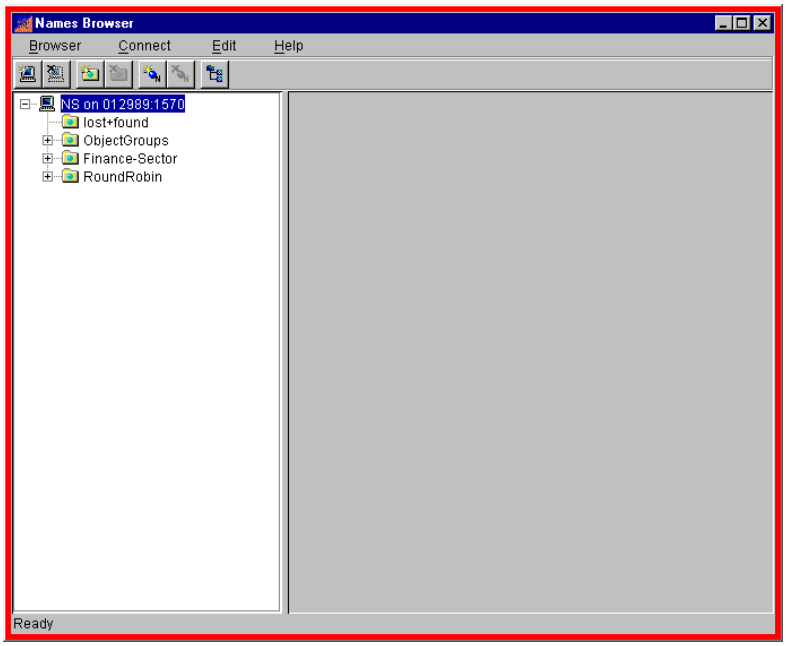

Figure 13 The Main OrbixNames Browser Window

The browser interface includes the following elements:

- *A menu bar*.
- *A toolbar*.
- *A navigation tree*. This tree displays a graphical representation of the names and naming contexts stored in OrbixNames.

Connecting to an OrbixNames Server

To connect to an OrbixNames server on a host in your network:

1 Select **Connect>Connect Name Service**, as shown in Figure 14.

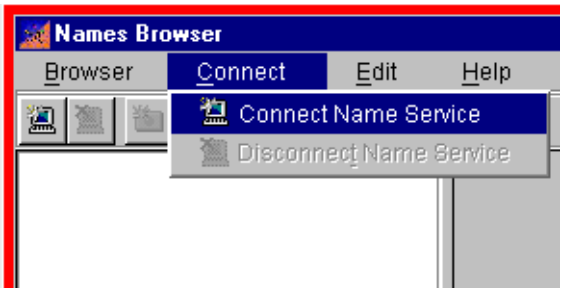

Figure 14 Activating the Naming Service Connection

1 The **Connect to Naming Service** dialog box appears as shown in Figure 15.

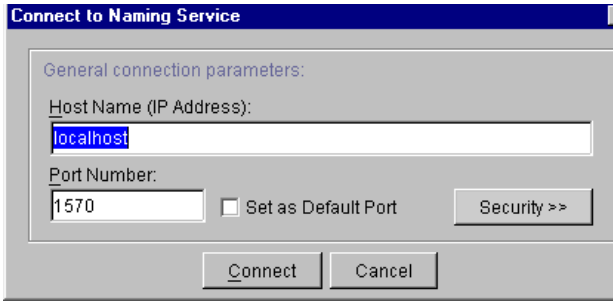

Figure 15 Connecting to an OrbixNames Server

1 In the **Host Name (IP Address)** text box, enter the name or IP address of the target host.

2 Select **Connect**. The browser navigation tree displays an unexpanded view of the current name bindings for the OrbixNames server at the target host, as shown in Figure 16.

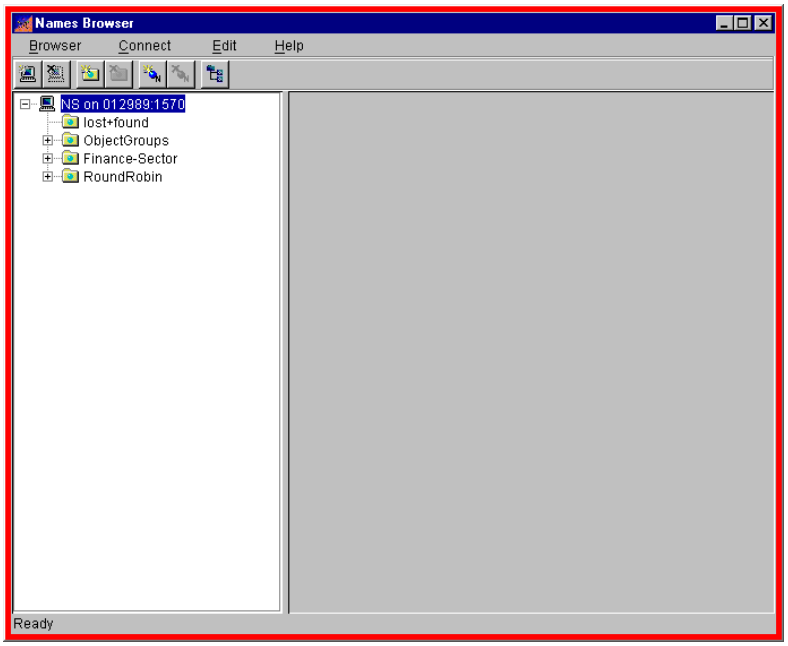

Figure 16 Current Bindings For a Selected Host

If you wish to connect to an OrbixNames server on a second host, repeat these steps for the new host. You do not need to disconnect from the original host.

Connecting to a Secure OrbixNames Server

Naming Services may be Secure Sockets Layer-enabled to provide security. Refer to the OrbixSSL documentation for further information.

Note

OrbixSSL must be installed to allow connection to secure Naming Services and other SSL-enabled CORBA services that will only accept secure connections.

To connect to a secure OrbixNames server on a host in your network:

1 Select **Connect>Connect Name Service**, as before.

2 The **Connect to Naming Service** dialog box appears as shown in Figure 17.

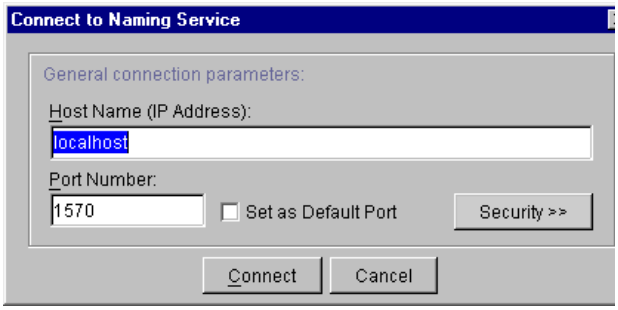

Figure 17 Connecting to an OrbixNames Server

3 In the **Host Name (IP Address)** text box, enter the name or IP address of the target host.

4 Click the **Security>>** button. The **Connect to Naming Service** dialog box expands to display SSL-specific security options, as shown in Figure 18. If the **Security>>** button is

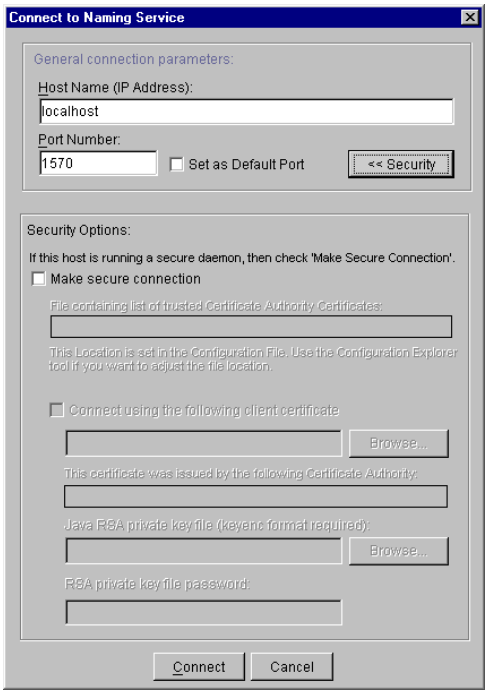

Figure 18 Connection to Naming Service Security Options

ghosted, then a suitable SSL security layer has not been installed.

5 Select the *Make secure connection* checkbox to request a secure connection. The location of the trusted Certificate Authority Certificates is set in the Configuration Explorer as `IT_CA_LIST_FILE`.

6 If the secure Naming Service requests a client certificate, select the *Connect using the following client certificate* checkbox, then click **Browse** to locate a suitable certificate file.

7 You may select a Java RSA private key using the appropriate **Browse** option.

8 You may also enter the RSA password for the private key file in the appropriate text box.

9 Select **Connect**. The browser navigation tree displays the current name bindings for the OrbixNames server at the target host.

Note

You may have only one secure connection active at any one time. Therefore, although you may have multiple insecure connections active in addition to a single secure connection, attempting a second secure connection will result in an exception. You must first disconnect from the original secure connection.

Disconnecting from an OrbixNames Server

To disconnect from an OrbixNames server:

1 In the navigation tree, select the host icon for the Naming Service you wish to disconnect from.

2 Select **Connect>Disconnect Name Service**. A **Warning** dialog box is displayed, as shown in Figure 20.

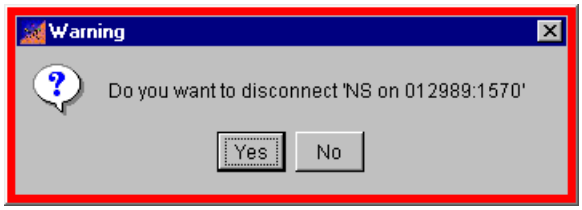

Figure 19 Disconnecting from the Naming Service

1 Select Yes to disconnect from the indicated Naming Service host.

2 Alternatively, clicking the secondary mouse button while a Naming Service host is selected will bring up a context dialog

box, as shown in Figure 20. This also allows connection or disconnection.

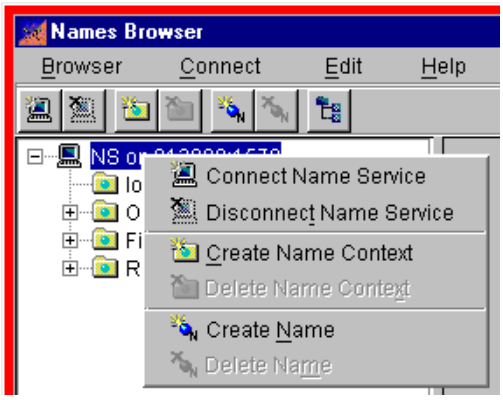

Figure 20 Context-Sensitive Connection Dialog

Managing Naming Contexts

The OrbixNames Browser allows you to create new naming contexts, modify existing naming contexts, and remove naming contexts from an OrbixNames server.

Note that removing a naming context recursively removes all context and name objects below that naming context.

Creating a Naming Context

To create a naming context:

1 In the browser navigation tree, navigate to the naming context within which you wish to create the new context.

2 Select **Edit>Create Name Context**. A new context is displayed as shown in Figure 21.

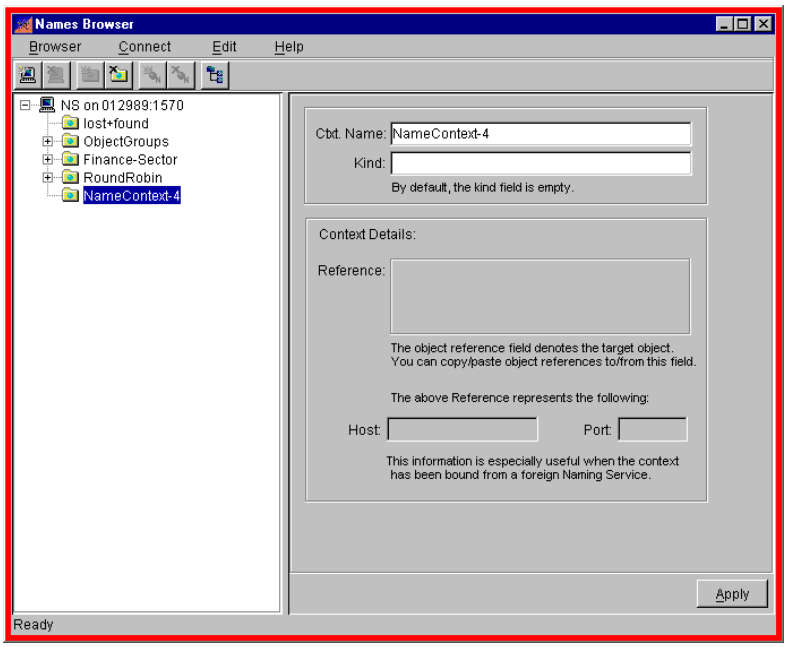

Figure 21 Creating a New Naming Context

3 Enter a context name in the **Ctxt. Name** text box.

4 If you wish, you can enter a context `kind` in the **Kind** text box.

5 Paste an object reference into the **Reference** text box. If you do not paste a reference, one will be created for you.

6 Click the **Apply** button. The new context's details are displayed.

Note that a `kind` value for a name in the CORBA Naming Service cannot be null. If you do not specify a `kind` value when assigning a name to a naming context, the OrbixNames Browser sets the `kind` to the null string.

Modifying a Naming Context

The OrbixNames Browser allows you to change the object reference associated with a specified naming context. Using this feature, you can link an existing context name to a context object associated with another name.

To change the object reference associated with a naming context:

1 In the browser navigation tree, navigate to the naming context you want to modify.

2 To change either the name or the kind of the naming context, enter a new name into either the **Ctxt. Name** or the **Kind** text box.

3 To change the object reference, paste a new object reference into the **Reference** text box, as shown in Figure 22.

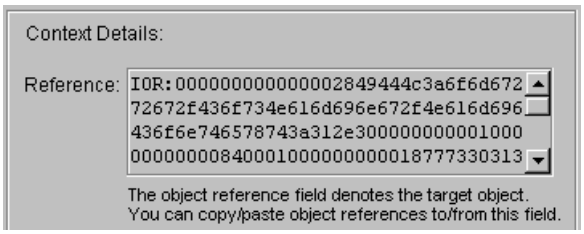

Figure 22 The Reference Text Box in the Context Details

4 Click the **Apply** button. The context's new details are displayed.

5 You can select **Edit>Refresh** to ensure that the navigation tree shows the updated context details.

Removing a Naming Context

To remove a naming context:

1 Select the icon of the naming context you want to remove.

2 Select **Edit>Delete Name Context**. A confirmation dialog box appears.

3 Select **Yes** to confirm the removal of the naming context.

4 Alternatively, clicking the secondary mouse button while a naming context is selected will bring up a context dialog box, as shown in Figure 23. This allows the creation or deletion of the selected naming context.

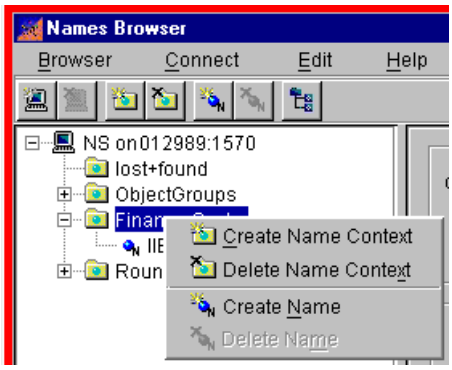

Figure 23 Context-Sensitive Naming Context Dialog

Managing Object Names

The OrbixNames Browser allows you to bind a name to an object in a CORBA application, modify the object binding for an existing name, and remove an object name from an OrbixNames server.

Binding a Name to an Object

Before attempting to bind a name to an object, ensure that you have access to the string form of the object reference. To get the string form of an object reference, pass the object reference as a parameter to the function `CORBA::ORB::object_to_string()` in the source code of your application.

To bind a name to an object:

1. Get the string form of a reference to the object.
2. In the browser navigation tree, navigate to the naming context in which you want to create the object name.
3. Select **Edit>Create Name**. A new name binding appears as shown in Figure 24.

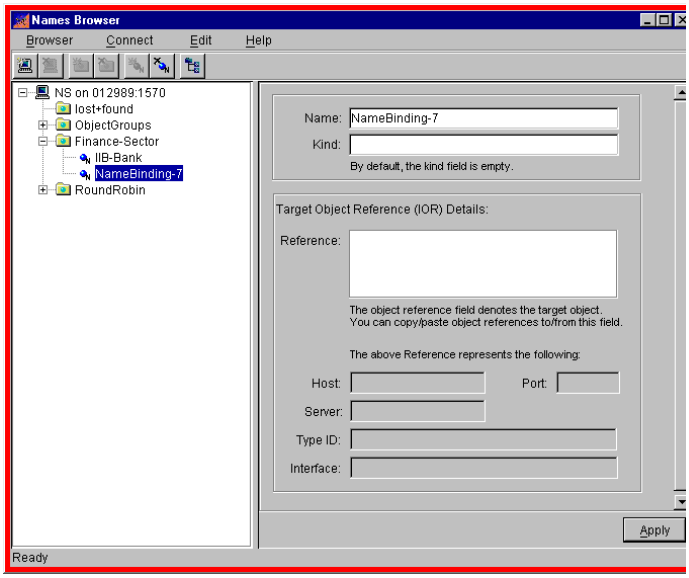

Figure 24 Creating a Name Binding

4. In the **Name** text box, enter the identifier value for the new `id`.
5. In the **Kind** text box, enter your desired `kind` value.
6. Paste the object reference string into the **Reference** text box.
7. Click the **Apply** button. The new object details are displayed, similar to the display in Figure 25.

If you do not specify a `kind` value when assigning a name to a CORBA object, the OrbixNames browser sets the `kind` to the null string.

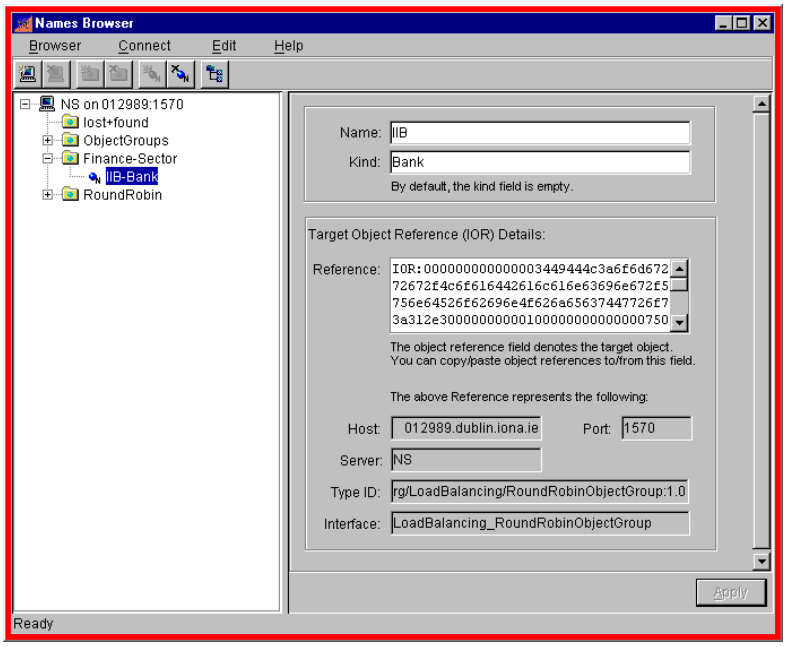

Figure 25 Viewing an Object Name in the Main Browser Window

Modifying an Object Binding

To change the object reference associated with a name in the CORBA Naming Service:

1. In the browser navigation tree, navigate to the object you want to modify.
2. To change the `id`, select the **Name** text box and enter the identifier value for the new name. To change the `kind`, select the **Kind** text box enter the kind value for the new name.
3. To change the object reference, paste the new object reference string into the **Reference** text box.
4. Click the **Apply** button to confirm the new object binding.

Removing an Object Name

To remove an object name from the CORBA Naming Service:

1. In the browser navigation tree, navigate to the object you want to modify.
2. Select **Edit>Delete Name**. A confirmation dialog box appears.
3. Select **Yes** to confirm the removal of the name.
4. Alternatively, clicking the secondary mouse button while a naming context is selected will bring up a context dialog box, as

shown in Figure 26. This allows the deletion of the selected object binding.

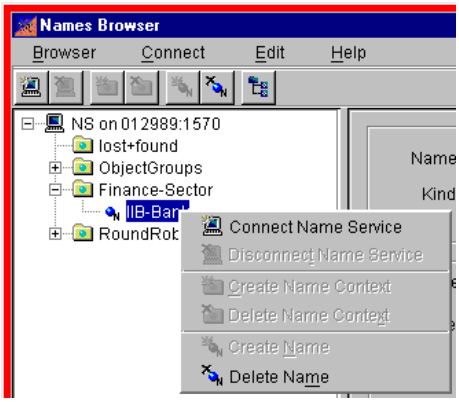

Figure 26 Context-Sensitive Object Binding Dialog

Navigating the OrbixNames Browser Button Bar

The OrbixNames Browser includes a number of "button bar" tool icons that allow quick access to Naming Service functions.

Icon	Description
	Connect to a Naming Service host.
	Disconnect from the selected Naming Service host.
	Create a naming context.
	Delete a naming context.
	Create an object binding.
	Delete an object binding.
	Refresh the naming tree.

Part V

OrbixNames Programmer's Reference

In this part

This part contains the following:

CosNaming

Synopsis

The `CosNaming` module, defined in the OrbixNames file `NamingService.idl`, contains all IDL definitions for the CORBA Naming Service and some definitions specific to Orbix. To access standard Naming Service functionality, use the `NamingContext` and `BindingIterator` interfaces defined in this module. These interfaces are described in detail in "CosNaming::NamingContext" on page 121, and "CosNaming::BindingIterator" on page 119.

This chapter describes data types, other than the interfaces `NamingContext` and `BindingIterator`, defined directly within the scope of the `CosNaming` module.

IDL

```
// IDL
module 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;

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,CannotProceed,InvalidName,AlreadyBound);
  void destroy () raises (NotEmpty);
  void list (in unsigned long how_many,
     out BindingList bl,out BindingIterator bi);
  Object resolve_object_group (in Name n)
   raises (NotFound, CannotProceed, InvalidName);
  Object OBfactory();
};

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

CosNaming::Binding

Synopsis

```
struct Binding {
  Name binding_name;
  BindingType binding_type;
};
```

Description

When browsing a naming graph in the Naming Service, an application can list the contents of a given naming context, and determine the name and type of each binding in it. To do this, the application calls the operation `CosNaming::NamingContext::list()` on the target `NamingContext` object. This operation returns a list of `Binding` structures, each structure representing a single binding in the naming context.

A `Binding` structure contains two member fields:

`binding_name`	The full compound name of the binding.
`binding_type`	The binding type, indicating whether the name is bound to an application object or a naming context.

Notes

CORBA compliant.

See Also

```
CosNaming::BindingList
CosNaming::BindingType
CosNaming::NamingContext::list()
```

CosNaming::BindingList

Synopsis

```
typedef sequence<Binding> BindingList;
```

Description

A value of this type contains a set of `Binding` structures, each of which represents a single name binding. An application can list the bindings in a given naming context using the `CosNaming::NamingContext::list()` operation, as described in the entry for CosNaming::Binding. An `out` parameter of this operation returns a value of type `BindingList`.

Notes

CORBA compliant.

See Also

```
CosNaming::Binding
CosNaming::BindingType
CosNaming::NamingContext::list()
```

CosNaming::BindingType

Synopsis

```
enum BindingType {nobject, ncontext};
```

Description

There are two types of name binding in the CORBA Naming Service: names bound to application objects, and names bound to naming contexts. Names bound to application objects cannot be used in a compound name, except as the last element in that name. Names bound to naming contexts can be used as any component of a compound name and allow you to construct a naming graph in the Naming Service.

The enumerated type `BindingType` represents these two forms of name bindings. This type has two possible values:

`nobject`	Describes a name bound to an application object.
`ncontext`	Describes a name bound to a naming context in the Naming Service.

Name bindings created using `CosNaming::NamingContext::bind()` or `CosNaming::NamingContext::rebind()` are `nobject` bindings. Name bindings created using the operations `CosNaming::NamingContext::bind_context()` or `CosNaming::NamingContext::rebind_context()` are `ncontext` bindings.

Notes

CORBA compliant.

See Also

```
CosNaming::Binding
CosNaming::BindingList
```

CosNaming::Istring

Synopsis

```
typedef string Istring;
```

Description

Type `Istring` is a place holder for an internationalized string format, which might be added to the CORBA Naming Service definitions by the OMG.

Notes

CORBA compliant.

CosNaming::Name

Synopsis

```
typedef sequence<NameComponent> Name;
```

Description

A `Name` represents the name of an object in the Naming Service. If the object name is defined within the scope of one or more naming contexts, the name is a compound name. For this reason, type `Name` is defined as a sequence of name components.

Two names that differ only in the contents of the `kind` field of one `NameComponent` structure are considered to be different names.

Names with no components, that is sequences of length zero, are illegal.

Notes

CORBA compliant.

See Also

`CosNaming::NameComponent`

CosNaming::NameComponent

Synopsis

```
struct NameComponent {
   Istring id;
   Istring king;
};
```

Description

A `NameComponent` structure represents a single component of a name associated with an object in the Naming Service. This structure has two fields:

`id`	An identifier that corresponds to the name of the component.
`kind`	An element that adds secondary type information to the component name.

The `id` field is intended for use purely as an identifier. The semantics of the `kind` field are application-specific and the Naming Service makes no attempt to interpret this value.

A name component is uniquely identified by the combination of both `id` and `kind` fields. Two name components that differ only in the contents of the `kind` field are considered to be different components.

Notes

CORBA compliant.

See Also

`CosNaming::Name`

CosNaming::BindingIterator

Synopsis

The operation `CosNaming::NamingContext::list()` allows you to obtain a list of bindings in a naming context. As described in "CosNaming::NamingContext" on page 121, this operation allows you to specify a maximum number of bindings to be returned. To provide access to all other bindings in the naming context, the operation returns an object of type `CosNaming::BindingIterator`.

A `CosNaming::BindingIterator` object stores a list of name bindings and allows you to access the elements of this list.

IDL

```
// IDL
module CosNaming {
  ...

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

See Also

`CosNaming::Binding`
`CosNaming::BindingList`
`CosNaming::NamingContext::list()`

CosNaming::BindingIterator::destroy()

Synopsis

```
void destroy ();
```

Description

The `destroy()` operation deletes the `CosNaming::BindingIterator` object on which it is called.

Notes

CORBA compliant.

CosNaming::BindingIterator::next_n()

Synopsis

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

Description

The `next_n()` operation returns the next `how_many` elements in the list of bindings, subsequent to the last element returned by a call to `next_n()` or `next_one()`. If less than `how_many` elements remain in the list, all the remaining elements are returned.

Parameters

`how_many`	The maximum number of bindings to be returned in parameter `bl`.
`bl`	The returned list of name bindings.

Return Value

Returns `true` if one or more bindings are returned in parameter `bl`, returns `false` if no more bindings remain.

Notes

CORBA compliant.

See Also

`CosNaming::BindingIterator::next_one()`

CosNaming::BindingIterator::next_one()

Synopsis

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

Description

The `next_one()` operation returns the next element in the list of bindings, subsequent to the last element returned by a call to `next_n()` or `next_one()`.

Parameters

`b` The returned name binding.

Return Value

Returns `true` if a binding is returned in parameter `b`, returns `false` if no more bindings remain.

Notes

CORBA compliant.

See Also

`CosNaming::BindingIterator::next_n()`

CosNaming::NamingContext

Synopsis

The interface `CosNaming::NamingContext` provides the operations that allow you to access the main features of the CORBA Naming Service, such as binding and resolving names. This interface also includes the Orbix-specific operations `OBfactory()` and `resolve_object_group()`, which you call when using the load balancing features of OrbixNames described in the chapters "Load Balancing with OrbixNames Using C++" or "Load Balancing with OrbixNames Using Java".

IDL

```
// IDL
module CosNaming {
  ...

  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, CannotProceed,
      InvalidName, AlreadyBound);
    void destroy () raises (NotEmpty);
```

```
    void list (in unsigned long how_many,
         out BindingList bl, out BindingIterator
bi);
    Object resolve_object_group (in Name n)
      raises (NotFound, CannotProceed,
InvalidName);
    Object OBfactory();
  };

  ...
};
```

Notes CORBA compliant.

See Also `CosNaming`

CosNaming::NamingContext::AlreadyBound

Synopsis

```
exception AlreadyBound {};
```

Description If an application calls an operation that attempts to bind a name to an object or naming context, but the specified name has already been bound, the operation raises an exception of type `AlreadyBound`.

The following operations can raise this exception:

```
CosNaming::NamingContext::bind()
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::bind_new_context()
```

Notes CORBA compliant.

CosNaming::NamingContext::bind()

Synopsis

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

Description The operation `bind()` creates a name binding, relative to the target naming context, between a name and an object. If the name passed to this operation is a compound name with more than one component, all except the last component are used to find the sub-context in which to add the name binding. The contexts associated with these components must already exist, otherwise the operation raises a `NotFound` exception.

Parameters

`n`	The name to be bound to the target object, relative to the naming context on which the operation is called.
`obj`	The application object to be associated with the specified name.

Notes CORBA compliant.

See Also

```
CosNaming::NamingContext::AlreadyBound
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::NotFound
CosNaming::NamingContext::rebind()
CosNaming::NamingContext::resolve()
```

CosNaming::NamingContext::bind_context()

Synopsis

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

Description

The `bind_context()` operation creates a binding, relative to the target naming context, between a name and another, specified naming context. This new binding can be used in any subsequent name resolutions: the entries in naming context `nc` can be resolved using compound names.

All but the final naming context specified in parameter `n` must already exist. This operation raises an `AlreadyBound` exception if the name specified by `n` is already in use.

The naming graph built using `bind_context()` is not restricted to being a tree: it can be a general naming graph in which any naming context can appear in any other naming context.

Parameters

`n` The name to be bound to the target naming context, relative to the naming context on which the operation is called.

`nc` The `NamingContext` object to be associated with the specified name. This object must already exist. To create a new `NamingContext` object, call `CosNaming::NamingContext::new_context()`.

Notes

CORBA compliant.

See Also

`CosNaming::NamingContext::AlreadyBound`
`CosNaming::NamingContext::bind_new_context()`
`CosNaming::NamingContext::CannotProceed`
`CosNaming::NamingContext::InvalidName`
`CosNaming::NamingContext::new_context()`
`CosNaming::NamingContext::NotFound`
`CosNaming::NamingContext::rebind_context()`
`CosNaming::NamingContext::resolve()`

CosNaming::NamingContext::bind_new_context()

Synopsis

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

Description

The operation `bind_new_context()` creates a new `NamingContext` object in the Naming Service and binds the specified name to it, relative to the naming context on which the operation is called. This operation has the same effect as a call to `CosNaming::NamingContext::new_context()` followed by a call to `CosNaming::NamingContext::bind_context()`.

The new name binding created by this operation can be used in any subsequent name resolutions: the entries in the returned naming context can be resolved using compound names.

All but the final naming context specified in parameter `n` must already exist. This operation raises an `AlreadyBound` exception if the name specified by `n` is already in use.

Parameters

`n`	The name to be bound to the newly created naming context, relative to the naming context on which the operation is called.

Return Value Returns a reference to the newly created `NamingContext` object.

Notes CORBA compliant.

See Also

```
CosNaming::NamingContext::AlreadyBound
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::new_context()
CosNaming::NamingContext::NotFound
```

CosNaming::NamingContext::CannotProceed

Synopsis

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

Description If a Naming Service operation fails due to an internal error, the operation raises a `CannotProceed` exception. However, the application might be able to use the information returned in this exception to complete the operation later. For example, if you use a Naming Service federated across several hosts and one of these hosts is currently unavailable, a Naming Service operation might fail until that host is available again.

A `CannotProceed` exception includes two member fields:

`cxt`	The `NamingContext` object associated with the component at which the operation failed.
`rest_of_name`	The remainder of the compound name, after the binding for the component at which the operation failed.

The following operations can raise this exception:

```
CosNaming::NamingContext::bind()
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::bind_new_context()
CosNaming::NamingContext::rebind()
CosNaming::NamingContext::rebind_context()
CosNaming::NamingContext::resolve()
CosNaming::NamingContext::resolve_object_group()
CosNaming::NamingContext::unbind()
```

Notes CORBA compliant.

See Also

```
CosNaming::Name
CosNaming::NamingContext
```

CosNaming::NamingContext::destroy()

Synopsis

```
void destroy ()
  raises (NotEmpty);
```

Description

The operation `destroy()` deletes the `NamingContext` object on which it is called. Before deleting a `NamingContext` in this way, ensure that it contains no bindings. If you call `destroy()` on a `NamingContext` that contains existing bindings, the operation raises a `CosNaming::NamingContext::NotEmpty` exception.

To avoid leaving name bindings with no associated objects in the Naming Service, call `CosNaming::NamingContext::unbind()` to unbind the context name before calling `destroy()`. See the entry for CosNaming::NamingContext::resolve() for information about the result of resolving names of context objects that no longer exist.

Notes

CORBA compliant.

See Also

```
CosNaming::NamingContext::NotEmpty
CosNaming::NamingContext::resolve()
CosNaming::NamingContext::unbind()
```

CosNaming::NamingContext::InvalidName

Synopsis

```
exception InvalidName {};
```

Description

If an operation receives an `in` parameter of type `CosNaming::Name` for which the sequence length is zero, the operation raises an `InvalidName` exception.

The following operations can raise this exception:

```
CosNaming::NamingContext::bind()
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::bind_new_context()
CosNaming::NamingContext::rebind()
CosNaming::NamingContext::rebind_context()
CosNaming::NamingContext::resolve()
CosNaming::NamingContext::resolve_object_group()
CosNaming::NamingContext::unbind()
```

Notes

CORBA compliant.

CosNaming::NamingContext::list()

Synopsis

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

Description

The operation `list()` returns a list of the name bindings in the naming context on which the operation is called. The parameter `how_many` specifies the maximum number of bindings that should be returned in the `BindingList` parameter, `bl`.

The `BindingList` parameter is a sequence of `Binding` structures where each `Binding` indicates the name and type of the binding—the type indicates whether the name is that of an object, or whether it is the name of a node in the naming graph which participates in name resolution.

If the naming context contains more than the requested number (`how_many`) of bindings, the `list()` operation returns a `BindingIterator` which contains the remaining bindings. This is returned in parameter `bi`. If the naming context does not contain any additional bindings, the parameter `bi` is a nil object reference.

Parameters

`how_many`	The maximum number of bindings to be returned in parameter `bl`.
`bl`	A list of at most `how_many` bindings contained in the naming context on which the operation is called.
`bi`	A `BindingIterator` object that provides access to all remaining bindings contained in the naming context on which the operation is called.

Notes CORBA compliant.

See Also
```
CosNaming::BindingIterator
CosNaming::BindingList
```

CosNaming::NamingContext::new_context()

Synopsis
```
NamingContext new_context ();
```

Description The operation `new_context()` creates a new `NamingContext` object in the Naming Service, without binding a name to it. After you create a naming context with this operation, you can bind a name to it by calling `CosNaming::NamingContext::bind_context()`.

Return Value Returns a reference to the newly created `NamingContext` object. There is no relationship between this object and the `NamingContext` object on which you call the operation.

Notes CORBA compliant.

See Also
```
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::bind_new_context()
```

CosNaming::NamingContext::NotEmpty

Synopsis
```
exception NotEmpty {};
```

Description An application can call the operation `CosNaming::NamingContext::destroy()` to delete a naming context object in the Naming Service. For this operation to succeed, the naming context must contain no bindings. If bindings exist in the naming context, the operation raises a `NotEmpty` exception.

Notes CORBA compliant.

CosNaming::NamingContext::NotFound

Synopsis

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

Description

Several operations in the interface `CosNaming::NamingContext` require an existing name binding to be passed as an `in` parameter. If such an operation receives a name binding that it determines is invalid, the operation raises a `NotFound` exception. This exception contains two member fields:

`why`	The reason why the name binding is invalid. See CosNaming::NamingContext::NotFoundReason for more details.
`rest_of_name`	The remainder of the compound name following the component that the operation determined to be invalid.

The following operations can raise this exception:

```
CosNaming::NamingContext::bind()
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::bind_new_context()
CosNaming::NamingContext::rebind()
CosNaming::NamingContext::rebind_context()
CosNaming::NamingContext::resolve()
CosNaming::NamingContext::resolve_object_group()
CosNaming::NamingContext::unbind()
```

Notes

CORBA compliant.

See Also

`CosNaming::NamingContext::NotFoundReason`

CosNaming::NamingContext::NotFoundReason

Synopsis

```
enum NotFoundReason {missing_node, not_context,
not_object};
```

Description

If an operation raises a `NotFound` exception, a value of enumerated type `NotFoundReason` indicates the reason why the exception was raised:

`missing_node`	A component of the name passed to the operation did not exist in the Naming Service.
`not_context`	The operation expected to receive a name bound to a naming context, for example using `CosNaming::NamingContext::bind_context()`, but the name received did not satisfy this requirement.
`not_object`	The operation expected to receive a name bound to an application object, for example using `CosNaming::NamingContext::bind()`, but the name received did not satisfy this requirement.

Notes

CORBA compliant.

See Also

`CosNaming::NamingContext::NotFound`

CosNaming::NamingContext::OBfactory()

Synopsis

```
Object OBfactory ();
```

Description

The operation `OBfactory()` returns a reference to the object group factory in the Naming Service. Before using the returned object, narrow it to type `LoadBalancing::ObjectGroupFactory`. You can then use this object to create new object groups and to find existing groups, as described in the chapters "Load Balancing with OrbixNames Using C++" or "Load Balancing with OrbixNames Using Java".

Return Value

Returns a reference to the object group factory. To use this object reference, first narrow it to type `LoadBalancing::ObjectGroupFactory`.

Notes

OrbixNames specific.

See Also

```
LoadBalancing
LoadBalancing::ObjectGroup
LoadBalancing::ObjectGroupFactory
```

CosNaming::NamingContext::rebind()

Synopsis

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

Description

The operation `rebind()` creates a binding between a name that is already bound in the target naming context and an object. The previous name is unbound and the new binding is created in its place. As is the case with `CosNaming::NamingContext::bind()`, all but the last component of a compound name must exist, relative to the naming context on which you call the operation.

Parameters

`n`	The name to be bound to the specified object, relative to the naming context on which the operation is called.
`obj`	The application object to be associated with the specified name.

Notes

CORBA compliant.

See Also

```
CosNaming::NamingContext::bind()
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::NotFound
CosNaming::NamingContext::resolve()
```

CosNaming::NamingContext::rebind_context()

Synopsis

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

Description

The `rebind_context()` operation creates a binding between a name that is already bound in the context on which the operation is called, and a naming context. The previous name is unbound and the new binding is made in its place. As is the case for `CosNaming::NamingContext::bind_context()`, all but the last component of a compound name must name an existing `NamingContext`.

Parameters

`n` The name to be bound to the specified naming context, relative to the naming context on which the operation is called.

`nc` The naming context to be associated with the specified name.

Notes

CORBA compliant.

See Also

```
CosNaming::NamingContext::bind_context()
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::NotFound
CosNaming::NamingContext::resolve()
```

CosNaming::NamingContext::resolve()

Synopsis

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

Description

The `resolve()` operation returns the object reference bound to the specified name, relative to the naming context on which the operation was called. The first component of the specified name is resolved in the target naming context.

The return type is IDL `Object`, which maps to type `CORBA::Object_ptr` in C++ or to type `org.omg.CORBA.Object` in Java. You must narrow the result to the appropriate type before using it in your application.

If the name `n` refers to a naming context, it is possible that the corresponding `NamingContext` object no longer exists in the Naming Service. For example, this could happen if you call `CosNaming::NamingContext::destroy()` to destroy a context without first unbinding the context name. In this case, `resolve()` raises a CORBA system exception.

Parameters

`n` The name to be resolved, relative to the naming context on which the operation is called.

Return Value

Returns a reference to the object associated with the specified name.

Notes

CORBA compliant.

See Also

```
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::NotFound
CosNaming::NamingContext::resolve_object_group()
```

CosNaming::NamingContext::resolve_object_group()

Synopsis

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

Description

The operation `resolve_object_group()` returns the `LoadBalancing::ObjectGroup` object associated with a name binding. Before using the returned object, narrow it to type `LoadBalancing::ObjectGroup`. You can then use this object to manipulate the contents of the object group, as described in the chapters "Load Balancing with OrbixNames Using C++" or "Load Balancing with OrbixNames Using Java".

The required `LoadBalancing::ObjectGroup` object must already exist and the specified name must be bound to it. To create a `LoadBalancing::ObjectGroup` object, first call the operation `OBfactory()` on a naming context to create a `LoadBalancing::ObjectGroupFactory` object, then use this object to create the required type of object group.

If the name passed to `resolve_object_group()` is bound to an object that is not of type `LoadBalancing::ObjectGroup`, the operation returns the associated object reference. However, if you then attempt to narrow this object to type `LoadBalancing::ObjectGroup`, the narrow operation will fail.

Parameters

`n` The name bound to the required object group, relative to the naming context on which the operation is called.

Return Value

Returns a reference to the object group to which the specified name is bound. To use this object reference, first narrow it to type `LoadBalancing::ObjectGroup`.

Notes

OrbixNames specific.

See Also

```
LoadBalancing
LoadBalancing::ObjectGroup
```

CosNaming::NamingContext::unbind()

Synopsis

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

Description

The operation `unbind()` removes the binding between a specified name and the object associated with it. Unbinding a name does not delete the application object or naming context object associated with the name. For example, to remove a naming context completely from the Naming Service, you should first unbind the corresponding name, then delete the `NamingContext` object by calling `CosNaming::NamingContext::destroy()`.

Parameters

`n` The name to be unbound in the Naming Service, relative to the naming context on which the operation is called.

Notes

CORBA compliant.

See Also

```
CosNaming::NamingContext::CannotProceed
CosNaming::NamingContext::destroy()
CosNaming::NamingContext::InvalidName
CosNaming::NamingContext::NotFound
```

LoadBalancing

Synopsis

The module `LoadBalancing`, defined in the OrbixNames file `LoadBalancing.idl`, provides access to the load balancing features of OrbixNames described in the chapters "Load Balancing with OrbixNames Using C++" or "Load Balancing with OrbixNames Using Java". The definitions in this module are specific to OrbixNames.

There are four IDL interfaces in the module `LoadBalancing`: `ObjectGroup`, `ObjectGroupFactory`, `RandomObjectGroup`, and `RoundRobinObjectGroup`. This chapter describes all data types defined directly within the scope of the `LoadBalancing` module, other than these four interfaces. These four interfaces are described in detail in subsequent chapters.

IDL

```
// IDL
module LoadBalancing {
  exception no_such_member{};
  exception duplicate_member{};
  exception duplicate_group{};
  exception no_such_group{};

  typedef string memberId;
  typedef sequence<memberId> memberIdList;

  struct member {
    Object obj;
    memberId id;
  };

  typedef string groupId;
  typedef sequence<groupId> groupList;

  interface ObjectGroup;
  interface RoundRobinObjectGroup;
  interface RandomObjectGroup;

  interface ObjectGroupFactory {
    RoundRobinObjectGroup createRoundRobin (in
groupId id)
      raises (duplicate_group);
    RandomObjectGroup createRandom (in groupId id)
      raises (duplicate_group);
    ObjectGroup findGroup (in groupId id)
      raises (no_such_group);
    groupList rr_groups();
    groupList random_groups();
  };

  interface ObjectGroup {
    readonly attribute string id;
    Object pick();
    void addMember (in member mem)
      raises (duplicate_member);
    void removeMember (in memberId id)
      raises (no_such_member);
    Object getMember (in memberId id)
      raises (no_such_member);
    memberIdList members();
```

```
    void destroy();
  };

  interface RandomObjectGroup : ObjectGroup {};
  interface RoundRobinObjectGroup : ObjectGroup
{};
};
```

See Also

`CosNaming::NamingContext::OBfactory()`
`CosNaming::NamingContext::resolve_object_group()`

LoadBalancing::no_such_group

Synopsis

```
exception no_such_group {};
```

Description

The operation `LoadBalancing::ObjectGroupFactory::findGroup()` returns a reference to a specified object group. This operation takes the group identifier as an `in` parameter and then searches for the group in the Naming Service. If no group exists for the specified identifier, the operation raises a `no_such_group` exception.

Notes

OrbixNames specific.

LoadBalancing::no_such_member

Synopsis

```
exception no_such_member {};
```

Description

An operation that finds or removes an existing member of an object group takes a member identifier as an `in` parameter. In such cases, the identifier must correspond to an existing group member. If it does not, the operation raises a `no_such_member` exception.

The following operations can raise this exception:

```
LoadBalancing::ObjectGroup::getMember();
LoadBalancing::ObjectGroup::removeMember();
```

Notes

OrbixNames specific.

LoadBalancing::duplicate_group

Synopsis

```
exception duplicate_group {};
```

Description

An operation that creates an object group takes the new group identifier as a parameter. If the group identifier is already used in the Naming Service, the operation raises a `duplicate_group` exception.

The following operations can raise this exception:

```
LoadBalancing::ObjectGroupFactory::createRandom();
LoadBalancing::ObjectGroupFactory::createRoundRobi
n();
```

Notes

OrbixNames specific.

LoadBalancing::duplicate_member

Synopsis

```
exception duplicate_member {};
```

Description

The operation `LoadBalancing::ObjectGroup::addMember()` adds a member to an object group. This operation takes a parameter that specifies the object to be added to the group, and the member identifier to be associated with the object. If the member identifier

is already used in the group, the operation raises a `duplicate_member` exception.

Notes OrbixNames specific.

LoadBalancing::groupId

Synopsis

```
typedef string groupId;
```

Description Each object group has an associated identifier, of type `groupId`. The format of this identifier is application specific and is not specified by OrbixNames. However, the identifier for each group must be unique within the Naming Service.

Notes OrbixNames specific.

See Also `LoadBalancing::groupList`

LoadBalancing::groupList

Synopsis

```
typedef sequence<groupId> groupList;
```

Description The operations `LoadBalancing::ObjectGroupFactory::random_groups()` and `LoadBalancing::ObjectGroupFactory::rr_groups()` allow you to obtain a list of object groups in the Naming Service. These operations return a list of group identifiers, as type `groupList`.

Notes OrbixNames specific.

See Also

```
LoadBalancing::groupId
LoadBalancing::ObjectGroupFactory::random_groups()
LoadBalancing::ObjectGroupFactory::rr_groups()
```

LoadBalancing::member

Synopsis

```
struct member {
  Object obj;
  memberId id;
};
```

Description An object group contains a set of member objects. For each object in the group, the group maintains a reference to the object and an identifier that is unique within the group. This information is stored in a `member` structure.

A `member` structure contains two fields:

`obj` A reference to the member object.

`id` The member identifier for the object. This value must be unique within the object group.

Notes OrbixNames specific.

See Also `LoadBalancing::memberId`

LoadBalancing::memberId

Synopsis

```
typedef string memberId;
```

Description

Each object reference in an object group has an associated member identifier, of type `memberId`. The format of this identifier is application specific and is not specified by OrbixNames. However, each member identifier must be unique within a given object group.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::member
LoadBalancing::memberIdList
```

LoadBalancing::memberIdList

Synopsis

```
typedef sequence<memberId> memberIdList;
```

Description

The operation `LoadBalancing::ObjectGroup::members()` returns a list of the member identifiers in a given object group. This list is returned as type `memberIdList`, which is a sequence of `memberId` values.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::memberId
LoadBalancing::ObjectGroup::members()
```

LoadBalancing::ObjectGroup

Synopsis

The interface `LoadBalancing::ObjectGroup` allows you to manage the contents of an existing object group. This interface is usually accessed in server applications.

This interface also supports the operation `pick()`, which OrbixNames calls when a client resolves a name bound to an object group. This operation selects a member of the group in accordance with the group selection algorithm.

The interfaces `LoadBalancing::RandomGroup` and `LoadBalancing::RoundRobinGroup` inherit this interface.

IDL

```
// IDL
module LoadBalancing {
  ...

  interface ObjectGroup {
    readonly attribute string id;

    Object pick();
    void addMember (in member mem)
      raises (duplicate_member);
    void removeMember (in memberId id)
      raises (no_such_member);
    Object getMember (in memberId id)
      raises (no_such_member);
    memberIdList members();
    void destroy();
  };

  ...
};
```

See Also

`CosNaming::NamingContext::resolve_object_group()`
`LoadBalancing::ObjectGroupFactory`
`LoadBalancing::RandomObjectGroup`
`LoadBalancing::RoundRobinObjectGroup`

LoadBalancing::ObjectGroup::addMember()

Synopsis

```
void addMember (in member mem)
  raises (duplicate_member);
```

Description

An Orbix server calls the operation `addMember()` to add a member object to a group. This operation takes an `in` parameter, of type `member`, that specifies the member identifier and provides a reference to the object. The member identifier must not already exist in the object group on which the operation is called. If the identifier exists, `addMember()` raises a `duplicate_member` exception.

Parameters

`mem` A structure containing a reference to the new member object and the member identifier.

Notes

OrbixNames specific.

See Also

`LoadBalancing::member`

LoadBalancing::ObjectGroup::destroy()

Synopsis

```
void destroy ();
```

Description

Calling operation `destroy()` on an object group completely removes that group from the Naming Service. It is not necessary to remove the members of a group before calling `destroy()`.

Operation `destroy()` does not affect the name binding associated with the group. Before calling `destroy()`, call `CosNaming::NamingContext::unbind()` to remove the associated name binding.

Notes

OrbixNames specific.

See Also

`CosNaming::NamingContext::unbind()`

LoadBalancing::ObjectGroup::getMember()

Synopsis

```
Object getMember (in memberId id)
  raises (no_such_member);
```

Description

An application calls the operation `getMember()` to obtain a reference to a specific member object in an object group. This operation takes the member identifier as an `in` parameter, of type `memberId`. If this identifier does not correspond to an object in the group on which `getMember()` is called, the operation raises a `no_such_member` exception.

Parameters

`id`	The identifier of the member object for which an object reference is required.

Return Value

Returns a reference to the object associated with the specified member identifier.

Notes

OrbixNames specific.

See Also

`LoadBalancing::memberId`

LoadBalancing::ObjectGroup::id

Synopsis

```
readonly attribute string id;
```

Description

This attribute stores the identifier of the object group. The format of this identifier is application specific and is not specified by OrbixNames. However, the group identifier must be unique within the Naming Service.

Notes

OrbixNames specific.

LoadBalancing::ObjectGroup::members()

Synopsis

```
memberIdList members ();
```

Description

The operation `members()` returns a list of all members in the group on which it is called. Only the identifier for each member is returned. To obtain a reference to a member object associated with a specific identifier, call the operation `LoadBalancing::ObjectGroup::getMember()`.

Return Value

Returns a list of identifiers of all members in the object group.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::memberIdList
LoadBalancing::ObjectGroup::getMember()
```

LoadBalancing::ObjectGroup::pick()

Synopsis

```
Object pick();
```

Description

The operation `pick()` selects a member of an object group and returns a reference to the member object. In a round-robin selection object group, the operation `pick()` implements a round-robin selection algorithm to choose a member of the object group. In a random selection object group the operation `pick()` randomly chooses a member of the group.

When a client resolves a Naming Service name that has been bound to an object group, OrbixNames calls operation `pick()` to determine which member object the name should resolve to.

Return Value

Returns a reference to the object selected by OrbixNames.

Notes

OrbixNames specific.

LoadBalancing::ObjectGroup::removeMember()

Synopsis

```
void removeMember (in memberId id) raises
(no_such_member);
```

Description

An Orbix server calls the operation `removeMember()` to remove a member object from a group. This operation takes an `in` parameter, of type `memberId`, which specifies the identifier of the member object to be removed. If this identifier does not correspond to an object in the group on which `removeMember()` is called, the operation raises a `no_such_member` exception.

Parameters

`id` The identifier of the member to be removed.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::memberId
```

LoadBalancing::ObjectGroupFactory

Synopsis

The interface `LoadBalancing::ObjectGroupFactory` allows you to create object groups and find existing groups in the Naming Service. To obtain a reference to a `LoadBalancing::ObjectGroupFactory`, call `CosNaming::NamingContext::OBfactory()` on any `CosNaming::NamingContext` object.

IDL

```
// IDL
module LoadBalancing {
  ...

  interface ObjectGroupFactory {
    RoundRobinObjectGroup createRoundRobin (in
groupId id)
      raises (duplicate_group);
    RandomObjectGroup createRandom (in groupId id)
      raises (duplicate_group);
    ObjectGroup findGroup (in groupId id)
      raises (no_such_group);
    groupList rr_groups();
    groupList random_groups();
  };

  ...
};
```

See Also

`CosNaming::NamingContext::OBfactory()`
`LoadBalancing::ObjectGroup`

LoadBalancing::ObjectGroupFactory::createRandom()

Synopsis

```
RandomObjectGroup createRandom (in groupId id)
  raises (duplicate_group);
```

Description

This operation creates a new object group. When OrbixNames calls the operation `LoadBalancing::ObjectGroup::pick()` to choose a member from the resulting group, a random selection algorithm is used.

The operation `createRandom()` takes a group identifier as an `in` parameter. This identifier must be unique within the Naming Service. If an existing group is already associated with this identifier, the operation raises a `LoadBalancing::duplicate_group` exception.

Parameters

`id` The group identifier for the new object group. This value must be unique within the Naming Service.

Return Value

Returns a reference to the `RandomObjectGroup` object for the newly created group.

Notes

OrbixNames specific.

See Also

`LoadBalancing::duplicate_group`
`LoadBalancing::groupId`
`LoadBalancing::RandomObjectGroup`

LoadBalancing::ObjectGroupFactory::createRoundRobin()

Synopsis

```
RoundRobinObjectGroup createRoundRobin (in groupId
id)
  raises (duplicate_group);
```

Description

This operation creates a new object group. When OrbixNames calls the operation `LoadBalancing::ObjectGroup::pick()` to choose a member from the resulting group, a round-robin selection algorithm is used.

The operation `createRoundRobin()` takes a group identifier as an `in` parameter. This identifier must be unique within the Naming Service. If an existing group is already associated with this identifier, the operation raises a `LoadBalancing::duplicate_group` exception.

Parameters

`id` The group identifier for the new object group. This value must be unique within the Naming Service.

Return Value

Returns a reference to the `RoundRobinObjectGroup` object for the newly created group.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::duplicate_group
LoadBalancing::groupId
LoadBalancing::RoundRobinObjectGroup
```

LoadBalancing::ObjectGroupFactory::findGroup()

Synopsis

```
ObjectGroup findGroup (in groupId id)
  raises (no_such_group);
```

Description

An application calls the operation `findGroup()` to obtain a reference to a specific object group. This operation takes the group identifier as an `in` parameter, of type `groupId`. If this identifier does not correspond to an existing object group in the Naming Service, the operation raises a `no_such_group` exception.

Parameters

`id` The group identifier for the required object group.

Return Value

Returns a reference to the `ObjectGroup` object for the required group.

Notes

OrbixNames specific.

See Also

```
LoadBalancing::groupId
LoadBalancing::no_such_group
```

LoadBalancing::ObjectGroupFactory::random_groups()

Synopsis

```
groupList random_groups ();
```

Description

The operation `random_groups()` returns a list of all random groups that currently exist in the Naming Service. Only the group identifiers are returned. To obtain a reference to a group associated with a specific identifier, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`.

Return Value

Returns a list of the identifiers of all random groups in the Naming Service.

Notes

OrbixNames specific.

See Also

`LoadBalancing::groupList`
`LoadBalancing::ObjectGroupFactory::findGroup()`

LoadBalancing::ObjectGroupFactory::rr_groups()

Synopsis

```
groupList rr_groups ();
```

Description

The operation `rr_groups()` returns a list of all round-robin groups that currently exist in the Naming Service. Only the group identifiers are returned. To obtain a reference to a group associated with a specific identifier, call the operation `LoadBalancing::ObjectGroupFactory::findGroup()`.

Return Value

Returns a list of the identifiers of all round-robin groups in the Naming Service.

Notes

OrbixNames specific.

See Also

`LoadBalancing::groupList`
`LoadBalancing::ObjectGroupFactory::findGroup()`

LoadBalancing::RandomObjectGroup

Synopsis

The interface `LoadBalancing::RandomObjectGroup` represents an object group in which OrbixNames applies a random selection algorithm when choosing a member object. This interface is a simple specialization of `LoadBalancing::ObjectGroup`, and adds no new attributes or operations.

IDL

```
// IDL
module LoadBalancing {
  ...

  interface RandomObjectGroup : ObjectGroup {
  };
};
```

See Also

`LoadBalancing::ObjectGroup`
`LoadBalancing::ObjectGroup::pick()`
`LoadBalancing::RoundRobinObjectGroup`

LoadBalancing::RoundRobinObjectGroup

Synopsis

The interface `LoadBalancing::RoundRobinObjectGroup` represents an object group in which OrbixNames applies a round-robin selection algorithm when choosing a member object. This interface is a simple specialization of `LoadBalancing::ObjectGroup`, and adds no new attributes or operations.

IDL

```
// IDL
module LoadBalancing {
  ...

  interface RoundRobinObjectGroup : ObjectGroup {
  };
};
```

See Also

`LoadBalancing::ObjectGroup`
`LoadBalancing::ObjectGroup::pick()`
`LoadBalancing::RandomObjectGroup`

Part VI

Appendices

In this part

This part contains the following:

Configuration Variables

There are two forms of Orbix configuration variables: those that are common to multiple Orbix products and variables that are specific to OrbixNames only.

Common Configuration Variables

You can set the following variables using the Configuration Explorer GUI tool, or by editing the `common.cfg` configuration file, or as environment variables.

Variable	Description
`IT_DAEMON_PORT`	TCP port number for the Orbix daemon.
`IT_DAEMON_SERVER_BASE`	The starting TCP port number for servers launched by the Orbix daemon.
`IT_DAEMON_SERVER_RANGE`	The number set in this variable is used together with that set in `IT_DAEMON_SERVER_BASE` to determine the range of port numbers available for Orbix servers.
`IT_IMP_REP_PATH`	The full path name of the Implementation Repository directory.
`IT_INT_REP_PATH`	The full path name of the Interface Repository directory.
`IT_LOCAL_DOMAIN`	The name of the local internet domain; for example, `ABigBank.com`.
`IT_LOCATOR_PATH`	The full path name of the directory holding the locator files.

OrbixNames-Specific Configuration Variables

You can set the following variables using the Configuration Explorer GUI tool, or by editing the `orbixnames3.cfg` configuration file, or as environment variables:

Variable	Description
`IT_NAMES_HOME`	This variable specifies the full path to the `bin` directory of your Orbix installation.
`IT_NAMES_IP_ADDR`	By default, a call to `CORBA::ORB::resolve_initial_reference("NameService")` expects the location of the OrbixNames server to be specified in the Orbix locator configuration files. You can also specify the IP address of the server host by setting the variable `IT_NAMES_IP_ADDR`. This value overrides the Orbix locator. If this value is set, `IT_USE_HOSTNAME_IN_IOR` must be set to `false`.
`IT_NS_PORT`	By default, an application contacts the OrbixNames server using the port number defined in the Orbix `IT_DAEMON_PORT` configuration variable. However, if the OrbixNames server uses another port, you can override `IT_DAEMON_PORT` by setting the value of `IT_NS_PORT`.
`IT_NAMES_REPOSITORY_PATH`	This variable specifies the path name to the Bindings Repository. The Bindings Repository is a persistent repository of name bindings maintained by the Naming Service. The results of all update operations, such as `bind()`, `rebind()`, and `bind_new_context()`, are committed to the Bindings Repository. An alternative approach is to use the '`-r`' flag of the naming service executable. This flag also specifies a Bindings Repository and overrides `IT_NAMES_REPOSITORY_PATH`.
`IT_NAMES_SERVER`	By default, a call to `CORBA::ORB::resolve_initial_reference("NameService")` expects an OrbixNames server to be registered in the Implementation Repository with the name `NS`. If this variable is set, `resolve_initial_references()` searches for an OrbixNames server with the name specified.
`IT_NAMES_SERVER_HOST`	By default, a call to `CORBA::ORB::resolve_initial_reference("NameService")` expects the location of the OrbixNames server to be specified in the Orbix locator configuration files. You can also specify the server host name by setting the variable `IT_NAMES_SERVER_HOST`. This value overrides the Orbix locator. If this value is set, `IT_USE_HOSTNAME_IN_IOR` must be set to `true`.
`IT_USE_HOSTNAME_IN_IOR`	When OrbixNames stores an IOR in the Bindings Repository, the host on which the object runs is embedded in the IOR. If `IT_USE_HOSTNAME_IN_IOR` is set to `true`, the name of the host is embedded in the IOR; if it is set to `false`, the IP address is embedded. The default setting is `true`.

Variable	Description
IT_NS_HASH_TABLE_SIZE	This variable specifies the size of the hash table associated with each naming context to store references to bindings. By default, this variable is set to 23. You can also alter this value when executing the OrbixNames server using the -h <hash table size> flag.
IT_NS_HASH_TABLE_SIZE_LOAD_FACTOR	This variable specifies the factor by which the hash table associated with a naming context is increased to when full.
IT_NAMES_TIMEOUT	This specifies the amount of time, in seconds, that the server may remain idle before timing out. The default value is -1, or infinite. This means that the server does not time out. You can also alter this value when executing the OrbixNames server using the -t <timeout> flag.
IT_NAMES_DIAGNOSTICS	This variable specifies the diagnostic level used by Orbix within the naming service. The default value is 0, with a maximum value of 255.
IT_NAMES_THREAD_POOL_SIZE	This variable sets the size of the thread pool used to handle incoming requests to the multi-threaded OrbixNames server. The default value of this variable is 11. You can also alter this value when executing the OrbixNames server using the -p <thread pool size> flag.
IT_NAMES_CACHE_SIZE	This variable sets the number of naming contexts that should be cached in memory by the OrbixNames server. The default value of this variable is 10. You can also alter this value when executing the OrbixNames server using the -e <cache size> flag.
IT_SSL_IIOP_LISTEN_PORT	This variable sets the port number that the secure OrbixNames server listens on.

Note

Entries in Orbix configuration files are scoped with a prefix; for example, Common.IT_DAEMON_PORT or OrbixNames.IT_NAMES_REPOSITORY_PATH. Environment variables are not scoped.

For further details of Orbix-specific configuration variables, refer to the C++ or Java edition of the ***Orbix Administrator's Guide***.

Index

D

E

F

G

H

I

K

L

M

N

O

P

R

S

T

U

V

X

Z