HP Object Oriented DCE C++ Class Library Programmer's Guide

HEWLETT PACKARD ®

Printed in U.S.A.

June 1994

E0694

First Edition

Preface

This document describes how to use Hewlett-Packard’s Object Oriented Distributed Computing Environment (OODCE) C++ class library to develop object-based client/server applications.

This document is intended to be used by an applications programmer who has experience with OSF’s DCE software, and the C++ and C programming languages.

This manual is organized as follows:

Chapter 1	“Introduction” provides an general description of OODCE, including the DCE programming model and the OODCE development model.
Chapter 2	“The Basics” provides information on a sample application and OODCE’s **`idl++`** compiler.
Chapter 3	“Using the Cell Directory Service” describes how to place bindings in the Cell Directory Service and how to register server information in remote procedure call (RPC) groups and profiles.
Chapter 4	“Error Handling” describes how to communicate errors between a server and a client.
Chapter 5	“Basic Pthreads” describes the basics of using Pthread objects.
Chapter 6	“Basic Security” describes how to use the basic security features offered by the OODCE library.
Chapter 7	“Basic Access Control List Management” describes how to use access control lists to provide security in OODCE.
Chapter 8	“Advanced Application Development” describes advanced features of OODCE for developing object-based systems.

Chapter 9	"Advanced ACL Management" describes how to use OODCE to write your own ACL database, instantiate a persistent database, and use multiple reference monitors.
Chapter 10	"Advanced Thread Programming" describes how to use thread attributes, thread-specific storage, and condition variables.
Appendix A	"OODCE Glossary of Terms" defines OODCE terms used in this manual.
Appendix B	"Basic Application Development Summary" provides a step-by-step guide to OODCE application development.

Related Documentation

The following HP OODCE documentation is supplied with this release.

HP DCE Application Development Tools Installation Notes (B2922-90601) provides hardcopy notes that describe how to install the HP OODCE software and access the online help.

HP OODCE Online Help System consists of the following help volumes:

- *HP OODCE Release Notes* provides information on contents, prerequisites, defects and limitations. It also provides hyperlinked access to all other OODCE help volumes.
- *HP OODCE Sample Applications* provides reference, tutorial, and background information on the Sleeper sample application.

- *HP OODCE Application Development Tools* provides an overview of the **idl++** compiler, a description of **idl++** generated files, a quick start for the compiler, pointers to reference information, an overview of the Tracing and Logging Facility, feature descriptions, and reference information on tracing and logging.
- *HP OODCE Class Reference* gives an overview of OODCE classes, a description of class hierarchy, and pointers to reference information (man pages) for all classes.

To access online help from the HP VUE Front Panel:

1. Click the Help icon on the VUE Front Panel (the ? icon). A *Welcome to Help Manager* help window appears.
2. In the Help Manager window, click the *HP Distributed Application Development Tools* product family title. A list of the help volumes appears.
3. To display one of the HP Tools help volumes, click its title.

To access help from a shell, enter this command at a shell prompt:

```
/usr/vue/bin/helpview -h HPtoolswelcome
```

This displays the *Welcome Help* help volume. The help volume contains hyperlinks to all of the other HP volumes.

Related OSF Documents

For additional information on the Distributed Computing Environment, see the following documents:

- *Introduction to OSF DCE*
- *OSF DCE User's Guide and Reference*
- *OSF DCE Application Development Guide*
- *OSF DCE Application Development Reference*
- *OSF DCE Release Notes.*

Object Oriented Programming Documents

For information on object oriented programming, see the following documents:

- Booch, Grady. *Object-Oriented Design: With Applications*. New York: Benjamin/Cummings Publishing Co. Inc. 1991.
- Coleman, D., Arnold, P., Bodoff, S., Dollin, C., Gilchrist, H., Hayes, F., and Jeremaes, P. *Object-Oriented Development: The Fusion Method*. New York: Prentice-Hall. 1993.
- Jacobson, Ivar. *Object-Oriented Software Engineering: a Use Case Driven Approach*. New York: Addison-Wesley. 1992.
- Mellor, Stephen and Shlaer, Sally. *Object-Oriented Systems Analysis*. New York: Prentice-Hall. 1992.
- Rumbaugh, James; Blaha, Michael; Premerlani, William; Eddy, Frederick; and Lorensen, William. *Object-Oriented Modeling and Design*. New York: Prentice-Hall. 1991.
- Taylor, David A. *Object-Oriented Information Systems: Planning and Implementation*. New York: John Wiley & Sons. 1992.

- Wirfs-Brock, R., Wilkerson, B., and Wierner, L. *Designing Object-Oriented Software*. New York: Prentice-Hall.

C++ Programming Documents

For information on C++ programming, see the following documents:

- Coplien, Jim. *Advanced C++; Programming Styles and Idioms*. New York: Addison-Wesley. 1992.
- Lippman, Stanley B. *C++ Primer*. 2nd Ed. New York: Addison-Wesley. 1989.
- Meyers, Scott. *Effective C++*. New York: Addison-Wesley. 1992.
- Murray, Robert B. *C++ Strategies and Tactics*. New York: Addison-Wesley. 1993.
- Stroustrup, Bjarne. *The C++ Programming Language*. 2nd Ed. New York: Addison-Wesley. 1991.

Hewlett Packard Documents

For information on programming with threads and the Distributed Computing Environment, see the following documents:

- *Programmer's Notes on HP DCE Threads* (B3190-90002)
- *Introduction to OSF DCE* (B3190-9005, ISBN 0-13-490624-1)
- *OSF Application Development Guide* (B3190-90006, ISBN 0-13-643826-1)
- *OSF Application Development Reference* (B3190-90007, ISBN 0-13-643834-2)

Contents

Contents

Contents

1

Introduction to OODCE

Distributed processing is a major paradigm shift for application developers. The ability to develop complex systems where processing is distributed across many different computing platforms is both very attractive and daunting. It is attractive because the distributed computing model provides system planners and developers with flexibility and power. It is daunting because of the problems inherent in combining large scale systems with different hardware and systems software.

With the success of the Open Software Foundation's (OSF) Distributed Computing Environment (DCE), however, the difficulties of designing and implementing distributed systems have been greatly reduced. DCE provides a solid foundation for the development of distributed heterogeneous systems. It provides a powerful communications mechanism capable of translating data between machines with different architectures and a set of basic services that include:

- Security
- Location service
- Time service
- Distributed file system.

Many DCE users, however, have asked for even simpler models, interfaces, and methods of using it. The C applications programming interfaces (API's) were, for them, too low level. They required that applications developers learn a substantial amount about DCE when creating even the simplest applications. In addition, many users have been moving towards object-oriented analysis and design, and using the **C++** programming language to implement their systems. These customers began requesting an API for DCE that was both easier to use and better supported the **C++** programming language. In response, we developed the Object Oriented DCE (OODCE) product.

The following sections describe the integration and use of the **C++** programming language and the Distributed Computing Environment from the Open Software Foundation. The focus is on using object oriented programming techniques to simplify developing distributed applications over DCE and the exploitation of the distributed programming model that exists within the DCE.

DCE

The OSF’s Distributed Computing Environment represents a collection of integrated services and tools that provide a basic infrastructure for developing distributed applications. Many companies contributed technologies to DCE. These companies and others are now offering the DCE infrastructure running on both open and proprietary operating platforms.

The services provided by DCE are as follows:

Remote Procedure Call	The Remote Procedure Call (RPC) mechanism provides the communications layer over which the other DCE services communicate. In particular, DCE RPC enables programs to call remote procedures that execute in other processes on a network.
Directory Service	A hierarchical directory service that can provide a scalable namespace across geographic boundaries. Currently, RPC uses the directory service to locate and bind to instances of server processes within DCE.
Security Service	The security service provides two basic functions: • An authentication mechanism that validates users at login time and provides mutual authentication between clients and servers operating in DCE • A privilege service that provides protected (secure) security attribute information for security principals that can be used to implement an authorization scheme based on access control lists.

Distributed Time Service	Provides time synchronization between networked hosts within DCE.
Distributed File System	Provides secure access to file data across a DCE administrative unit called a **cell**.
Threads Service	Since much DCE depends on a multi-threading capability, DCE provides a user space implementation of POSIX threads API (IEEE Standard 1003.4a) that can be used with operating systems that do not provide their own threading mechanism.

In DCE, all of these services are integrated and form a standard, well defined infrastructure on which to build distributed applications. A basic set of tools is also provided to support developing DCE-based applications and managing the DCE services.

The `C++` Language

`C++` is an object-oriented programming language that supports the objected oriented development paradigm. This paradigm is well suited to the development of large scale systems because it can clearly define and specify the behavior of small individual components in easy to understand terms. Once defined and implemented, these “Software ICs” can be assembled in many different ways to create larger components. This approach reuses not only the implementation itself, but also the specification and analysis effort that was invested in that implementation. The resulting gain in productivity is potentially great.

The advent of the integrated circuit freed engineers from dealing with the vagaries of individual transistor circuits and permitted them to deal with circuits at a much higher level. This, in turn, brought down development costs and made many products commercially feasible for the mass market.

Distributed systems can also benefit from the abstraction that the object-oriented paradigm supports. The definition of behaviors in terms that relate to the object itself (such as Write a Check, or Debit an Account) tends to keep implementation detail out of specifications and analyses. This makes it easier to provide different implementations for the same object, even on different machines and architectures.

Product Objectives

The primary objective of the Object Oriented DCE product is to reduce the cost of developing DCE-based distributed systems. To do this, OODCE:

- Simplifies the development of DCE based systems.
- Provides an environment for developing `C++` based systems that maintain the `C++` development model; that is, clients and servers can both be written in `C++`. Useful distributed systems can be implemented without resorting to the C language API.
- Lives within and exploits the object model already provided by the DCE RPC service. The definition of a new distributed object model is not the intent of this product.
- Provides useful class abstractions that represent conceptual entities within DCE that hide complexity and provide correct default behavior.

The main objectives center around enhancing the DCE development model and easing the learning curve. In essence, DCE applications developers need no longer concern themselves with the details of DCE itself when they use OODCE. Like electronics engineers today, they need only to use the “OODCE chips” provided and concern themselves only with the application they are writing.

The next section explains the DCE programming model in more detail, then provides a top level overview of the OODCE representation of that model.

The DCE Programming Model

DCE is not an object-based system; however, the RPC mechanism supports an object model. This section explains the DCE programming model and develops a basic set of **`C++`** classes that can be used to exploit that model.

DCE Objects, Types, and Interfaces

DCE provides a number of features that can be considered object oriented. While these features are not fully exploited in the C usage model, their full power can be brought to the forefront when examining DCE from the object oriented viewpoint. Central to the DCE programming model is its support of objects that export a set of well defined interfaces. These objects can be typed, and are managed by a larger object called a server. Servers are responsible for:

- Making the objects locatable and accessible by remote clients
- Enforcing the security policy defined for the objects
- Performing other management tasks as defined by the system designer.

An OODCE object is an entity that is manipulated by a set of well defined operations. Systems can be envisioned in which a server manages thousands of objects. Because of the potentially large number of objects, the DCE designers had to develop a new method of object identification, the Universally Unique Identifier (UUID).

A UUID is a large number that is generated and is guaranteed to be unique for all time across all DCE cells in the world. Each object can be assigned a UUID so that it can be located and used by remote clients.

Operations that can be performed on a DCE object are grouped into logical sets called interfaces. DCE interfaces define the calling syntax that is used by both the requestor (DCE client) and the provider (DCE object) of an operation. The specification of an interface is independent of the mechanism used to convey requests between the requester and provider and the method by which the operations are provided or implemented. Specification of DCE interfaces use an Interface Definition Language (IDL) to define the operation signatures.

To support change and enhancement, DCE interfaces support the concept of versioning. DCE interface versions are identified by a major number and a minor number. An increase in the minor number signifies a compatible upgrade to the interface, that is, interfaces that share the same major number and have a higher minor number are fully compatible with interfaces that have a lower minor number. On the other hand, interfaces that have different major numbers are not considered compatible. Because the version of an interface is so significant, DCE interface names include both a UUID and its version.

DCE objects that share characteristics can be typed. DCE types associate objects with code that implements them. Like other DCE entities, UUIDs are used to name types. Object types can be used to locate groups or categories of objects rather than a specific object. For example, a client that wants to print to a laser printer can search for objects that have a “laser printer” type.

A DCE manager is a specific implementation of a type. Objects of the same type within a server process usually share the same implementation. However, each instance of a DCE object of a particular type usually maintains its own state information. The same type, however, may need to be implemented using a different manager on a machine with a different architecture or from a different vendor.

DCE objects are accessed via a server process that is said to export interface information on behalf of the objects it supports. The server process can be thought of as an object manager that makes the objects it supports accessible across a network or process boundary via the DCE interfaces they support. The server process listens for incoming client requests. These requests get dispatched to the object of the correct type that implements the requested interface and operation.

The following figure shows the basic programming model implemented by DCE.

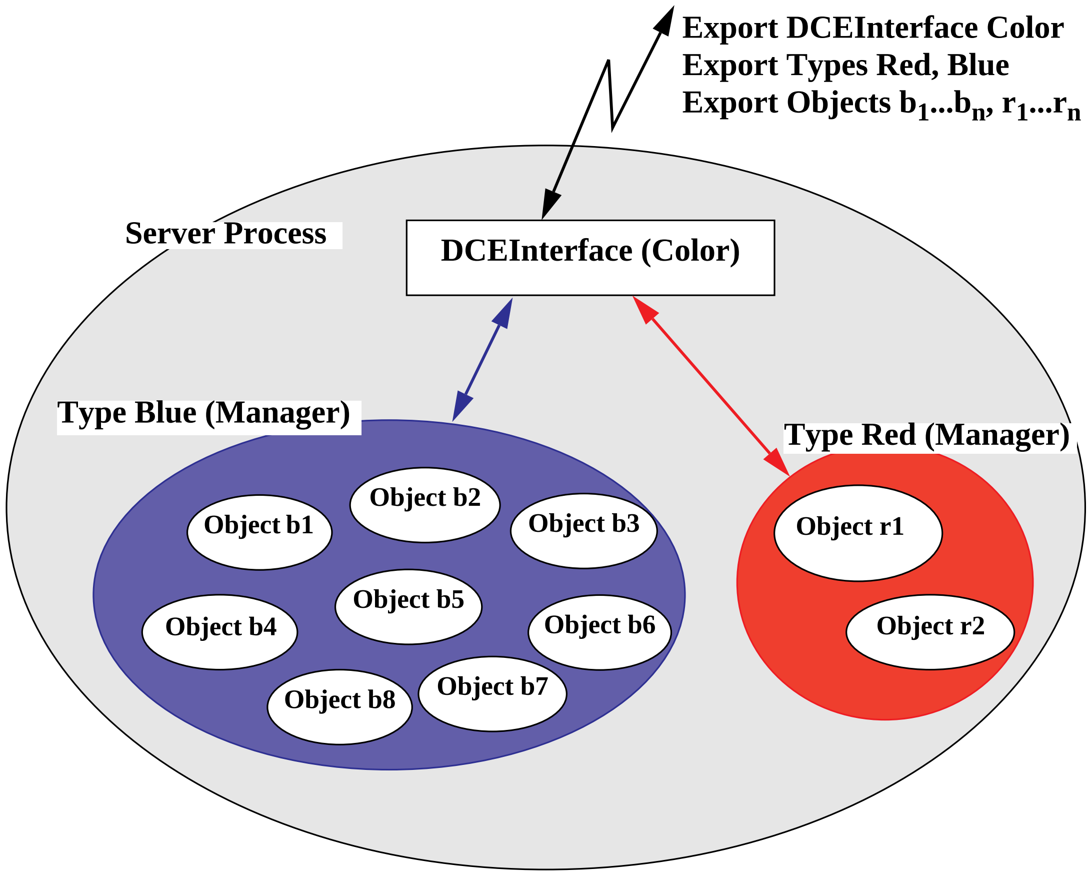

Figure 1-1. The DCE Programming Model

Mapping C++ Onto the DCE Programming Model

From the description of the DCE programming model in the previous section, a corresponding object model suitable for **C++** development can be derived. OODCE maps the DCE entities into two slightly different views for the **C++** developer: one from the viewpoint of a server, the other from the viewpoint of a client.

On the server side, the DCE notion of an interface is mapped onto the **C++** notion of an abstract class. The operations defined in DCE interfaces become pure virtual member functions of this abstract class. The DCE notion of type maps onto concrete **C++** classes, derived from the abstract interface class. DCE managers are the implementation of these concrete classes. DCE objects then map to instances of the **C++** objects (or collections of **C++** objects) that are constructed from these concrete classes. The following figure shows a representation of the OODCE programming model from the viewpoint of a server.

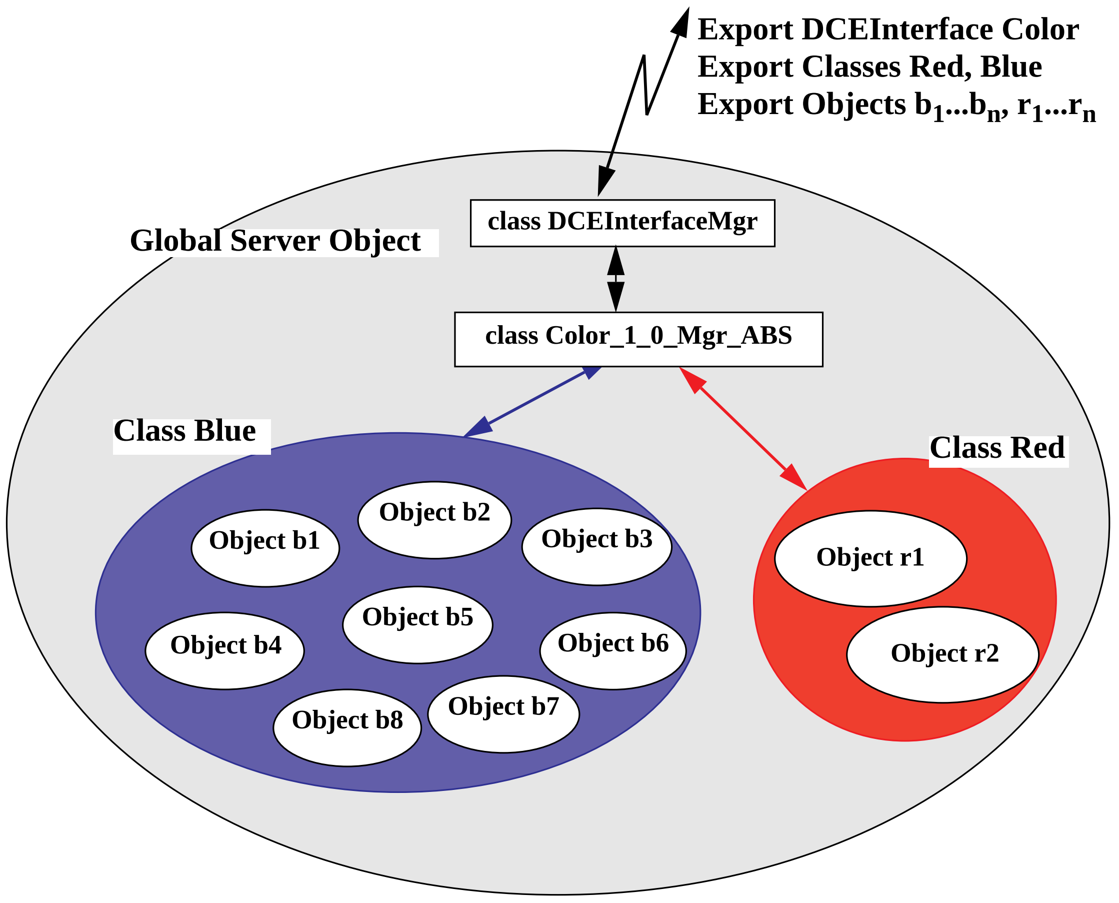

Figure 1-2. The OODCE Server Side Programming Model

On the client side, the DCE interface maps onto a concrete **C++** class that provides version control and accesses remote operations via member functions of the class. Using the facilities provided by OODCE, the DCE client has no control over implementation details on the server, so the DCE type of an object is lost. The applications developer can insert "IsA" type operations into the interface definition to obtain the object's type directly from the object, and, using parameters in calls to a factory object, can determine the class of an object when it is created.

The following figure shows a representation of the OODCE programming model from the viewpoint of a client.

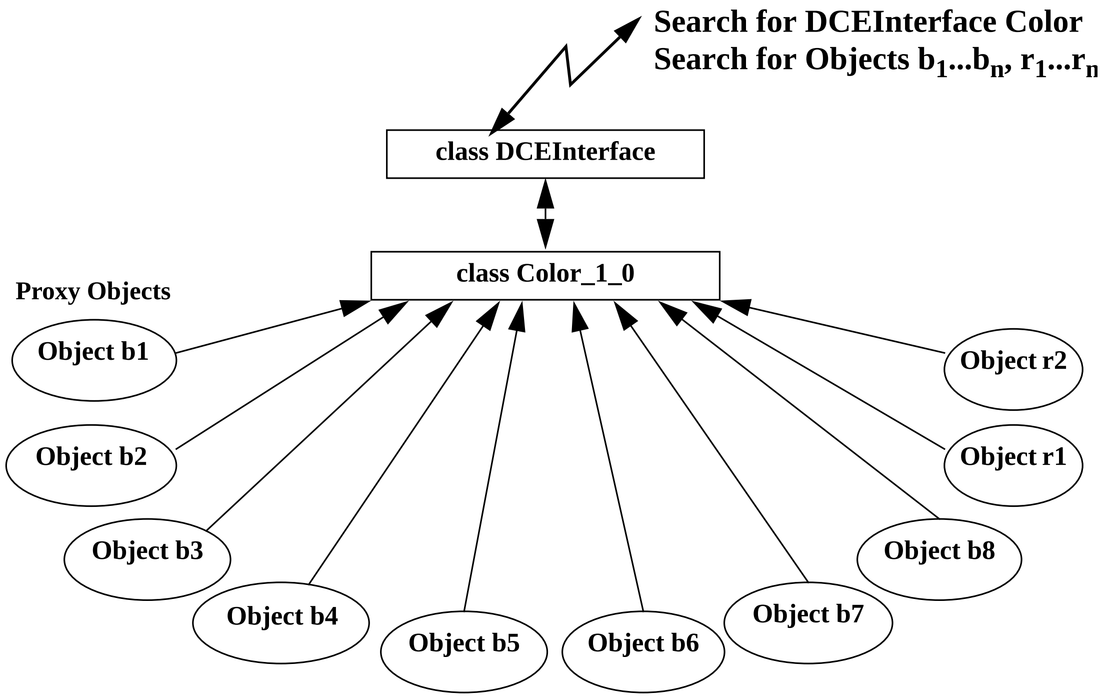

Figure 1-3. The OODCE Client Side Programming Model

A client proxy object locates and binds to a DCE object that is implemented at the server by a specific **C++** object instance. There is a direct mapping between a DCE object UUID and a **C++** object instance (or a set of **C++** objects). A specific client object acts as a proxy for a specific **C++** object on the server that implements the same interface and provides the illusion that the client side code is accessing a local **C++** object.

NOTE

This illusion can only be maintained in as much as DCE can support it. Unexpected and unrecoverable communication failures are always a possibility and need to be dealt with by the client when they happen.

Local member function calls on the client are translated into remote method calls that are sent to the server process that manages the matching DCE object. On the server, requests from a client are sent to a specific **C++** object as a local member function call. Results are returned in a similar manner.

The OODCE class library maintains this mapping though specific system level objects and a tool that generates the required classes from an IDL specification.

The DCE and OODCE Error Models

DCE provides two methods to detect errors that occur during remote procedure calls: status codes and exceptions. Status codes are the predominant method used to return error information. The use of exceptions within DCE is relatively sparse and appears to be inconsistent with the status code model. In addition, the threads component of DCE appears to have a slightly different error model than the rest of the system, complicating development for DCE applications developers. The exception handling in **C++** provides a very convenient and extensible model for error handling that is particularly suited to library code. Therefore, the different DCE error models have been mapped into a single, **C++** exception-based error model.

The result is a very consistent and extensible error model for DCE development. Each DCE status code is given a corresponding **C++** exception class. The exception classes are arranged in a hierarchy that is rooted in a single base class. Wrapper code is provided around the DCE error model such that **C++** developers need only deal with **C++** exceptions when calling facilities in the class library. The existence of a single, extensible error model also encourages developers to use exceptions in their own **C++** code to handle fault conditions.

Security Model

The `C++` language does not support security. DCE, however, provides facilities for mutual authentication, communications security, and access control. To provide access to this functionality from the `C++` development environment, the DCE security model is provided through the definition of system classes. To facilitate development, base classes are provided. Because many details of a security policy are application specific, some of these classes are abstract, providing only interface details rather than full implementations.

The classes provide a framework within which security can be implemented and the DCE security services are exploited. Although the class library provides little in the way of actual implementation of security models, it does provide a mechanism by which basic security checks can be automatically invoked. Security implementations coded to the provided system class can be automatically called to do security checks before the object that is the target of the request is invoked. As much security checking as possible should be done outside of an object implementation to allow for consistency and portability of the object code.

DCE Reference Monitors

OODCE supports the concept of a reference monitor by defining an abstract class for it. A reference monitor performs a basic set of security checks before a remote method call is passed to an object. A reference monitor can check to ensure that the client is properly authenticated, the protection level is appropriate, and that the client is authorized to use the requested interface or operation. OODCE supports the specification of a reference monitor object separately for each interface, but many interfaces can share a single reference monitor object. A separate reference monitor object can be specified for the server management functions supported directly by the global server object within the OODCE class library. OODCE supplies a basic implementation of a usable reference monitor; however, an applications developer can create a custom reference monitor to enforce the security policy to meet system requirements.

Access Control Lists

The primary means for storing and accessing authorization information in DCE is Access Control Lists (ACLs). ACLs match DCE identities with sets of permissions or capabilities. A match can be made on the basis of any field in a DCE identity including the principal ID, group ID, and organization ID, and whether or not the identity is foreign (comes from another cell).

NOTE OODCE does not support foreign identities.

Permissions or capabilities are defined on an application basis, and usually imply that the identity they are associated with is granted (or denied) the ability to perform one or more operations.

To support ACLs, OODCE provides a set of abstract classes and a reference implementation for developers to use.

Login Context Management

OODCE provides abstract classes and a set of simple implementations to establish and maintain a DCE identity. These classes can be built upon by customers requiring elaborate schemes or security measures. In many cases, the provided classes are adequate for prototyping and initial development.

Naming in OODCE

OODCE fully supports the DCE naming scheme at the DCE Remote Procedure Call (RPC) Naming Service Interface (NSI) level. The class library supports methods and optional parameters for interfaces and objects that cause them to be registered in the endpoint map and optionally in the directory service. Once registered by the class library, the names can be used as parameters in the client-side object constructor calls. OODCE will use the names to locate and bind to the server-side object. A set of utility classes define methods for the various naming entities to include objects, groups, profiles and names.

Threads Support in OODCE

Because thread construction, monitoring, and control is not defined in the **C++** standard, OODCE provides a set of classes to perform these functions for the POSIX threads (Pthread) model. The classes permit **C++** developers to create new threads by using a class constructor, modifying the attributes of the executing thread and other threads, and constructing and using mutex locks and condition variables. Some abstraction is provided in the implementation of these classes. For example, mutex lock objects are automatically initialized when created. Developers do not need to make a separate call to initialize them as they do when using the C API.

The OODCE Development Model

The development model supported by OODCE is one in which **C++** is used as the primary development language. The basic DCE development model is maintained through the use of the **idl++** compiler and the OODCE class library.

Development of an OODCE object starts with the specification of an interface using the DCE Interface Definition language (IDL). The IDL specification describes the remote procedures and the data syntax used between a client and a server.

OODCE IDL++ Compilation and Results

The IDL specification is passed to the **idl++** compiler. The **idl++** compiler generates the required communication stubs for inter-object communication via the specified interface. The **idl++** compiler also generates class information and code that interfaces with the DCE runtime to provide a **C++** environment for the developer.

The following figure shows the information that the **`idl++`** compiler generates.

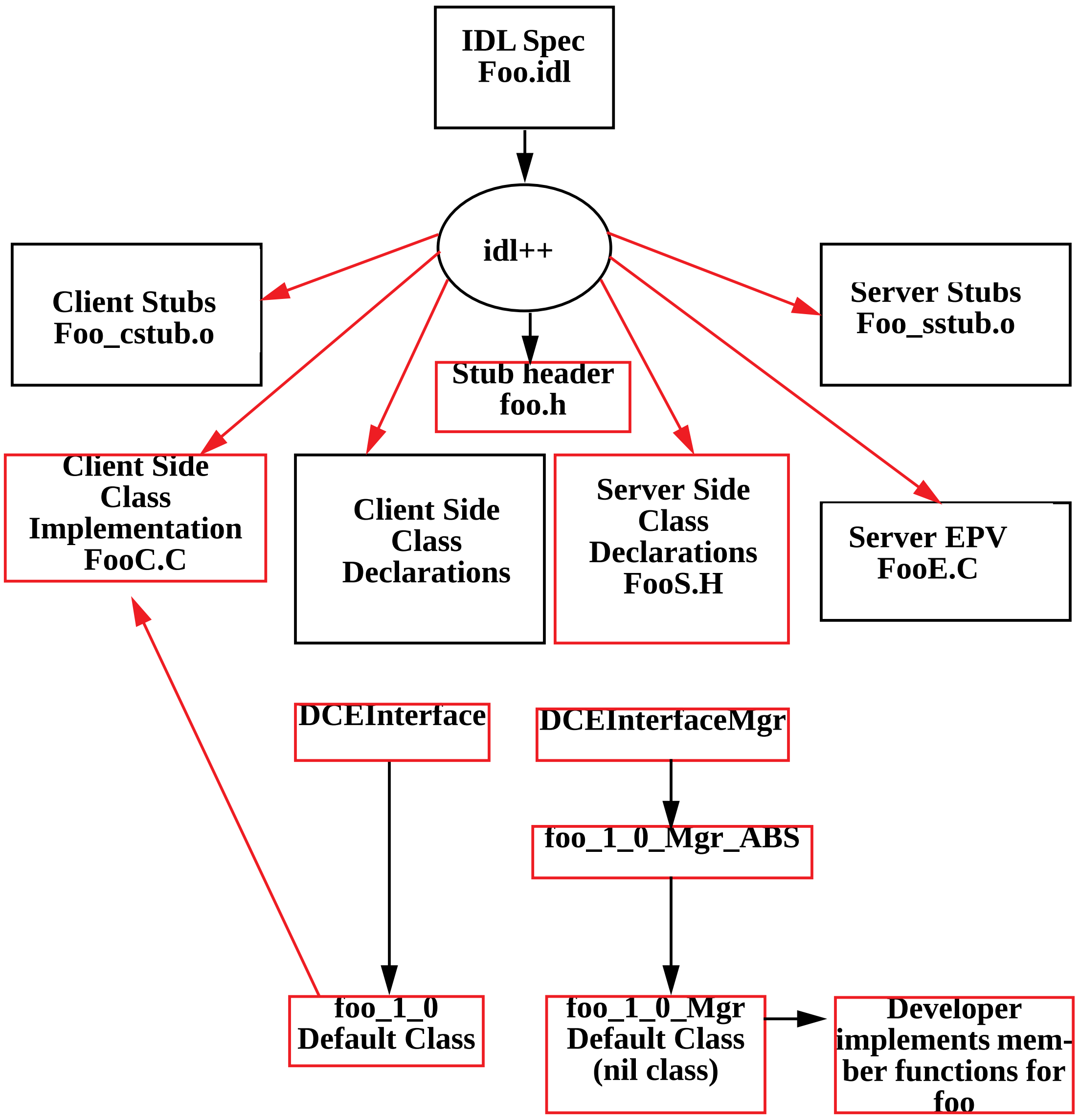

Figure 1-4. IDL to C++ Mapping

Server Side Class Definitions

The **`idl++`** compiler generates an abstract class definition from the IDL specification. The name of this abstract class is based on the interface name and version specified in the IDL specification. In the previous figure, the IDL specifies version 1.0 of interface foo, therefore the abstract class name is **`foo_1_0_Mgr_ABS`**.

The remote procedures defined in the IDL specification are represented as pure virtual member functions of this abstract class. The abstract class is derived from a common base class called **`DCEInterfaceMgr`**, which specifies a common interface for all server-side classes generated by the **`idl++`** compiler.

The **`idl++`** compiler also generates a definition of a concrete class that is derived from the abstract class. This class represents the nil or default DCE class for the IDL specification. The name of this class is also derived from the interface name and version, for example, **`foo_1_0_Mgr`** is the name of the concrete class in the preceding example. Other concrete classes can be derived from the abstract class that can provide different implementations for the same interface. Objects can provide more than one interface by inheriting from either the abstract or concrete class for each interface to be supported.

Both the abstract and concrete class definitions are emitted in a file whose name is based on the input file name, similar to the standard **`idl`** compiler. The name of the emitted file is the name of the input file, stripped of its suffix, with the suffix “**`S.H`**” appended to it. In the preceding example, **`Foo.idl`** is the name of the input file, therefore **`FooS.H`** is the name of the file that contains the server class definitions.

Server Entrypoint Vector

In addition to the class definitions, the **`idl++`** compiler generates a server entrypoint vector (EPV) for the interface. The EPV generated by the **`idl++`** compiler provides the mapping between the C calls made by the server stub and the **`C++`** method calls provided by the class library and the developer. An incoming client request must be mapped into a member function call on a specific **`C++`** object instance. The EPV locates the correct **`C++`** object and makes the requested member function call.

Optional security checks can be made before the member function call. These checks call implementations of security classes that implement the security policy for the object. The EPV also handles the translation of server side exceptions into a network transmittable form for communication to the client. When processing is complete, the member function returns to the EPV, which provides any output data to the DCE runtime for transmission back to the client-side object.

The EPV is emitted in a file whose name is also derived from that of the input file. In this case, the suffix of the target is “**`E.C`**”. In the example shown in Figure 1-4, where the name of the input file is **`Foo.idl`**, the server EPV file is named **`FooE.`**C.

Server Stubs

The **`idl++`** compiler generates a server stub file that contains the code to do the following:

- Receive incoming procedure calls from remote clients
- Unmarshal the parameters for IN parameters
- Dispatch the call to the correct function to perform the operation
- Marshals any OUT parameters
- Transmits any OUT parameters or exceptions back to the client.

In OODCE, the function called from the server stub is in the server EPV so that the correct **`C++`** object can be located and additional processing can be performed before the method is called.

Client-Side Class Definitions

The **`idl++`** compiler generates a concrete class on the client side that is used to access objects that implement the interface defined in the IDL file. Remote procedures declared in the IDL specification are mapped onto member functions in the generated client class. The client class is derived from a base abstract class, **`DCEInterface`**, that forms the root class of all client-side classes derived from IDL files. The name of the client-side concrete class is based on the name of the interface and its version.

In the example shown in Figure 1-4, the client-side concrete class is **`foo_1_0`**. The name of the file in which the class definition is emitted is based on the name of the input file. In this case, the definitions are placed in a file named **`FooC.H`**.

The implementation for the client-side concrete class is also generated by the **`idl++`** compiler. This implementation provides functionality to locate and bind to any remote object that implements the IDL specification. The implementation of the member function of the concrete class provides access to the remote procedure calls that communicate with the server object. The implementation is emitted in a file with the "**`C.C`**" suffix; therefore, in the example shown in Figure 1-4, the name of the file that contains the concrete class implementation is **`FooC.C`**.

Application code that requires access to a remote DCE object simply constructs an instance of the concrete class (supplying optional location information if desired) and makes member function calls to that object. All member function calls are passed by the proxy object to the remote object to which it is bound for processing. Communication, server-side exceptions and errors, and security failures are modelled as exceptions and need to be caught by the application code that uses the object. Checks made by the client object ensure type safeness between the client and server object.

Server Functionality

The **`idl++`** compiler generates the environment that allows the implementation of, and access to IDL interfaces through **`C++`** objects. However, these objects can only be located and accessed if there is a server that exports them. Servers interact with the DCE sub-systems to advertise their objects and to listen for incoming client requests.

Server functionality is embodied in a **`Server`** class defined within the class library. A global object called **`theServer`** can be used to register objects with the DCE environment and then to listen for client requests. The **`Server`** class provides default implementations for the basic duties of a DCE server.

Specific implementations can be provided by overriding the default implementations of the server class through **`C++`** inheritance.

Typically, server code involves the registration of object implementations with a server object, initialization of server preferences for naming and security, and then instructing the server object to listen for client requests.

2

The Basics

The first step of OODCE development is to define the required interfaces for communication between a client and a server. In OODCE, the server exports interfaces to objects within the server, which can be called by a client. The server exports interfaces that can be called by a client.

An OODCE object is an entity that is manipulated by well-defined operations. Every OODCE object has a class that specifies the type or category of the object. The OODCE objects of a class are manipulated using a specific set of one or more interfaces. An OODCE interface consists of a set of related operations that can be applied to any object of the class.

DCE interfaces are defined using the DCE Interface Definition Language (IDL). They are then processed by a compiler called **`idl,`** which generates system data structures and communication stubs for the client and server. The IDL language is described in the *DCE Application Development Guide*.

OODCE uses the same IDL language, but an enhanced version of the compiler is used to process the interface specification. The compiler program is called **`idl++.`** The **`idl++`** compiler generates the client and server stub and header files needed for a DCE Remote Procedure Call (RPC) interface. **`idl++`** also generates a number of **`C++`** files that provide communication between OODCE clients and servers. The following example demonstrates these files and their purposes.

The following code defines an IDL specification for a **`sleeper`** object. A **`sleeper`** object can be told to sleep for a specified period of time. To accomplish this, the **`sleeper`** object exports a single method call called **`Sleep`**. The **`Sleep`** method accepts one argument, which is the number of seconds to sleep. The caller of this method is blocked for the requested period of time, until the **`Sleep`** method returns. The code is as follows:

```
[uuid(),
version(1.0)
] interface sleeper
{
void Sleep(
     [in] handle_t h,
     [in] long time);
}
```

`Sleeper` Interface Definition

The IDL definition shown in the previous section completely specifies the interface of the **`sleeper`** object, defining the remote procedure signature of the only method call, **`Sleep`**.

OODCE does not require syntax changes to IDL, but it does enforce the following conventions:

- Each defined remote procedure call must use explicit binding management. In explicit binding, the binding information is passed as the first argument to remote procedures. DCE IDL also supports the implicit and automatic binding modes; however, they must not be used in IDL files passed to the **`idl++`** compiler. The functionality provided by implicit and automatic binding is supported at a higher level in OODCE.
- An interface version number must be specified so that **`idl++`** can generate class names that allow multiple versions of an interface to be used within an application without name clashes.
- Custom binding is an IDL feature not readily supported by OODCE. This can be supported at a higher level and should not be specified in the IDL file.

If the preceding IDL specification resides in a file named **`sleeper.idl`**, the **`idl++`** compiler generates the following files. Some of these relate to client functionality and must be linked with the client. The others relate to server functionality and must be linked with the server.

- **`sleeperS.H`**
- **`sleeperE.C`**
- **`sleeperC.H`**
- **`sleeperC.C`**
- **`sleeper_cstub.c`**
- **`sleeper_sstub.c`**

The following sections describe the content and purpose of each of these files.

Manager Classes

In OODCE, the server-side functionality of an interface is provided by implementing a **`C++`** class generated from an IDL specification. These class definitions are placed in a header file that uses the following naming convention.

`<IDL File Name>S.H`

In the case of the **`sleeper`** interface, **`sleeperS.H`** contains the class that must be implemented by the developer to provide the server-side functionality. This is analogous to the manager routines that are written for a DCE server.

The **sleeperS.H** file contains two class definitions derived from the **sleeper.idl** file. The first class is **sleeper_1_0_ABS** and is an abstract class. The mapping provided by **idl++** is such that the remote procedure call **Sleep,** defined in the IDL file, is a pure abstract member function in this class. The following code is the abstract manager class for the **sleeper** interface. Some details have been removed to improve readability.

Abstract Manager Class

The following abstract class inherits from **DCEInterfaceMgr**, which is a base class that encapsulates the functionality common to all DCE interfaces.

```
class sleeper_1_0_ABS : public virtual DCEObj,
     public DCEInterfaceMgr {
public:
     // Declare Class Constructors
     sleeper_1_0_ABS(uuid_t* obj, uuid_t* type);

     sleeper_1_0_ABS(uuid_t* type);

     // Declare Class pure virtual member functions
     // These correspond to the remote procedures
     // declared in sleeper.idl
     // These need to be implemented by the developer

     virtual   void Sleep(idl_long_int time) = 0;

};
```

This class also virtually inherits from **DCEObj,** which encapsulates functionality common to all DCE objects (for example, object UUID). Virtual inheritance is further described in “How to Develop Manager Objects with Multiple Interfaces” in Chapter 8.

This abstract class is the root class for alternative implementations of the **sleeper** interface. Concrete classes derived from this root class can provide for multiple implementations of an IDL interface that can co-exist in the same program. This is analogous to the manager type concept in DCE.

Concrete Manager Class

The **sleeperS.H** file also contains a second, concrete class for the **sleeper** interface. This concrete class is derived from the **sleeper_1_0_ABS** abstract class and provides for a default implementation class for the **sleeper** interface.

Developers of **sleeper** objects can define and implement their own **sleeper** class derived from **sleeper_1_0_ABS** or can provide an implementation for the concrete class provided by **idl++**.

The following code shows the details for the default concrete class. Some details have been removed to improve readability. No other implementation is necessary since the implementation for the constructors is provided by default.

```
class sleeper_1_0_Mgr : public sleeper_1_0_ABS {
public:
     // Declare Class Constructors
     sleeper_1_0_Mgr(uuid_t* obj);
     sleeper_1_0_Mgr();
     // Declare Class member functions
     // These correspond to the remote procedures
     // declared in sleeper.idl
     // These need to be implemented by the developer

     virtual   void Sleep(idl_long_int time);
};
     // Sleep member function implementation
     sleeper_1_0_Mgr :: Sleep(long int time)
     {
        sleep((unsigned int) time);
     }
```

Entry Point Vector and Code

The **`idl++`** compiler generates a server side communications stub file. This stub file assumes that an entry point vector structure exists for an interface. An entry point vector contains entry points for each remote procedure call defined in an interface. In C, these entry points are pointers to procedures that implement a particular remote procedure. In OODCE, the remote procedures are implemented as member functions of **`C++`** objects rather than entry point procedures. However, an entry point must exist for each remote procedure to allow the communications stubs to operate correctly. The **`idl++`** compiler automatically generates the entry point vector for an interface and places it in a file with the following naming convention:

`<IDL File Name>E.C`

For the **`sleeper`** example, the entry point vector support is placed in **`sleeperE.C`** and provides the interface from the C-based communications stubs generated by the **`idl++`** compiler to the **`C++`** object that implements the interface.

A manager object is an object that implements the remote operations defined in the IDL file. In OODCE, each instance of a **`C++`** manager object that implements an interface is identified by a unique identifier called the object UUID. Each client request must be routed to a specific instance of a **`C++`** manager object based on the interface and DCE object UUID carried in the Remote Procedure Call (RPC) packet. The Entry Point Vector (EPV) code generated by the **`idl++`** compiler implements this routing functionality and ensures that the correct **`C++`** manager object on the server is called for each client request. The EPV code uses an internal table called the object map to route client requests to the correct **`C++`** manager object (Figure 2-1).

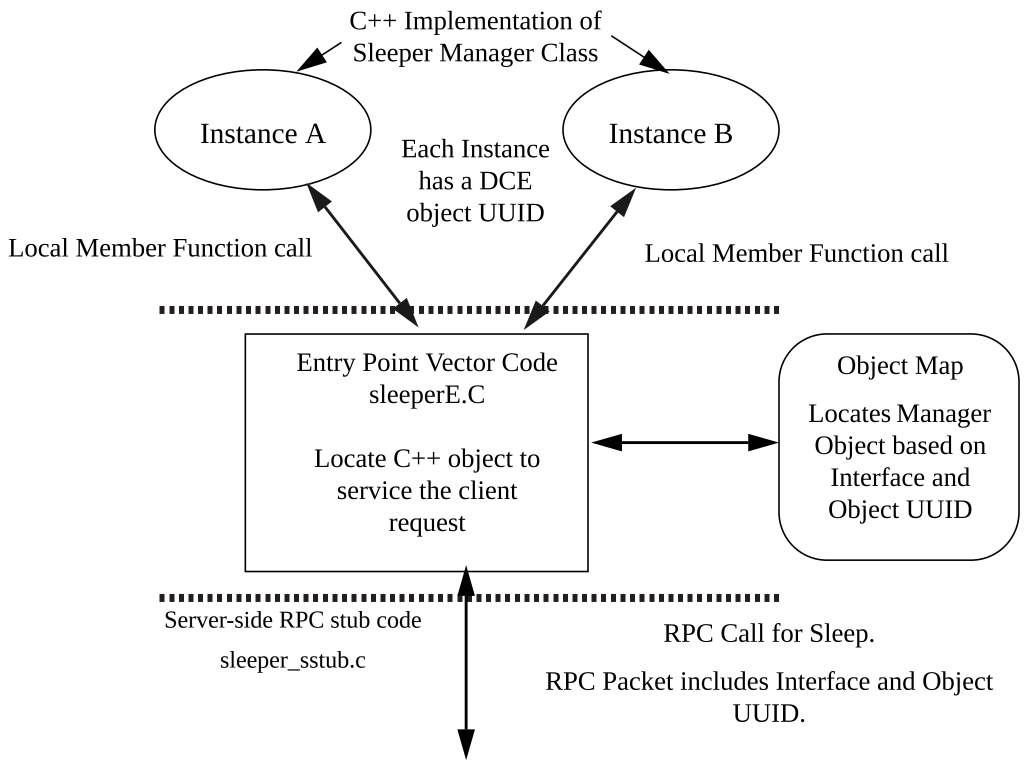
C++ Implementation of Sleeper Manager Class
Instance A
Instance B
Each Instance has a DCE object UUID
Local Member Function call
Local Member Function call
Entry Point Vector Code sleeperE.C
Locate C++ object to service the client request
Object Map
Locates Manager Object based on Interface and Object UUID
Server-side RPC stub code
sleeper_sstub.c
RPC Call for Sleep.
RPC Packet includes Interface and Object UUID.

Figure 2-1. Server-Side Communications — EPV Dispatch

For the `sleeper` example, the code in `sleeperE.C` must locate (using the **Object Map**) an instance of a `sleeper` manager object (based on Interface and Object UUID) derived from `sleeper_1_0_ABS`. Once the manager object has been found, the `Sleep` member function is called on that object. A server can have multiple instances of a `sleeper` object, and each instance would have a different DCE UUID. Each client RPC request specifies a DCE object UUID and the EPV code locates the correct instance of a manager object based on this information.

The EPV code is also responsible for calling optional security checks that can be performed before the manager object's member functions are invoked. This feature is further described in Chapter 6, Basic Security.

Client Class

In OODCE, server objects are accessed via a client object. This object is used to locate server-based manager objects and to make remote procedure calls to member functions of those objects. The client object class definition is derived from an interface specification and can be found in a file that uses the following naming convention:

`<IDL File Name>C.H`

The **`idl++`** compiler also generates a default implementation of this class. This implementation can be used to locate and access corresponding remote manager objects that implement the interface from which the client class was defined. You can provide other implementations of this client object class or overload member functions that control access to the server object. This kind of modification is discussed in Chapter 8. The default implementation for the client class can be found in a file with the following naming convention:

`<IDL File Name>C.C`

For the **`sleeper`** example, the client class files generated are called **`sleeperC.H`** and **`sleeperC.C`**. The **`sleeper`** client class inherits from a base class called **`DCEInterface`**, which encapsulates basic functionality for the client object.

Locating Manager Objects

One of the main functions of the **`DCEInterface`** class is locating objects. When the client object is constructed, location information can be supplied to the constructor through arguments. By default, location information is not required. In this case, any manager object found that implements the required interface (it is not guaranteed to be found) is used. The **`DCEInterface`** base class locates objects based on the following information:

- Interface (default). Any object that implements the requested interface is used.
- Cell Directory Service (CDS) name. This name can refer to a server, a group or a profile entry in the CDS.
- Host address and protocol sequence.
- Object reference.

The **`DCEInterface`** class also provides constructors that allow binding information to be passed to the client object. This allows the location of a manager object to be performed outside of the class and the obtained bindings are passed to the constructor.

The member function that locates an object is a virtual function and can be overloaded to provide for custom location policies. This is further described in “How to Implement a Custom Naming Policy” in Chapter 8.

The other main function of the **`DCEInterface`** class is to specify client side security preferences.

Client Class Example

The following code shows the client class specification for the **`sleeper`** example. Some details have been removed to improve readability.

```
class sleeper: public DCEInterface {
public:
     // Define Class Constructors
     // Locate by DCEInterface (default)
     sleeper(DCEUuid& to = NullUuid);

     // Binding passed to constructor no lookup performed
     sleeper(rpc_binding_handle_t bh,
          DCEUuid& to = NullUuid);
     // Binding passed to constructor no lookup performed
     sleeper(rpc_binding_vector_t* bvec);

     // Lookup by name (uses CDS)
     sleeper(unsigned char* name,
     unsigned32 syntax = rpc_c_ns_syntax_default,
     DCEUuid& to = NullUuid);

     // Lookup by host address and protocol sequence
     sleeper(unsigned char* netaddr,
     unsigned char* protseq, DCEUuid& to = NullUuid);

     // Lookup by Object Reference
     sleeper(DCEObjRefT* ref);

     // Member functions for client object

     void Sleep(idl_long_int time);

};
```

Notice that the **`Sleep`** function defined in the IDL file becomes a member function of the client class. Client programs call this local member function to invoke the **`Sleep`** operation on a server-side **`sleeper`** manager object.

DCE Stub Files

The **`idl++`** compiler generates communication stubs for the client and the server. These files perform the same function for OODCE as they do for DCE-based applications. The generated stub files have the following naming convention:

`<IDL File Name>.h` — Common header file

`<IDL File Name>_cstub.c` — Client-side communications stub

`<IDL File Name>_sstub.c` — Server-side communications stub.

The stub files handle the communication between client and server and perform marshalling and unmarshalling of data types passed across an RPC interface. For the **`sleeper`** example, the following files are generated: **`sleeper.h`**, **`sleeper_cstub.c`** and **`sleeper_sstub.c`**.

Writing the Server Program

The **`idl++`** compiler generates the basic framework class structure for client and server development. You must fill in the missing functionality to develop the server and client applications. The following sections describe the basic functionality that you must provide to complete development of an OODCE system.

There are two important items that must be developed to complete the server-side of an OODCE system.

- An implementation for the objects managed by the server. This is done by implementing the manager classes generated by the **`idl++`** compiler for the interfaces supported by the objects.
- A main function that initializes the server and registers information in the DCE for the server and the objects it supports.

Implementing the Manager Object

For the **`sleeper`** example, the first task is to implement the server concrete class generated by the **`idl++`** compiler from the **`sleeper`** IDL file. You can use the generated default concrete class (**`sleeper_1_0_Mgr`**) and complete the implementation or derive a new class from the generated abstract class (**`sleeper_1_0_ABS`**).

For the **`sleeper`** example, implementing the default concrete class involves providing an implementation for the **`Sleep`** member function that corresponds to the remote procedure call defined in the IDL specification for **`sleeper`**. The semantics of **`Sleep`** are to block for a period of time. The time is determined by the value of a single argument passed into **`Sleep`**.

Since this is a server-based manager object, multiple clients can call the object concurrently. This means that the implementation of **`Sleep`** must be threadsafe (re-entrant and non-blocking for the process).

The following implementation uses the UNIX **`sleep`** system call to provide the implementation of the **`Sleep`** member function of the **`sleeper`** object.

```
void sleeper_1_0_Mgr::Sleep(long int time)
{
     // Call Unix sleep function
     sleep(time);
}
```

This code is all that is required to provide an implementation of a basic **`sleeper`** object. More complex forms of a **`sleeper`** object are described in Chapter 8, Advanced Application Development.

Implementing Server Main Function

The second step required to complete the implementation of a **`sleeper`** server program is to implement a main function that initializes the server and optionally registers information in the DCE. For the basic example of **`sleeper`**, a server process with a single **`sleeper`** manager object is used. The server does not use the CDS or security functionality.

OODCE provides a class called **`DCEServer`** that manages objects and interacts with DCE. The **`DCEServer`** class is responsible for object registration, protocol selection, the CDS and security preferences, cleanup, and listening for client requests. The **`DCEServer`** class is concrete and the OODCE library provides the implementation. Many of the member functions supported by the **`DCEServer`** class are virtual and can be overridden in derived classes. Reasons for deriving from the **`DCEServer`** class are discussed in Chapter 8, Advanced Application Development.

Only one instance of a **`DCEServer`** object can exist per process. This is because DCE only allows one **`rpc_server_listen`** call per process. For convenience, the OODCE library contains a Global Server Object (GSO) called **`theServer`** that can be used by server programs. The implementation of the **`DCEServer`** class is threadsafe, so the global server object can be safely accessed from any thread within the server process.

The basic steps for developing a server main function are as follows:

1. Construct manager objects that are accessed via the server.
2. Create and activate a signal handling thread to perform server cleanup.
3. Register objects created in Step 1 with the GSO.
4. Select communication protocols (optional).
5. Set naming preferences for the GSO (optional).
6. Set security preferences for the GSO (optional).
7. Instruct the GSO to listen for client requests.

The following sections describe these steps.

Constructing Manager Objects

The functionality provided by a server in OODCE is provided by **`C++`** objects that implement the server-based classes generated by the **`idl++`** compiler. Each of these objects has a different object UUID and can be activated either when the server process starts or can be activated later.

This step initializes the objects that are active when the server process starts. Chapter 8, Advanced Application Development, which includes a description of factories and activation, describes the case where objects are activated after the server is listening.

Using the **`sleeper`** server example, the server-side class is named **`sleeper_1_0_Mgr`**. The implementation of this class is described in “Implementing the Manager Object” earlier in this chapter. The following code sample constructs an instance of this class. In this case, the DCE object UUID is created automatically by the constructor; however an object UUID could be passed to the constructor.

```
sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
```

The object constructed in this code implements the **`Sleep`** member function of the **`sleeper`** interface. This object can be accessed by a client to perform the **`Sleep`** operation using DCE RPC. To make this object available to clients, it must be registered with the GSO. This is discussed in “Complete Server Main Function Example” later in this chapter.

The following figure shows the main **`C++`** objects that exist in the server program immediately after **`obj`** is created.

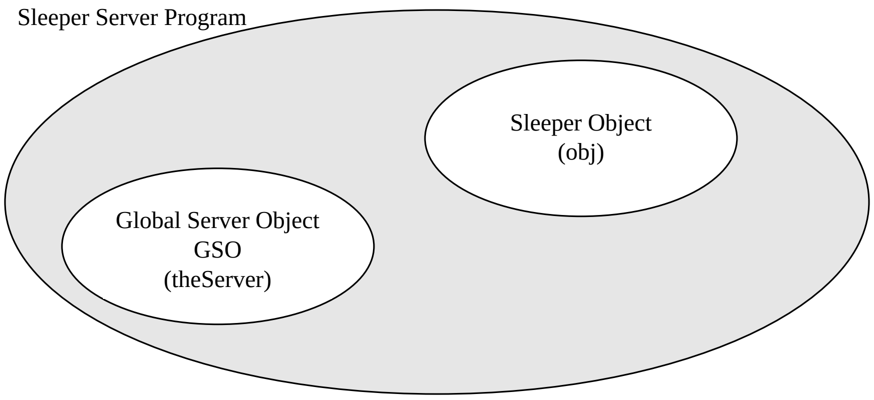

Figure 2-2. Server Program Initial State

Creating a Signal Handling Thread

One of the biggest problems facing a server developer is performing cleanup tasks after the receiving asynchronous signals (for example, SIGTERM). If appropriate cleanup is not executed when such a signal is sent to a process, stale information can be left in DCE that can cause performance problems and sometimes failure in other system elements (such as stale bindings being passed to clients). The **`DCEServer`** class implementation provided by OODCE provides a default signal handler that performs appropriate cleanup.

The following code fragment shows how to activate the default signal handler for the GSO in the **`sleeper`** program.

```
DCEPthread*cleaner =
          new DCEPthread(theServer->ServerCleanup, NULL);
```

This code creates a thread that executes the **`ServerCleanup`** member function on the global server object. **`ServerCleanup`** sets up a signal handling function that waits for signals and executes a server shutdown if a signal is sent to the process. If an application requires more control over signal handling, **`ServerCleanup`** can be replaced with a different implementation.

Registering Manager Objects With the Server Object

The GSO needs to know about the objects it is managing; therefore, it requires that each object be registered. The act of registration places information about the object in the private state of the GSO. This information is used to register the server side objects with the local DCE runtime and external services such as the CDS.

The following code fragment shows how this registration is done.

```
theServer->RegisterObject(*obj);
```

This code registers the object (obj) created in “Implementing the Manager Object,” earlier in this chapter, with the GSO. Every manager object must be registered with the GSO so that it can be accessed by clients (Figure 2-3).

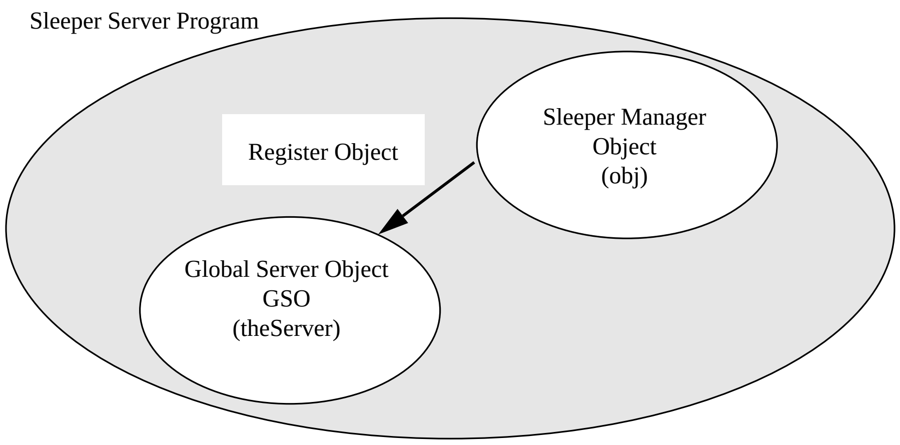
Sleeper Server Program
Register Object
Sleeper Manager Object (obj)
Global Server Object GSO (theServer)

Figure 2-3. Registering the Manager Object With the GSO

Listening for Clients

A manager object within a server can only be accessed by a remote client if the server program is listening for requests. Local client objects can access manager objects even if the server is not listening. Invoking the **`Listen`** member function instructs the GSO to register manager object information (registered with GSO in the previous section) with DCE, and then call the DCE listen loop. **`Listen`** only returns if a problem exists during registration or if the server stops listening for client requests. The latter is normally the result of calling the **`Shutdown`** member function on the GSO. The following code shows how to start the listen loop.

```
theServer->Listen();
```

The call to **`Listen`** registers object and interface information associated with the **`sleeper`** manager object with the local RPC runtime and the Endpoint Mapper before entering a listen loop (Figure 2-4).

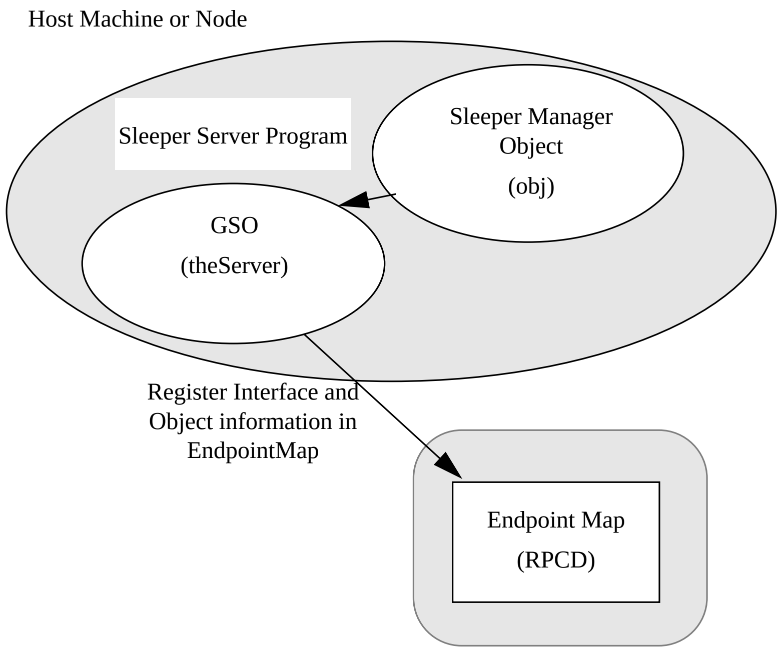

Figure 2-4. Effect of Calling `Listen`

Complete Server Main Function Example

The following code shows the implementation of the complete main function for the **`sleeper`** example.

```
main()
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(*obj);

     // Listen for client requests
     theServer->Listen();
}
```

This code is relatively simple compared to the equivalent functionality written in C. Much is taking place that is transparent:

- When the **`sleeper`** object is registered with the GSO, an entry for the **`sleeper`** manager object is placed in the Object Map table. The Object Map is used by the EPV code to route client requests to the **`C++`** manager object associated with a particular object UUID and interface.
- By default, the GSO listens for client requests on all protocols supported by DCE on the platform (i.e., TCP/IP and UDP/IP on UNIX). The GSO registers information about manager objects (**`obj`**) in the Endpoint Map for the host on which the server program is running. The following is an example of the Endpoint Map information for a **`sleeper`** server.

```
<object> 9ca883f0-ef51-11cc-bea4-080009627155
<interface id> 7395e26e-5ba4-11cc-988b-080009253b97,1.0
<string binding> ncacn_ip_tcp:15.1.161.80[2704]
<annotation>

<object> 9ca883f0-ef51-11cc-bea4-080009627155
<interface id> 7395e26e-5ba4-11cc-988b-080009253b97,1.0
<string binding> ncadg_ip_udp:15.1.161.80[4820]
<annotation>
```

- Should the call to **`listen`** return (because the server was told to stop listening by a management application or some internal failure was detected), the destructor for the GSO executes when the main function exits. This destructor cleans up and removes server information from the Endpoint Map and local RPC runtime. The GSO performs a similar clean up if the server process receives a signal such as SIGTERM or if a user types ^C. Note that a SIGKILL does not cause the cleanup handlers to be called since the process exits immediately.

The **`sleeper`** main function must be linked with the implementation of the **`Sleep`** member function and the entry point code and server stub code generated from the **`sleeper`** IDL specification. The result is a program, that when executed, produces a **`sleeper`** server process with a single manager object that implements the **`sleeper`** interface specified in the **`sleeper.idl`** file.

Selecting Communication Protocols

By default the GSO listens for client requests over all of the DCE-supported transport protocols available on a node. OODCE allows communication protocol preferences for the GSO to be set if fine control is required for application purposes. This section describe the features available for controlling the selection of communications protocols.

Selecting a Specific Protocol

Instead of using all available protocols, you can force the use of a single protocol for interactions between the client and server. To do this in OODCE, use the overloaded **`UseProtocol`** member functions of the GSO. There are three such member functions for specifying that the GSO should use a particular protocol. They differ in the way that the endpoint information is determined. Basically a communications endpoint can be:

- Chosen by the DCE runtime
- Supplied as an argument to one of the **`UseProtocol`** functions
- Derived from an IDL specification of an interface.

The **`UseProtocol`** member functions may be called multiple times to select the use of more than one protocol, but these calls must be made before the server enters the **`listen`** loop. The syntax for the three versions of **`UseProtocol`** is shown in the following paragraphs.

Choosing Communications DCEEndpoint by DCE Runtime

In the following code, the communications endpoint for the specified protocol is chosen by the DCE runtime.

```
DCEServer::UseProtocol(char*protocol_name,
                       unsigned32max_requests);
```

This call tells the GSO to use a specific protocol for communications between the client and the server. The second argument specifies the maximum concurrent calls supported in this protocol.

The **`sleeper`** example is modified to use this call, as follows:

```
main()
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(obj);

     // Make Server object use TCP/IP
     theServer->UseProtocol(“ncacn_ip_tcp”, 5);

     // Listen for client requests
     theServer->Listen();
}
```

How To Set the Protocol Endpoint

The following call specifies a protocol for the server program to use as well as the communications endpoint. The DCE runtime uses the endpoint information provided in the call to **`listen`** for client requests.

```
DCEServer::UseProtocol(char* protocol_name,
     char*endpoint,
     unsigned32 max_requests);
```

The modification to the basic **`sleeper`** server main function is similar to the main function listing. In the following code, the endpoint string is specified on the **`UseProtocol`** member function.

```
main()
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(obj);

     // Make Server object use TCP/IP
     // Set the DCEEndpoint for TCP.IP
     theServer->UseProtocol(“ncacn_ip_tcp”
          “1025”, 5);

     // Listen for client requests
     theServer->Listen();
}
```

Using Well Known Endpoints

It can be useful to have server programs listen for client requests on what is known as a well known endpoint. In this case, the endpoint information is published and does not change for every instance of a server program. DCE allows endpoint information to be specified and published in the IDL specification. Clients and servers that use the IDL specification for communication then use the endpoint information specified in the IDL file.

The following code shows the **`sleeper`** IDL specification modified to support well known endpoints.

```
[uuid(),
version(2.0),
endpoint (“ncacn_ip_tcp:[1025], “ncadg_ip_udp:[6677]”)
] interface sleeper
{
     void Sleep(
          [in] handle_t h,
          [in] long time);
}
```

The GSO can be told to use well known ports declared in the IDL specification by using the following member function calls.

```
DCEServer::UseAllProtocols(DCEInterfaceMgr& interface,
     unsigned32 max_requests)

DCEServer::UseProtocol(char* protocol_name,
     DCEInterfaceMgr& interface,
     unisigned32 max_requests);
```

The first member function instructs the GSO to use all DCE-supported protocols configured for the local platform. The endpoint information for those protocols is obtained from the IDL specification relating to the first parameter, **`interface`**. This parameter is of the **`DCEInterfaceMgr`** class, which is the base class of all server manager classes created by the **`idl++`** compiler.

The second function allows the specification of a specific protocol for which the endpoint information is obtained from the IDL file relating to the second parameter, **`interface`**.

The following **`sleeper`** server main example uses the first function.

```
main()
{
     // Create single instance of sleeper object
     sleeper_2_0_Mgr* obj = new sleeper_2_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(obj);

     // Use all Protocols specified in sleeper.idl
     theServer->UseAllProtocols(*obj, 5);

     // Listen for client requests
     theServer->Listen();
}
```

This example instructs the GSO to use all available protocols and to obtain their endpoints from the IDL file related to the interface supported by **`obj`**, which is the **`sleeper`** IDL file.

Writing the Client Program

To create an application that accesses the **`sleeper`** server, an instance of the client object generated by the **`idl++`** compiler can be created. Since the **`sleeper`** server does not use the CDS, some location information must be passed to the constructor of the client class. Once the client object is created, its member functions can be called to access the server object.

In the following code sample, location information is provided in the form of a host address and protocol sequence, but the object UUID is not specified. The location code in the client object automatically binds to a compatible object (i.e., an object that supports the **`sleeper`** interface) at a server running at the provided host address.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0(“porter”, “ip”);

     // Call remote sleep operation
     sleepObj.Sleep(10);

}
```

In this example, the client object goes to the host ‘‘porter’’ and tries to find a server program that implements **`sleeper`** objects. Once a server program has been located, (in this case using the DCEEndpoint Map on “porter”) an object associated with the server is chosen to handle the **`Sleep`** request. If the example made a second call to **`Sleep`**, the second call would be routed to the same **`C++`** object at the server.

The preceding code fragment must be linked with the client class implementation generated by the **`idl++`** compiler and the client-side stub files.

The following figure shows how a client object uses the DCEEndpoint Map to locate the server-based manager object.

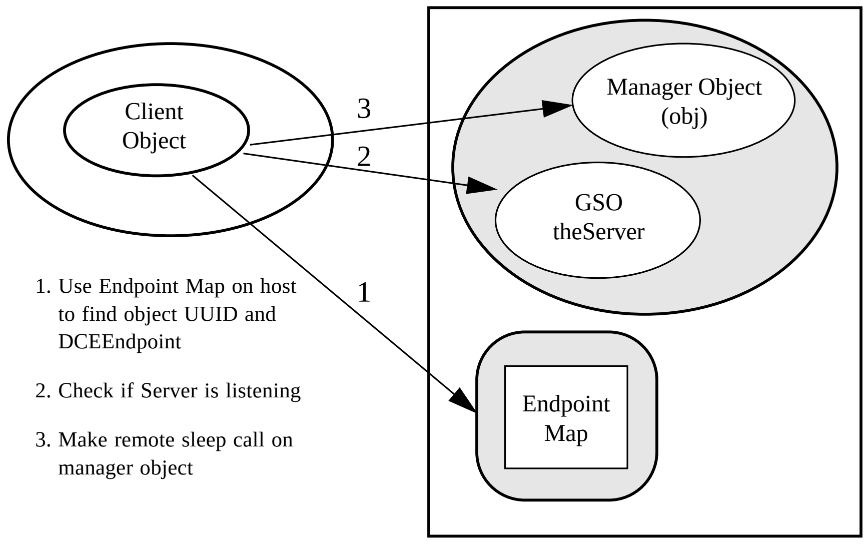
Client Object
3
2
1
Manager Object (obj)
GSO theServer
Endpoint Map
1. Use Endpoint Map on host to find object UUID and DCEEndpoint
2. Check if Server is listening
3. Make remote sleep call on manager object

Figure 2-5. Client Location of Server-Based Manager Objects

3

Using the Cell Directory Service

Setting naming preferences for the Global Server Object (GSO) is one of the steps for implementing a server main function. This step is only required if host independence is required for the location of server based manager objects. (Note that in code discussed in Chapter 2, the host name was passed to the constructor of the client object.) The **`sleeper`** client code does not register the server in the Cell Directory Service (CDS), and as such cannot take advantage of certain location mechanisms supported by OODCE.

The implementation of the OODCE GSO provides a policy for using the CDS to locate objects. Because it is likely that specialized naming schemes may be required for certain applications, the default policy can be changed by deriving from the **`DCEServer`** class and overloading the member functions that access the CDS. This is described in “How to Implement a Custom Naming Policy” in Chapter 8.

Placing Bindings in the CDS

Binding information can be placed in the CDS by giving the GSO a name. Once named, the GSO creates a server entry in the CDS and registers its binding information under that name. The **`SetName`** member function is used to give the GSO a name.

How to Set the CDS Name Used By a Server Program

Using the **`sleeper`** example, the server bindings can be registered in the CDS by adding the **`SetName`** call to the implementation of main. The following code fragment shows this modification.

```
main()
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(*obj);

     // Set the name of the server object to use CDS
     theServer->SetName(
          “/.:/subsys/HP/sample-apps/Sleeper”);
     theServer->SetProfile(“/.:/cell-profile”);
     // Listen for client requests
     theServer->Listen();
}
```

When this code fragment is linked into a program and executed, the bindings for the GSO get registered in the CDS under the name of **`/.:/subsys/HP/sample-apps/sleeper`**.

This binding registration allows clients to perform a server lookup by interface on the CDS. With this mode of lookup, a client can bind to any server in a cell that exports the **`sleeper`** interface. The name **`/.:/cell-profile`** must exist in every cell. It represents a top level CDS profile for the registration of services within a cell. See the *DCE Application Development Guide* for more information on CDS profiles.

When a server has a name, the GSO cleanup code knows that it must remove information from the CDS when the server process terminates or the server is shutdown. Leaving stale information in the CDS can present an administrative and performance problem in large cells. Having the server object automatically remove information from the CDS relieves you of this task.

Placing binding information in the CDS allows more choices for the client to locate manager objects at the server. In the **sleeper** example in Chapter 2, the host address is passed to the client constructor. In that example, knowledge of the host address where a **sleeper** server is running is required to locate a manager object. Using the CDS can provide more location transparency in that the client need not know the host on which the server is running.

By setting the server name, the client program can now find **sleeper** manager objects within a cell based on that server name or by interface using the cell profile. Using a server name allows server-managed objects to be associated with a user friendly, location-transparent name. If the server program moves from node to node (e.g., due to maintenance of the node on which it was first executing), its manager objects can still be located via the server name.

How a Client Uses the CDS to Find Manager Objects

The following code fragments show changes that can be made to the client program of the **sleeper** example that can take advantage of the server using the CDS. The following code fragment demonstrates locating a server based on name lookup.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0(
          “/.:/subsys/HP/sample-apps/Sleeper”);

     // Call remote sleep operation
     sleepObj.Sleep(10);
}
```

In this case, the location code of the client object is directed to a specific server entry in the CDS. The location code returns the server bindings registered under the name **`/.:/subsys/HP/sample-apps/Sleeper`**. The client then binds to a **`sleeper`** object on that server. If no bindings have registered or the entry does not exist, the constructor fails and the client object is not created. This results in an exception being thrown by the constructor.

The second fragment demonstrates locating a server based on lookup by interface. In this case, no information is passed to the **`sleeper`** client constructor.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj;

     // Call remote sleep operation
     sleepObj.Sleep(10);
}
```

In this code fragment, the client code attempts to find a **`sleeper`** object within the cell by doing a server lookup based on only interface information in **`/.:/cell-profile`**. The IDL specification of **`sleeper`** includes a UUID that identifies that interface. The client object location code initiates a search into the directory using the interface UUID for **`sleeper`** as a key. A search such as this uses CDS profiles to navigate through the directory based on a UUID. The root of the search can be defined by setting the RPC_DEFAULT_ENTRY environment variable. If RPC_DEFAULT_ENTRY is not set or is set to NULL, the search begins at **`/.:/cell-profile`**. The search continues until a server entry is found that exports the **`sleeper`** interface UUID.

Binding information is extracted from the server entry information and is used to contact the host on which the server program is running. Once the host is known, OODCE performs the same steps as the non-CDS example to locate a manager object. The client can bind to a manager object on any server in the cell that has exported the **`sleeper`** interface to the CDS. The client is unaware of the name or location of the server used.

How to Register Server Information into RPC Groups

The **`sleeper`** code on page 2 shows how to register server information into the CDS using OODCE. The CDS also supports groups, whereby server entries can be grouped together and related under a single name. It can be useful to group servers based on some property and use this as a basis for location. For example, a group could be based on type or function (e.g., printers).

In another example of user groups, all **`sleeper`** servers in a cell are to be registered in a single CDS group. This provides a way for administrators to discover all of the **`sleeper`** servers running within a cell and provides more structure and opportunity for locating the **`sleeper`** servers. A server program can be added to a CDS group by setting its group name using the **`SetGroup`** member function on the GSO. In the following code, the **`sleeper`** server main function is modified to show this.

main(int argc, char** argv)

```
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr*obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(*obj);

     // Set the name of the server object to use CDS
     theServer->SetName(argv[1]);

     // Add the server name to a group of sleeper servers
     theServer->SetGroup("/.:/Sleeper_Servers");

     // Listen for client requests
     theServer->Listen();
}
```

Executing this code fragment causes the server name to be added as a member of the group **`/.:/Sleeper_Servers`**. In this example, the server name is passed as the first argument to the program in the **`argv`** argument. This is a case where all **`sleeper`** server programs running in a cell are registered as members of the **`/.:/Sleeper_Servers`** group. This group expands and shrinks as **`sleeper`** servers are added and removed from the cell.

This means that the clients are able to use a CDS group to locate an instance of a **`sleeper`** object as shown in the following code.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0
     ("/.:/Sleeper_Servers");

     // Call remote sleep operation
     sleepObj.Sleep(10);
}
```

The object location code of the client class uses the group name passed in the constructor to select a server for the **`sleeper`** object that can be used by the client. The CDS interface to DCE automatically traverses members of the list to retrieve server entry binding information. Once a binding to a **`sleeper`** server is returned, the host information is used to contact the **`sleeper`** server program at the host to select a manager object. The search of the CDS group continues until a manager object is found (stale entries in the CDS are ignored and OODCE performs checks to see if the server program is active).

How to Register Server Information in RPC Profiles

Profiles are another CDS attribute defined to be used by RPC based services. Profiles provide a way for navigating the directory tree using a UUID as a key. Profiles can be chained together to allow directed searches across large namespaces.

There can be administrative benefits to developing a profile structure in the CDS that represent the organizational boundaries that may exist for a cell or application domain. For example, a cell represents a laboratory environment.

Each laboratory has a set of departments and each department has a set of projects. Profiles can be set up in the directory for the laboratory, departments, and projects. Resources in the form of DCE servers can be assigned to each organizational entity and registered in the appropriate profile. For example, each project may have its own compute and storage servers, but printing resources may be assigned to a department.

An OODCE server program can be automatically registered in a CDS profile by using the **`SetProfile`** member function of the server object.

NOTE **`SetProfile`** must be used in conjunction with **`SetName`**.

The following is a modification of the **`sleeper`** server main function that shows the use of **`SetProfile`** on the server object.

```
main(int argc, char** argv)
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     theServer->RegisterObject(*obj);

     // Set the name of the server object to use CDS
     theServer->SetName(argv[1]);

     // Add the server name to a group of sleeper servers
     theServer->SetGroup("/.:/Sleeper_Servers");

     // Add the server name to department profile
     theServer->SetProfile("/.:/Department_X");

     // Listen for client requests
     theServer->Listen();
}
```

This code fragment adds the server name as an element of the **/.:/Department_X** profile in the CDS. When the server program exits, the cleanup code automatically removes the information from the CDS.

Clients may pass a profile name to the client object constructor as shown in the following code. The location code of the client object uses the profile name and the **sleeper** interface UUID to locate a server object. Manager object location is performed in a similar manner to that described for groups (see “How to Register Server Information into RPC Groups” earlier in this chapter). OODCE uses the CDS interface to search the namespace to obtain bindings to **sleeper** server programs. It then contacts the **sleeper** server to locate a manager object. The following code sets the starting search point in the directory at the profile **/.:/Department_X**.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0
          (“/.:/Department_X”);

     // Call remote sleep operation
     sleepObj.Sleep(10);
}
```

Profiles can be linked together to allow directed searches through the directory. If there is no **sleeper** server registered in **/.:/Department_X**, the search may move on to another profile in the chain. See the *OSF DCE Application Development Guide* for a more detailed description of CDS profiles.

Using Object UUIDs in the CDS

Object information is not registered in the directory by default. OODCE registers server programs that manage objects into the CDS. Although the CDS supports placing DCE object information in the directory, its design does not support fast changing information. The CDS is designed to be a globally scalable directory rather than a fine grained object repository.

There may be classes of objects that are very public. In general, these objects are stable and slow changing and exist for a reasonable period of time. Such objects should be registered in the CDS to publish their object IDs into the cell. By default, OODCE does not register object information in the CDS. It does, however, provide a way to accomplish this on a per object basis. The **`RegisterObject`** member function of the GSO takes an optional boolean argument (default: false) that if set to true results in the UUIDs of its manager objects being registered in the CDS.

The main function of the **`sleeper`** server can be modified, as in the following code, to force the GSO to register the DCE object UUID for the **`sleeper`** manager object in the CDS.

```
main()
{
     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread*clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     // Force registration of object ID in CDS
     theServer->RegisterObject(*obj, true);

     // Set the name of the server object to use CDS
     theServer->SetName(
     “/.:/subsys/HP/sample-apps/Sleeper”);
```

```
      // Listen for client requests
      theServer->Listen();
}
```

When this code executes, the DCE object UUID associated with the **`sleeper_1_0_Mgr`** object is registered in the CDS along with the communication bindings under the name of **`/.:/subsys/HP/sample-apps/Sleeper`**.

The DCE object UUID for the manager object is automatically created when it is constructed.

A DCE object UUID can also be associated with client objects in OODCE. This UUID specifies the object identifier for the manager object that the client wishes to call. It is not required, however, that these manager objects be registered in the CDS for location to succeed. The location code built into the client objects uses the CDS and the Endpoint Map information to find and bind to the specific server manager object that has the object UUID requested by the client.

When a DCE object UUID is associated with a client object, it is assumed that the manager object was created previously by a server program and is already active or can be activated by the server program. The location code may use the CDS to locate the set of server processes in a cell that could support the required object. Then the Endpoint Map that is local to each server in the set is consulted to locate the actual manager object.

DCE object UUIDs can be associated with a client object using the constructor or the **`SetServerObject`** member function implemented by every client object. The following client code fragments show both forms.

```
extern uuid_t theObject; // Identifier for manager object
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0
          ("/.:/Department_X",&theObject);

     // Call remote sleep operation
     sleepObj.Sleep(10);

}
```

This code passes an object UUID to the constructor of the **`sleeper`** client object. In this case, the location code in the client object searches the CDS directory starting at the profile called **`/.:/Department_X`** for servers that support the **`sleeper`** interface. If no object information is registered in the CDS with the server entries found in the search, the Endpoint Map that is local to a server program located using the CDS is searched to obtain a match on the object information provided to the constructor. The search continues until a manager object that supports that **`sleeper`** interface that has the object UUID requested by the client is found or the search is exhausted.

The following client code fragment operates in the same manner as the previous one as far as object location is concerned.

```
extern uuid_t theObject; // Identifier for manager object
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj = sleeper_1_0
("/.:/Department_X");

     // Associated server object with client
     sleepObj.SetServerObject(&theObject);

     // Call remote sleep operation
     sleepObj.Sleep(10);

}
```

Note that the string form of a UUID can be passed to the constructor and **`SetServerObject`** instead of the UUID structure form used in the previous examples.

4

Error Handling

In distributed computing, parts of an application can fail at any time and clients of a server object need to able to detect and process such failures.

Handling Exceptions Raised by the OODCE Library

The OODCE library uses **C++** exceptions as the model for error handling. This provides a clean and consistent model for dealing with problems that can occur when using DCE. The caller of the OODCE library should be prepared to handle exceptions that are raised within the library. All OODCE calls including constructors should be surrounded by a **C++** try/catch clause.

The exceptions defined in the library accommodate the public status codes defined in DCE. The exceptions are arranged into a class hierarchy so that error handling can be scoped. The following figure shows the hierarchy:

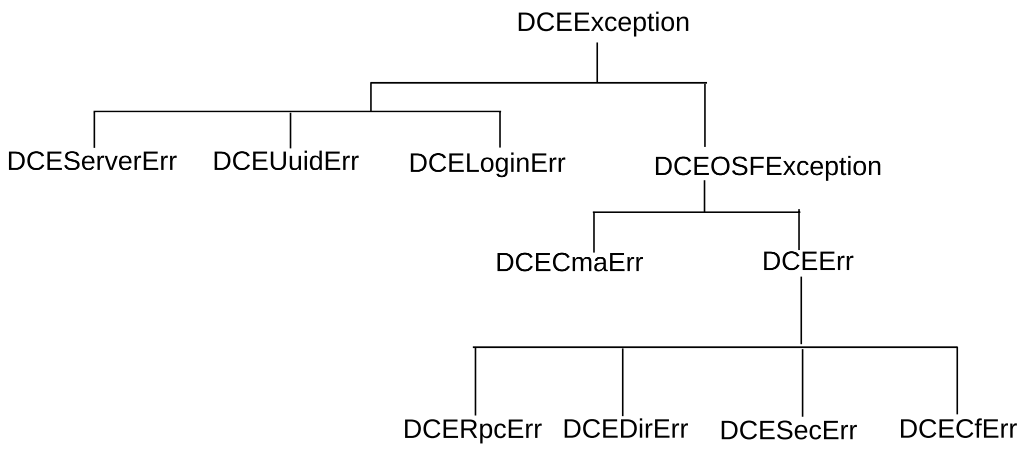

Figure 4-1. Exception Class Hierarchy

The subclasses **DCECmaErr**, **DCERpcErr**, **DCEDirErr**, **DCESecErr**, and **DCECfErr** encapsulate all of the exceptions dealing with Pthreads, Remote Procedure Calls, directory, security, and configuration subsystems of DCE respectively.

The exceptions defined by the library accommodate the public status codes defined in DCE. The exceptions are arranged into a class hierarchy so that error handling can be scoped. A class called **DCEErr** is defined as the base class for all public errors that can occur while using DCE. The subclasses **DCERpcErr**, **DCEDirErr**, **DCESecErr** and **DCECfErr** are derived from **DCEErr**.

The following example shows a typical use of exceptions in server implementation.

```
main()
{    // Set up try block for exceptions
     try {

     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
     // Start Signal handler
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with Server
     // Force registration of object ID in CDS
     theServer->RegisterObject(obj);

     // Set the name of the server object to use CDS
     theServer->SetName(
          “/.:/subsys/HP/sample-apps/Sleeper”);

     // Listen for client requests
     theServer->Listen();

     }
     catch(DCEErr & err) {
          // Process DCE Error
          cerr << “DCE Error: “ << (char*)err << endl;
```

```
      }
      catch(...) {
      // Process unknown exception
      cerr << “Unknown Error server failed” << endl;
      }
}
```

The first catch block traps any DCE error that may occur during the execution of the server main function. The action taken in this case is to convert the exception type into a character string and print a user message. Every public status code defined by DCE has a corresponding exception class in OODCE. Each of these classes can be converted back into a status code (**`error_status_t`**) or to a character string that describes the error and the DCE subsystem that generated it.

Since specific exception classes exist for each DCE status code, specific DCE fault conditions can be trapped and processed. For example, the following code catches an exception that indicates that the user has not done a DCE login and prints out an appropriate error message.

NOTE Using the CDS requires authentication. All processes that access the CDS must do so within a login context. See Chapter 6, Basic Security, for further information on authentication.

```
main()
{     // Set up try block for exceptions
      try {

      // Create single instance of sleeper object
      sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;
      // Start Signal handler
      DCEPthread* clean = new DCEPthread(
           theServer->ServerCleanup, NULL);


      // Register sleeper object with Server
      // Force registration of object ID in CDS
```

```
          theServer->RegisterObject(obj);

          // Set the name of the server object to use CDS
          theServer->SetName(
               “/.:/subsys/HP/sample-apps/Sleeper”);

          // Listen for client requests
          theServer->Listen();

          }
          catch (DCEExcRpcSNoNsPermission()) {
               cerr << “No Login Context for Server\n”;
               cerr << “Please use dce_login command\n”;
          }


          catch(DCEErr & err) {
               // Process DCE Error
               cerr << “DCE Error: “ << (char*)err << endl;
          }
          catch(...) {
          // Process unknown exception
          cerr << “Unknown Error server failed” << endl;
          }
     }
```

Communicating Errors Between Your Application Server and Client

This section describes how fault conditions are transmitted between a server and client program. You can decide whether the application uses exceptions or status codes to communicate these error conditions.

Communicating OODCE Exceptions from Server to Client

By default, the server uses exceptions to communicate errors to the client. There are two ways to do it, as follows:

- Propagate exceptions raised by the OODCE library directly to the client program without translation. This happens automatically if the server does not catch the exception. In this case, the client must recognize and handle OODCE errors directly.
- The server program catches an OODCE library exception and translates it into an application-specific exception that the client program is more likely to know how to handle.

In general, the second approach is recommended, although there might be situations where the first approach is more desirable.

Fatal Library Errors

If a library constructor fails, it is assumed to be a fatal error. A constructor that is aware of its own failure throws an exception that can be caught by the caller, but recovery is unlikely. Similarly, if memory is exhausted and the **`C++`** “new” facility fails, the **`new_exception`** member function is called. **`new_exception`** throws an insufficient memory exception, but the process of throwing the exception is likely to fail. If your application dumps core and **`new_exception`** appears in the stack trace, this indicates insufficient memory.

Communicating Application Server Exceptions to the Client Program

By default the **`idl++`** compiler generates special stub code that can catch and process **`C++`** exceptions on the server and return them across the network to be thrown within the client application. This allows the client program to catch and handle exceptions that occurred during a remote call. This gives information to that client code to determine how to deal with a problem at a server. This level of detail is normally useful for determining if a remote call can be replayed on the same server program (with perhaps some modifications to the parameters) or if a new server program should be found to service the call.

Interoperability with C Based DCE Systems

Because OODCE uses the DCE fault stream for raising server and communication errors back to the client during a remote procedure call, mixed C and **`C++`** clients and servers may use the exception handling mechanism to communicate server side errors. Exceptions that are received by a **`C++`** client from a C server are translated into **`C++`** exceptions by the default client class implementation and raised to client application code (client class implementations provided by the developer should also perform this action). When C clients receive exceptions from **`C++`** servers, the client stub code raises the exception to client application code as in the normal DCE case. On C-based legacy systems (existing DCE clients or servers), new, non-DCE exceptions that are raised by a **`C++`** side object are caught on the C side, but may not be recognized.

NOTE

When using the **`idl++`** compiler to generate stub code for a C-based server, be sure to specify the **`-no_sepv`** option to suppress generation of an OODCE server entrypoint vector. This entrypoint vector is specific to OODCE server objects and will not be recognized by C-based server code.

Using Application Status Codes Instead of Exceptions

If your application uses status codes to communicate errors from server to client, your server must catch exceptions raised from within the library and translate them into an application-specific status code to pass back to the client. Even if you use application status codes, your client must catch exceptions that are raised by the server's EPV code, which is executed before control reaches your application code. For example, if you registered a **`RefMon`** object with the server, it is invoked by the EPV before control reaches your application code. If the security check fails, an exception is raised. There is no opportunity for your application to translate that exception into an application-specific status code. Your client must be able to recognize an authorization failure in its exception handling code.

5

Basic Pthreads

This chapter provides information on the basics of using pthreads objects. “Creating a Signal Handling Thread” in Chapter 2 describes how to create the default signal handling thread for a graceful cleanup.

A thread is an independent flow of execution taking place within a process. Threads are related in that they can share memory, but are executed independently. Because they have access to the same memory, care must be taken so that one thread doesn’t destroy the work of another thread. This would happen if one thread overwrites a result just written by another thread. The DCEPthread classes, based on the X/Open Pthread standard, provide the means for creating independent threads and coordinating their use of resources.

A Pthread is the representation of a thread of execution. A **`DCEPthread`** object corresponds to a single thread. The **`DCEPthread`** class provides limited information about a thread and limited control of that thread. A **`DCEPthread`** object represents the thread before, during, and after its execution. The thread may also continue to execute after the **`DCEPthread`** object has been deleted.

Creating a New Thread

To create a new thread, construct a **`DCEPthread`** object and pass to the constructor the name of the routine to be executed and the argument to be passed to the routine. The following is an example of creating a checkpointing thread for a persistent database:

```
DCEPthread checkpoint_thread = DCEPthread(checkpoint,NULL);
```

`checkpoint_thread` is a **`DCEPthread`** object representing the thread of execution that is performing checkpointing. The **`DCEPthread`** object supports member functions for:

- Changing the priority of the thread
- Changing the scheduling policy
- Setting its initial stacksize
- Joining with the thread.

In this example, `checkpoint` is the name of the routine that is executed when the thread starts to run. **`checkpoint()`** is passed a NULL argument in this example.

When writing the checkpoint routine, declare the prototype as follows:

```
DCEPthreadResult checkpoint(DCEPthreadParam param);
```

The pthread attributes that had to be explicitly defined when using DCE directly are initialized to default values. The man pages describe the choices for the attributes, the default values, and how to override the defaults, if necessary.

Creating a Mutual Exclusion Lock

To create a mutual exclusion lock, declare a **`DCEPthreadMutex`** object:

```
// protects bank account balance from concurrent deposits
// and withdrawals
DCEPthreadMutex balance_lock;
```

The **`Lock`** member function acquires the lock, and **`UnLock`** releases the lock:

```
balance_lock.Lock();
balance_lock.UnLock();
```

The mutex attributes that had to be explicitly defined when using DCE directly are initialized to default values. Man pages describes choices for the attributes, the default values, and how to override the defaults, if necessary.

How to Operate on Your Current Thread

To execute pthread operations on your current thread, you first need a **`DCEPthread`** object that represents your current thread. This is obtained by constructing a **`PthreadSelf`** object.

```
PthreadSelf mythread();
```

`mythread` can be used to:

- Yield or delay the current thread
- Change cancellation policy or priority
- Test cancellation
- Exit.

For example, the following invocation causes the current thread to stop running and allows the threads dispatcher to choose another thread to run.

```
mythread.Yield();
```

6

Basic Security

In chapters 1 through 4, the **`sleeper`** server example has not used security. The interactions between the client and server objects have not been protected in any way. This section describes how to use the basic security features offered by the OODCE library. More advanced security support is described in Chapter 9, Advanced ACL Management.

How to Set Up a Login Context for Client and Server Programs

To successfully run the client and server programs for the **`sleeper`** examples that use the CDS, the user (principal) that executes the programs must have logged into DCE. This is also true for any programs that are to use the DCE security features. The client and server programs shown up to this point in the manual inherit the security credentials of their parent process (normally a UNIX shell). Establishing a login context is usually more important to a server program than a client program. In general, server programs run for a long period of time (relative to the clients that use them) and should have their own security identity (principal name) within the DCE. This section describes how to establish and maintain a DCE login context with OODCE.

A login context contains the information necessary for a principal to access distributed services. Once obtained, a login context has a finite time during which it is valid. After that time, it cannot be used. A login context must be refreshed to be used continuously.

Normally, client programs are run by a user that has acquired a login context through the **`dce_login`** command. In this case, having the client program inherit the user’s credentials is probably desirable. Server programs, on the other hand, should probably be given their own principal name rather than running as a user principal. Server programs usually run for a long period of time.

This means their login context will expire and become invalid and need to be refreshed. Therefore, server developers are faced with writing a significant amount of code to establish and maintain a login context for a server principal.

To help develop this code, the OODCE library provides two abstract classes:

- **`DCELoginContext`** — maintains and refreshes a DCE login context
- **`DCEPassword`** — provides an abstraction for accessing password information.

These classes are abstract because multiple implementations could exist for maintaining a login context and accessing password data. For example, there are several ways to obtain a login context based on the scope of the certification required. (See the *OSF DCE Application Development Guide* for more details.) There are also several ways of accessing and storing a password (e.g., files and smart cards). Although multiple implementations can be supported, it is expected that an implementation may be reused by many different programs and systems. The OODCE library provides a default implementation of the classes that are described in the following section.

Using DCELoginContext and DCEPassword in OODCE

OODCE supplies an implementation for both **`DCELoginContext`** and **`DCEPassword`** via the **`DCEStdLoginContext`** and **`DCEMemPassword`** classes. These should be suitable for many application uses.

The **`DCEStdLoginContext`** class inherits from the **`DCELoginContext`** abstract class. The implementation of **`DCEStdLoginContext`** establishes a login context on behalf of a principal. This login context is both validated and certified with the DCE security server. (See the *OSF DCE Application Development Guide* for more details.) The **`DCEStdLoginContext`** implementation automatically refreshes the login context before it expires allowing servers to run continuously.

The **DCEMemPassword** class inherits from the **DCEPassword** abstract class. The implementation of **DCEMemPassword** provides a mechanism by which a password can be entered, in clear text, from stdin. This clear text password is encrypted and stored in memory. All clear text versions of the password contained in memory buffers and files are overwritten.

A login context can be associated with a client or a server process. The following code fragment shows a modification to the **sleeper** server main function that sets up a login context for the server principal. The principal name used by the server program is specified as an argument to the server program. The **DCEMemPassword** object is passed the principal name in the constructor and then prompts the user that started the program for a password.

```
void main(int argc, char** argv)
{
     // Check Arguments
     if (argc < 2) {
     cerr << “Usage: “ << argv[0];
     cerr << “ principal_name\n”;
     exit(1);
     }

     // Construct the Sleeper Object
     sleeper_1_0_Mgr sleeper;
     // Construct Cleanup Thread
     DCEPthread*exitThd = new DCEPthread(
          DCEServer::ServerCleanup, (void*)(0));

     // Construct DCEPassword object. Pass in Principal
     // name. This object will prompt for password
     DCEMemPassword thisPass(argv[1]);

     // Construct Login DCEContext object
     DCEStdLoginContext thisContext(&thisPass);

     // DCEServer has established a login context at
     // this point
     // Register Sleeper object with the server object
     theServer->RegisterObject(sleeper, true);

     // Register server in CDS.
     theServer->SetName(
          “/.:/subsys/HP/sample-apps/Sleeper”);

     // Accept all other defaults and activate the server
     theServer->Listen();

     // Destructors are called at this point
     // object take care of appropriate cleanup
     // with DCE runtime
}
```

To execute this code, the following user interactions would occur. Assume that a principal name **sleepy** has been defined for the server program. The user would type the following command:

```
a.out sleepy
> Please Enter Password for Principal sleepy
>
```

Once the password has been entered (terminal echo is turned off for password entry) the login context is established and the server program initializes and enters the DCE listen loop.

How to Set Security Preferences

The initial task in using DCE security involves setting security preferences. These preferences are set by the client and server programs and must be compatible for communication to occur.

The server program is responsible for setting the following preferences:

- Security Principal name associated with the server.
- Authentication service type used by the server.
- Mechanism for retrieving security keys. By default, a file name is expected which relates to the name of the key store.

How to Set Server Security Preferences

The server program sets these preferences by calling the **`SetAuthInfo`** function on the GSO. This method determines the service to call the rpc_server_register_auth_info Application Programing Interface (API). The following code fragment shows the use of this function with the sample **`sleeper`** service.

```
void main()
{

      // Create sleeper manager object
      sleeper_1_0_Mgr sleeperObj;

      // Create Cleanup thread
      DCEPthread* exitThd = new DCEPthread
            (DCEServer::ServerCleanup,
                         (void*)(0));

      // Register manager object
      theServer->RegisterObject(sleeperObj);

// Set the server authentication preferences
// first argument is the server principal name
// the second argument is the authentication model
// required. The third argument is given to the default
// key retriever object as the name of the file in which
// the secret keys are to be stored.
theServer->SetAuthInfo((unsigned char*)”auth_sample”,
                rpc_c_authn_dce_secret,
                (void*)”authpwd”);

// Activate the server object
theServer->Listen();
}
```

In this code fragment, the principal name and the authentication service type are set, but the default key retrieval mechanism is not changed. In OODCE, the **`DCEKeyRetriever`** abstract class (declared in the **`DCEServer.H`** file) is defined for security key retrieval. Concrete classes can be derived from this base class and implemented to provide an alternative mechanism for retrieving security keys.

How to Set Client Security Preferences

The client must set the following security properties:

- Authentication service type to be used by the client
- Protection level to be used for communication
- Authorization service type used by the client
- Server principal name.

At the client, security preferences are set by calling **`SetAuthInfo`** on the client class generated by the **`idl++`** compiler (for example, **`authrpc_1_0`** and **`authClient`**. The client can only communicate with a server manager object if these security preferences are compatible with those set at the server. A manager object is an object that implements the remote operations defined in the IDL file. The following code sample shows the client **`SetAuthInfo`** used on the client object for the **`sleeper`** example:

```
main(int argc, char** argv)
{
 // Construct an instance of the of the Sleeper Class
 // Using the constructor that takes a Network Address
 // and a protocol sequence
 sleeper_1_0 authClient = sleeper_1_0((unsigned
char*)argv[1],
                (unsigned char*)”ip”);

// Setup client side security preferences
// The first argument is the server principal name.
// The second argument is the protection level.
// The third argument is the authentication model
// The fourth argument is the client identity which in
// this case is NULL and inherited from the parent
// process i.e. the user is assumed to be logged onto DCE.
// The Fifth argument is the authorization protocol
// which in this case is set check just the names of

// the principals involved
authClient.SetAuthInfo((unsigned_char_t *)”auth_sample”,
                rpc_c_protect_level_pkt_privacy,
                rpc_c_authn_dce_secret,
                (rpc_auth_identity_handle_t)NULL,
          rpc_c_authz_name);

// Invoke remote operation on Sleeper Object
authClient.Sleep(10);
}
```

Notice that the client, not the server, is responsible for establishing the protection level and authorization model. If the server program needs to enforce certain values on these properties, it must perform its own checking via a reference monitor.

How to use a Reference Monitor

By default, once security preferences have been set in the client program, the DCE runtime automatically invokes the appropriate security code to authenticate the server, applies appropriate encryption to RPC calls, and provides access to authorization data. To provide mutual authentication (that is, allow the server to check that the client is as claimed) and to ensure that the server is willing to meet the client preferences for protection and authorization, a reference monitor must be used.

A reference monitor can perform basic checks before any application code is entered. In general, these checks are as follows:

- Is the Client program authenticated (i.e., has the client principal established a login context?)
- Does the protection level requested by the client meet the requirements of the server program?
- Does the authorization model requested by the client meet the requirements of the server program?

In OODCE, a reference monitor is automatically invoked from the entry point stub code generated by the **`idl++`** compiler (see “Entry Point Vector and Code” in Chapter 2). OODCE defines an abstract base class for a reference monitor object. This class is called **`DCERefMon`**. The **`DCERefMon`** class (declared in the **`DCERefMon.H`** file) is an abstract class that provides an interface to reference monitor type functionality. Multiple reference monitor implementations may be required. These implementations can be provided through classes that are derived from **`DCERefMon`** and reused across a number of manager objects and systems. OODCE supplies a default implementation of a reference monitor with a concrete class called **`DCEStdRefMon`**. You can write more specialized implementations.

A **`DCERefMon`** object can be registered with an interface exported by a manager object. A single manager object at the server may have different reference monitors for different interfaces it implements.

Also, many manager objects may share the same reference monitor object or different reference monitor policies can be supported by one interface.

A reference monitor check is made before a member function is called on a manager object. The following figure shows the flow of control for an RPC call into a manager object that has a reference monitor.

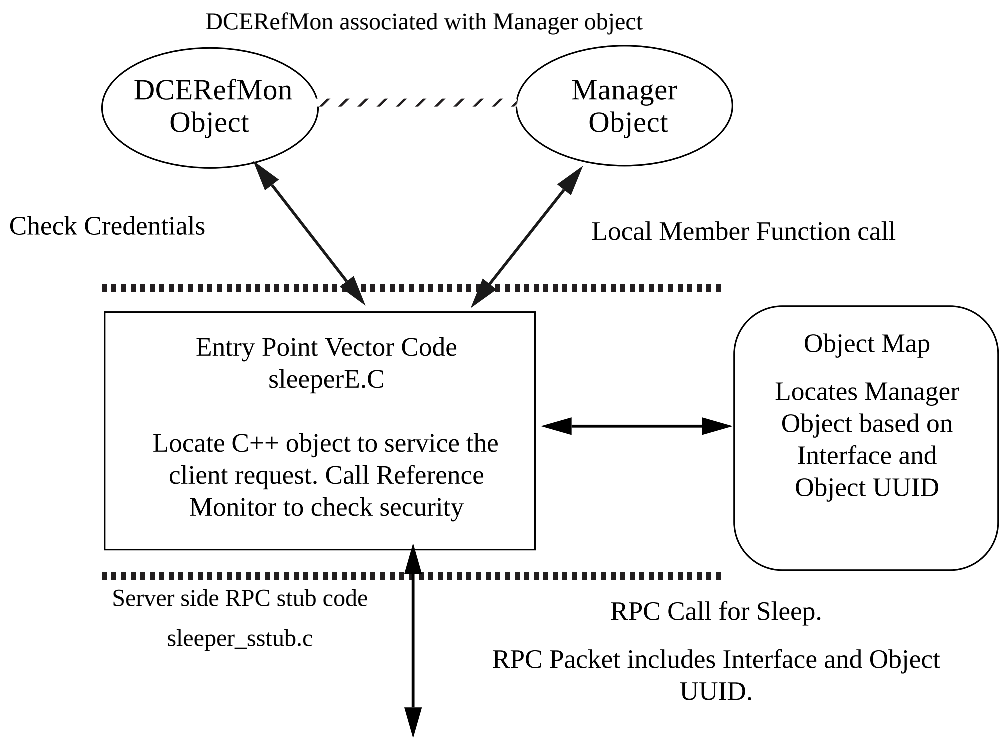
DCERefMon associated with Manager object
DCERefMon Object
Manager Object
Check Credentials
Local Member Function call
Entry Point Vector Code sleeperE.C
Locate C++ object to service the client request. Call Reference Monitor to check security
Object Map
Locates Manager Object based on Interface and Object UUID
Server side RPC stub code
sleeper_sstub.c
RPC Call for Sleep.
RPC Packet includes Interface and Object UUID.

Figure 6-1. Invocation of Reference Monitor Functionality

How to use Default Reference Monitor in Server Code

The following code fragment shows the **`DCEStdRefMon`** implementation in the **`sleeper`** example.

```
void main()
{

     // Construct sleeper manager object
     sleeper_1_0_Mgr sleeperobj;
     // Create cleanup thread
     DCEPthread* exitThd = new DCEPthread(
          DCEServer::ServerCleanup,
          (void*)(0));

     // DCEReference monitor object
     DCEStdRefMon*  thisRefMon;


// Create instance of the reference monitor object
// and initialize the server preferences.
     thisRefMon = new DCEStdRefMon(
          rpc_c_protect_level_pkt_privacy,
               rpc_c_authn_dce_secret,
               rpc_c_authz_name);

// Register the reference monitor with the
// sleeper manager object
sleeperobj.SetRefMon(thisRefMon);

     // Register interface object with server object
     theServer->RegisterObject(sleeperobj);

     // Activate the server object
     theServer->Listen();

}
```

7

Basic Access Control List Management

OODCE supports DCE authorization by making it easier for the server developer to create, manage, and consult Access Control Lists (ACL) for an authorization decision. OODCE supports the DCE standard **`rdacl`** interface. The **`acl_edit`** tool can be used to edit ACLs managed by the OODCE ACL Management subsystem. See the *DCE Application Development Guide* for a description of DCE ACL management.

Some ACL management classes can be used directly in every application. Others that require additional functionality can be reimplemented and fit within the same framework. This section describes how to use the functionality that OODCE supports directly. Chapter 9, Advanced ACL Management, describes how to write custom ACL management classes that extend the basic functionality.

Unlike a reference monitor, which provides a global security policy, ACL management allows the enforcement of a policy concerning who may perform which operations on which objects (in this context, an object is a named piece of data or state). ACL management is usually provided when the server manages many different named objects, each with its own authorization policy.

NOTE Up to this point in the manual, the **`sleeper`** application has demonstrated the step-by-step construction of a distributed application. The **`sleeper`** application does not need ACL management, because it does not manage multiple objects (or a state). However, the **`sleeper`** example will continue to be used in order to focus on what code needs to be added to an application to implement ACL management functionality. For a better example of an application for which ACL management is more appropriately used, see the **`dispatch`** or **`aclsleeper`** OODCE sample application.

The general steps for including ACL management in an application are as follows:

1. Initialize the ACL system.
2. Create a database object as an ACL repository.
3. Add and delete ACLs to and from the database.
4. Use the ACL manager to make authorization decisions.

How to Initialize the ACL Management System

Every server that uses the OODCE ACL management features must have only one **`DCEAclMgr`** and only one **`DCEAclStorageManager`** C++ object. The **`DCEAclStorageManager`** is a global object that is created automatically at load time. To get access to this global object, you must include **`DCEAclStorageManager.H`** in one of your source files. In the **`aclsleeper`** example, it is included in **`sleeper.C`**.

When using the current implementation of the Acl Management objects, the **`DCEAclMgr`** global object must be created explicitly by your server code by using the **`DefineAclMgr`** macro. The macro takes a reference to a **`DCERefMon`** object as its first parameter, therefore, a **`DCERefMon`** object must be created before the macro invocation in the server source. The following is a sample of code from the server main function that shows the initialization in context with the rest of **`sleeper`** server.

```
#include <DCEAclMgr.H>
#include “sleeperS.H”
// thisRefMon must be global in order to initialize the
// global variable acl_manager hidden inside the
// DefineAclMgr macro.
DCEStdRefMon* thisRefMon = new DCEStdRefMon(
     rpc_c_protect_level_none,
     rpc_c_authn_none,
     rpc_c_authz_none);

DCEUuid obj_uuid(“34c53cfa-9b3d-11cc-adaf-080009627155”);
DefineAclMgr(*thisRefMon, obj_uuid, “sleeper server”);

void main()
{
  aclsleeper_Mgr sleeper = aclsleeper_Mgr(obj_uuid);
  try{
  theServer->RegisterObject(sleeper, true);
```

```
    unsigned_char_t *cds_name =
        “/.:/subsys/HP/sample-apps/aclsleeper”;
    theServer->SetName(cds_name);

    theServer->SetAuthInfo(
       (unsigned char*)”auth_sample”,
       rpc_c_authn_dce_secret,
       (void*)”authpwd”);

   theServer->Listen();

  }
```

In this sample, a reference monitor object is set up to allow unauthenticated access. This allows users of **acl_edit** to access the ACLs even if the users are unauthenticated. For general security preferences, construct a **DCERefMon** accordingly. The end point mapper uses the information in the object UUID parameter to distinguish your ACL manager from other servers that are also exporting the **rdacl** interface. The interface UUID is the same for all servers doing ACL management. (The object UUID will distinguish between applications that support the **rdacl** interface.)

It is needed because the same interface UUID is exported by all such servers. The object UUID is also registered with CDS (to enable **acl_edit** to contact the right server) using the **DCEServer::SetName** function.

Creating a New Constructor

A constructor is a convenient place to put code that must be executed only once. When using ACL management, a manager object needs access to an **DCEAclSchema** and an **DCEAclDb** object. If your application only uses one manager object or if the **DCEAclSchema** and **DCEAclDb** objects are allocated locally to each of several manager objects, the manager’s constructor is a convenient place to create and initialize these objects.

In all of the previous **sleeper** examples, the **Sleeper_1_0_Mgr** generated manager class was used, because it was sufficient. However, the **Sleeper_1_0_Mgr** defines a NULL body for the manager constructor. For this ACL management example, it is more convenient to define a **sleeper** manager class that inherits from the generated **sleeper** abstract class, but uses the private state to store the schema and database pointers and initializes these pointers in the constructor. The new class, **aclsleeper_Mgr**, is defined as follows:

```
#ifndef sleeper_H_defined
#define sleeper_H_defined
#include <InterfaceMgr.H> // DCEInterface Manager Base
Class
#include <sleeperS.H> // needed for ABS definition
#include <DCEObj.H> // DCEInterface Manager Base Class
#include “sleeper.h” // IDL Generated header file

class DCEAclSchema;
class DCEAclDb;

extern sleeper_1_0_epv_t sleeper_1_0_mgr;
extern rpc_if_handle_t sleeper_1_0_s_ifspec;

class aclsleeper_Mgr : public sleeper_1_0_ABS {
public:
          // Declare Class Constructors
 aclsleeper_Mgr(uuid_t* obj);
```

```
  aclsleeper_Mgr();

      // Declare Class member functions
      // These correspond to the remote procedures
      // declared in sleeper.idl
      // These need to be implemented by the developer

      virtual   void Sleep(
       /* [in] */ idl_long_int time
      );

 private:
  DCEAclSchema *_schema;
  DCEAclDb *_database;

  void CreateAclSchema();
  void CreateAclDatabase();
  void CreateAcl();
 };

 #endif
```

The constructor implementation is as follows:

```
  aclsleeper_Mgr::aclsleeper_Mgr(uuid_t* obj):
      DCEObj(obj),
      sleeper_1_0_ABS(obj, (uuid_t*)(0))
 {
   CreateAclSchema();
   CreateAclDatabase();
   CreateAcl();
 }
```

How to Create a New ACL Schema

An ACL schema specifies a set of permissions that are valid and meaningful within an ACL. A permission is represented by three data items:

- A user friendly way of referring to the permission. This is called the *printstring*. For example: the token “r” represents read permission.
- A description of the permission semantics. This is called the *helpstring*. The standard helpstring for the read permission is “read.”
- A specification of the bit in a permission bitmap that represents the permission. The permission bits defined by DCE can be found in <**`dce/aclbase.h`**>.

The ACL schema allows both the application and **`acl_edit`** to use symbolic permission names rather than bitstrings.

There are two ways to create a new schema:

- Pass a static table of permissions to the **`DCEAclSchema`** constructor.
- Create an empty **`DCEAclSchema`** and add permissions to it.

Passing a Static Table of Permissions to the `DCEAclSchema` Constructor

Using this static table approach creates a non-expandable schema. New permissions cannot be added after the schema is constructed. The following code fragment is taken from **`Sleeper.C`** and is the implementation of a private method called from the **`sleeper`** manager constructor. It shows the use of a static table in defining an **`DCEAclSchema`**.

```
#include <DCEAclDb.H>
#include <DCEAclSchema.H>
#include <DCEModifyableAcl.H>
#include <DCEAclStorageManager.H>

void aclsleeper_Mgr::CreateAclSchema()
{
  const int NUM_PRINTSTRINGS = 2;
  const sec_acl_permset_t sec_acl_perm_sleep =
         sec_acl_perm_unused_00000800;
  static sec_acl_printstring_t sleeper_printstrings[] = {
      { “s”, “sleep”, sec_acl_perm_sleep},
      { “c”, “control”, sec_acl_perm_control}
   };
   _schema = new DCEAclSchema(1,  // number of slices
                               NUM_PRINTSTRINGS,
                               sleeper_printstrings);
   _schema->SetControlPermissions(sec_acl_perm_control);
}
```

Creating an Empty `DCEAclSchema` and Adding Permissions

You can create an empty **`DCEAclSchema`** and add permissions to it. This creates an expandable schema, although schemas should not be changed without careful consideration once data is stored in a database using that schema. If an expandable table is needed, the following code could replace the **`DCEAclSchema`** constructor invocation:

```
_schema = new DCEAclSchema;
_schema->AddPrintstring(
    “s”,
    “sleep”,
    sec_acl_perm_sleep);
_schema->AddPrintstring(
    “c”,
    “control”,
    sec_acl_perm_control);
```

The **DCEAclSchema** should include a permission specification for governing write access to the ACLs. DCE uses **sec_acl_perm_control** for this purpose, but OODCE has not hard coded this policy. Therefore, you must invoke **SetControlPermissions** to specify which permission is to be used for this purpose.

sec_acl_perm_sleep is not defined in the standard DCE header file, <**dce/aclbase.h**>. It must be defined using one of the available permission bits.

How to Create a New Database

This section describes how to instantiate a new ACL database. The **DCEAclDb** implementation does not support persistence. It maintains a binary tree of ACLs in memory, which is destroyed when the server is shut down. To create a new database, use the ACL schema described in the section “How to Create a New ACL Schema,” earlier in this chapter. The database can be created by calling the following function from your server main function, your factory constructor, or from a manager member function.

```
// from Sleeper.C
void aclsleeper_Mgr::CreateAclDatabase()
{
   _database = acl_storage_mgr.CreateNewDatabase(
      “Sleeperdb”,
      _schema);
 }
```

In this example, the database object pointer is kept local to the manager object because it need not be shared across multiple objects. If the pointer needs to be global, it cannot be initialized until after main starts to execute. **CreateNewDatabase** may not be invoked to initialize a global pointer in its definition.

This is because **`CreateNewDatabase`** cannot be successfully called until the global acl_storage_manager pointer is initialized. There is no way to tell the loader which of these global objects should be constructed first. If the loader tries to create a global database object before it creates the acl_storage_manager, **`CreateNewDatabase`** fails.

How to Create a New ACL

The **`rdacl`** interface does not provide a way to create a new ACL. It can only edit existing ACLs. Your application must add ACLs into the database programmatically using the ACL management object interfaces, as shown in the following code.

```
// Sleeper.C
void aclsleeper_Mgr::CreateAcl()
{
  DCEModifyableAcl *macl = database-
>CreateAcl(“sleeperobj”);
  // user:cell_admin:c
  macl->AddAclEntry(_schema->MakeBitmap(“c”),
                     sec_acl_e_type_user,
                     new DCESecId(“cell_admin”,
                               secDomainPerson));

  // group:nsa:sc
  macl->AddAclEntry(_schema->MakeBitmap(“sc”),
                     sec_acl_e_type_group,
                    new DCESecId(“cssi”, secDomainGroup));

  // any_other:s
  macl->AddAclEntry(_schema->MakeBitmap(“s”),
                     sec_acl_e_type_any_other);
  macl->CommitAcl();
}
```

“**sleeperobj**” is the key that finds this ACL once it is in the database. In an application that manages multiple objects, you can choose to name each ACL with the name of the object it protects. This code, which is implemented as a private member function of the **sleeper** manager object, is called by the constructor. It creates an ACL that is consulted when a **sleep** request is received.

NOTE The ACL named “**sleeperobj**” could be edited using the following **acl_edit** command:

```
acl_edit /.:/subsys/HP/sample-apps/aclsleeper/sleeperobj
```

See the **dispatch** sample application’s create_queue_acl routine for a more complete example of creating different kinds of ACL entry types.

When all ACL entries have been inserted, commit the ACL to the database using **CommitAcl**. When this call completes, the ACL can be consulted in authorization decisions, and the modifiable ACL pointer, macl, may no longer be referenced.

How to Check Authorization Privileges.

The most important facility provided by the ACL management library is making authorization decisions following the standard DCE ACL checking algorithm. This section describes how application managers can use OODCE to make an authorization decision.

In most cases, your manager will find it convenient to use the **IsAuth** function in the database object. Using the database pointer, invoke **IsAuth** as follows:

```
// from Sleeper.C
void aclsleeper_Mgr::Sleep(long int time)
{
   if (_database->IsAuth(“sleeperobj”,
                        _schema->MakeBitmap(“s”))){
     sleep((unsigned int)time);
   }
   else
     traceobj << “Not authorized to sleep\n”;
}
```

The first argument is the name of the ACL. In this example, if the requesting client has permission to sleep, **`IsAuth`** returns TRUE, otherwise it returns FALSE.

`IsAuth` is also a member function of the **`DCEAcl`** object. If you have already looked up an **`DCEAcl`** object, **`IsAuth`** can be invoked directly through the **`DCEAcl`** object without suppling the object name.

8

Advanced Application Development

This chapter describes some of the advanced features of OODCE for developing object-based systems on DCE. This chapter uses advanced **C++** concepts. A working knowledge of the **C++** language is required to fully understand the information that follows.

How to Implement a Custom Naming Policy

Some applications may have specialized naming requirements that cannot be supported by the implementation of the OODCE library defaults. In some cases, an alternative naming system may be required (e.g., shared files) to achieve performance or functional requirements of a system. OODCE supports custom naming policies through overloading of the member functions that deal with the Cell Directory Service (CDS) in the default case.

Modifying Server Export and UnExport Functions

To support a custom naming policy within a server program, a new class should be created that is derived from the basic **DCEServer** class. The implementation of this new class should reimplement the **Export** and **UnExport** member functions. The **Export** member function exports or publishes server binding information into the naming system. **UnExport** removes that information when it is no longer valid or required. The following is an example of such a class definition:

```
class NewServer: public DCEServer {
public:
     NewServer(): DCEServer() {} // Must call base class
     void Export(); // overload Export from base
     void UnExport(); // overload UnExport from base
};
```

Notice that the constructor for the **`NewServer`** class calls the constructor for the base class **`DCEServer`**. This must be done so that the global object pointer called **`theServer`** (the Global Server Object) gets set to point to a **`NewServer`** object when a **`NewServer`** class is constructed.

The following code fragment shows the **`NewServer`** class used in the **`sleeper`** server main program.

```
main()
{
     // Create an instance of NewServer
     NewServer* newServer = new NewServer;
     // Above constructor sets theServer to point
     // to newServer.

     // Create single instance of sleeper object
     sleeper_1_0_Mgr* obj = new sleeper_1_0_Mgr;

     // Start cleanup thread
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);

     // Register sleeper object with DCEServer
     theServer->RegisterObject(obj);

     // Set the name of the server object to use CDS
     theServer->SetName(
          “/.:/subsys/HP/sample-apps/Sleeper”);

     // Listen for client requests
     theServer->Listen();
}
```

The constructor for the **`NewServer`** class creates the Global Server Object (GSO) and assigns its pointer to the global variable **`theServer`**. After construction, **newServer** becomes the GSO but can still be accessed via the **`theServer`** variable.

When the **`Listen`** member function call is made on the GSO, it calls the overridden version of **`Export`**. The implementation of **`Export`** is expected to register location information into the naming system. When the program exits, or when the cleanup code executes, the overridden version of **`UnExport`** is called to remove location information from the naming system.

Two protected member functions are provided with the **`DCEServer`** class definition to support the development of custom naming policies. These protected member functions are available to the derived classes. They are as follows:

- **_GetObjectList —** returns a pointer to a list of **`Object_Set_t`** structures (defined in **`Server.H`**), which contains object and export information for every object registered with the server object.
- **_GetInterfaceList** — returns a pointer to a list of each unique interface provided by the objects.

The information provided by these functions and binding information that can be obtained from the RPC runtime using the **rpc_server_inq_bindings** call can be used to implement the overridden versions of **`Export`** and **`UnExport`**.

How to Support Custom Naming for the Client

If a custom naming policy is used, the client object needs to be able to locate server objects based on this policy. To achieve this, the default location code of the client object must be overridden. The client class generated by the **`idl++`** compiler defines a number of virtual functions that can be overridden. The client class includes protected member variables that can be used and initialized by the location code (defined in **`Interface.H`**).

To change the semantics of the default location code of client objects, you must reimplement the **`BindInterface`** member function defined in the **`DCEInterface`** class. This member function can be used to locate server manager objects using a suitable scheme for the application. **`BindInterface`** may use any protected member variable defined in the **`DCEInterface`** class to accomplish this location.

The overridden **`BindInterface`** must, as part of its implementation, initialize the following protected state variables:

- **_handle** — Holds the DCE binding handle to the manager object at the server.
- **_service_bound** — Boolean flag that indicates that binding has occurred. This should be set to TRUE on return from **`BindInterface`**.
- **_object** — Contains the UUID of the server object to which the instance refers. Setting this member variable is optional and depends on whether or not object information was specified prior to **`BindInterface`** being called.

Creating a New Client Class to Support Custom Naming

The following class definition shows how to provide a new client class that overrides the **`BindInterface`**.

```
class NewSleeper : public Sleeper_1_0 {
public:
   NewSleeper(unsigned char* name);//Must call base class
              sleeper_1_0(name)) {};
   void BindInterface(); // Override base class
                         // BindInterface
};
```

The client code that uses this new version of **`sleeper`** is similar to other clients that have been described and is as follows.

```
main()
{
   // Create instance of client object
   NewSleeper sleepObj;

   // Call remote sleep operation
   sleepObj.Sleep(10);
}
```

This code calls the overridden version of **`BindInterface`** to locate a server object to perform the sleep operation. The actual call to **`BindInterface`** is performed automatically by the constructor and hidden from the application code.

How to Develop Manager Objects with Multiple Interfaces

The current DCE IDL does not support the concept of inheritance; for example, each DCE interface is self-contained and there is no subclassing or inheritance at the interface level. This means that an IDL interface definition cannot be derived from another one. OODCE does not change this, and IDL syntax and interface semantics remain as they are. In OODCE, however, **`C++`** inheritance can be used to produce complex objects that support multiple DCE interfaces. This allows you to create general purpose IDL specifications and then use **`idl++`** to generate server side **`C++`** classes. These server side classes can then be combined using **`C++`** inheritance into a composite class that can support the Remote Procedure Calls (RPC) of all of the IDL interfaces supported by the composite class.

How to use `C++` Inheritance for Server Development

In the original **`sleeper`** server, the DCE object supports a single interface specified in **`sleeper.idl`**. Now an extra requirement will be added to the **`sleeper`** object. It will maintain statistical information on its use. The **`sleeper`** object will maintain a value indicating the maximum, minimum, and mean sleep times since it was created.

This support could be developed by simply creating a new version of **`sleeper.idl`** and adding additional Remote Method Calls (RMC) calls to retrieve these values. Since the original specification of the **`Sleep`** RPC is not being changed, this can be called a minor version change. This means that existing **`sleeper`** clients can still call the new version of the interface.

However, an interface that exports statistical information such as maximum, minimum, and mean could have a more general purpose use with other remote objects. For the new version of the **`sleeper`** server, a new interface called **`stat`**, defined in **`stat.idl`**, can be created. The new **`sleeper`** manager object on the server will implement both the **`sleeper`** and the **`stat`** interfaces.

The IDL specification for **`stat`** is as follows:

```
[uuid(8cf8f504-b5cf-11cc-b95f-080009627155),
version(1.0),
interface stat
{
     [idempotent] long Max([in] handle_t h);
     [idempotent] long Min([in] handle_t h);
     [idempotent] long Mean([in] handle_t h);
}
```

Both **`sleeper.idl`** and **`stat.idl`** are passed through the **`idl++`** compiler to generate the client- and server-side classes for the IDL files. The server-side manager classes can then be combined using **`C++`** inheritance to produce a new class that includes the member functions for the RPCs defined in **`sleeper.idl`** and **`stat.idl`**. The following is an example of such a class:

```
class SleeperStat:public virtual DCEObj,
     public sleeper_1_0_ABS,
     public stat_1_0_ABS {
private:
     unsigned32 max; // max sleep time
     unsigned32 min; // min sleep time
     unsigned32 calls; // Number of times called
     unsigned32 total; // total time slept
public:
     SleeperStat(uuid_t* obj):
     DCEObj(obj),
     sleeper_1_0_ABS(obj, (uuid_t*)(0)),
     stat_1_0_ABS(obj, (uuid_t*)(0)) {
          max = 0;
          min = 0;
          calls = 0;
          mean = 0;
     }

     virtual void Sleep(idl_long_int time);
     virtual idl_long_int Max();
     virtual idl_long_int Min();
     virtual idl_long_int Mean();
};
```

SleeperStat combines the functionality of the **sleeper** and **stat** interfaces. It is derived from the abstract classes generated by **idl++**, namely **sleeper_1_0_ABS** and **stat_1_0_ABS**. The **SleeperStat** class is also inherited from the **DCEObj** class. In this case, virtual inheritance is used instead of regular inheritance. The **sleeper_1_0_ABS** and **stat_1_0_ABS** classes are also derived from **DCEObj** using virtual inheritance. This is important since it forces a single copy of a **DCEObj** in an instance of the **SleeperStat** class. The **DCEObj** class stores information about the DCE interfaces implemented by a particular class. This is used by the GSO to register interface and object information with DCE.

If virtual inheritance was not used, a **SleeperStat** object would have three copies of a **DCEObj** and the GSO would not know which one to use.

The other important part of the **SleeperStat** class is the constructor. Notice that each class is initialized from the **SleeperStat** constructor. Basically **DCEObj** is initialized first and then passed to the constructor for **sleeper** and **stat** classes.

The **SleeperStat** class description can be inherited by other classes. The virtual inheritance of the **DCEObj** class allows it to be included with other classes generated by the **idl++** compiler and still forces a single **DCEObj** copy in the resulting class. This allows interface composition to continue as long as a single copy of **DCEObj** exists.

To implement the **sleeper** and **stat** IDL interfaces, you must implement the following member functions.

```
//corresponds to Sleep remote procedure call in
//sleeper.idl
SleeperStat::Sleep(_idl_longtime);

//corresponds to max remote procedure call in stat.idl
_idl_long SleeperStat::Max();

//corresponds to min remote procedure call in stat.idl
_idl_long SleeperStat::Min();

//corresponds to mean remote procedure call in stat.idl
_idl_long SleeperStat::Mean();
```

The main function for a server program that implements a **SleeperStat** object is almost identical to that of the simple **sleeper** server example. The only change is the construction of a **SleeperStat** object rather than a **sleeper** object. The result is a single instance of a manager object that implements both the **sleeper** and **stat** interfaces. The code for the new server main function is as follows.

```
main()
{
     // Create single instance of SleeperStat object
     SleeperStat*obj = new SleeperStat((uuid_t*)(0));
     // Start Signal handler to do cleanup
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);
     // Register sleeper object with Server
     theServer->RegisterObject(obj);
     // Listen for client requests
     theServer->Listen();
}
```

Execution of this code results in a server process that exports a single manager object that implements two DCE interfaces.

Client programs that use the original **`Sleeper`** client proxy objects access the **`SleeperStat`** object via its **`sleeper`** interface. Client programs that use a **`stat`** client proxy object call the **`stat`** member functions of the **`SleeperStat`** object via its **`stat`** interface. The client proxy objects can be in either the same or in different programs. For example, both of the following client programs could be used to access the **`SleeperStat`** manager object (**`obj`**) created in the server code.

```
main()
{
     // Create instance of client object
     sleeper_1_0 sleepObj;

     // Call remote sleep operation
     sleepObj.Sleep(10);
}
```

This code calls the **`sleeper`** interface member function of **`SleeperStat`**.

The following code calls the **`stat`** interface member functions of **`SleeperStat`**.

```
main()
{
     // Create instance of client object
     long meanVal;
     stat_1_0 statObj;

     // Call remote sleep operation
     meanVal = statobj.Mean();
}
```

Using C++ Inheritance for Client Access

The client proxy classes generated by the **idl++** compiler can be combined into a single class using **C++** inheritance. The following composite client class accesses **SleeperStat** objects.

```
class SleeperStatClient: public sleeper_1_0,
      public stat_1_0 {};
```

This class inherits the **Sleep** member function from the **sleeper_1_0** class and the **Min**, **Max** and **Mean** member functions from **stat_1_0**. The client program is now as follows.

```
main()
{
      long meanVal;
      // Create instance of client object
      SleeperStatClient sleepStatObj;

      // Call remote sleep operation
      sleepStatObj.Sleep(10);

      // Call remote mean operation
      mean = sleepStatObj.Mean();
}
```

This client program locates a **SleeperStat** object within a server and makes calls on both of the interfaces it supports. Since each class inherited by the **SleeperStatClient** class maintains its own binding information for the manager object, you need only keep the bindings synchronized between the two client classes. This can be enforced in the constructor for **SleeperStatClient** by initializing one class first and then using the binding information for that class in the second class.

Dynamic Object Management

The examples so far involve objects that are declared before the call to server listen. These types of objects are static and are available when the server program executes. OODCE allows objects to be created or activated dynamically (that is, while the server program is listening for client requests) on behalf of a client. The following sections discuss object creation and activation.

Dynamic Object Creation

This section describes how to use OODCE to create objects within a server process. Creation is different than activation in that activation assumes that an object has already been created.

How To Use a Factory to Dynamically Create Manager Objects

A factory is often associated with the creation of objects. OODCE can support a factory by specifying an IDL interface for a factory object. The factory object on the server is then responsible for creating other objects to be managed by the server. The following is an example of an IDL specification for a factory object:

```
[uuid(6bf99034-97f7-11cc-9cb1-080009627155),
 version(1.0)]
interface Factory
{
import “ObjRef.idl”;
DCEObjRefT* Create(
      [in] handle_t h,
      );

void Delete (
      [in] handle_t h,
      [in] uuid_t theobject
     );
}
```

This interface provides remote procedure call specifications for creating and deleting objects at a server. This example uses object references that are described in the section Creating and Using Object References later in this chapter. These object references are alternative names for the objects to which they point.

To show how factory objects work, a simple design change can be made to the **`sleeper`** application. In this new version, the server program can create and delete **`sleeper`** objects on behalf of clients. The new **`sleeper`** client now creates a **`sleeper`** manager object within the server program when the client object is constructed. It deletes that **`sleeper`** manager object when the client object's destructor is run.

Example Main Function for Sleeper Program With Dynamic Objects

The following is the code segment for the new server main function. This version does not create **`sleeper`** manager objects. Note that when this code is executed, no **`sleeper`** objects are created. Instead, only a factory object that can create **`sleeper`** objects is constructed and registered with the GSO.

```
// Define extern for string form of the
// UUID for the sleeper interface.
extern char* sleeper_interface_uuid_str;
main()
{
     // Create single instance of Factory Object
     // Use sleeper interface UUID for Factory
     // object UUID.
     Factory_1_0_Mgr*factory = new Factory_1_0_Mgr(
          sleeper_interface_uuid_str);
     // Start Cleanup Thread
     DCEPthread* clean = new DCEPthread(
          theServer->ServerCleanup, NULL);
     // Register sleeper object with Server
     theServer->RegisterObject(factory);
     // Listen for client requests
     theServer->Listen();
}
```

In the creation of the factory object, it was specified that the factory object UUID be the same as the interface UUID for the **`sleeper`** so that it is easier for clients to associate this factory object with the **`sleeper`** class. The following figure shows the initial state for a **`sleeper`** server with a factory.

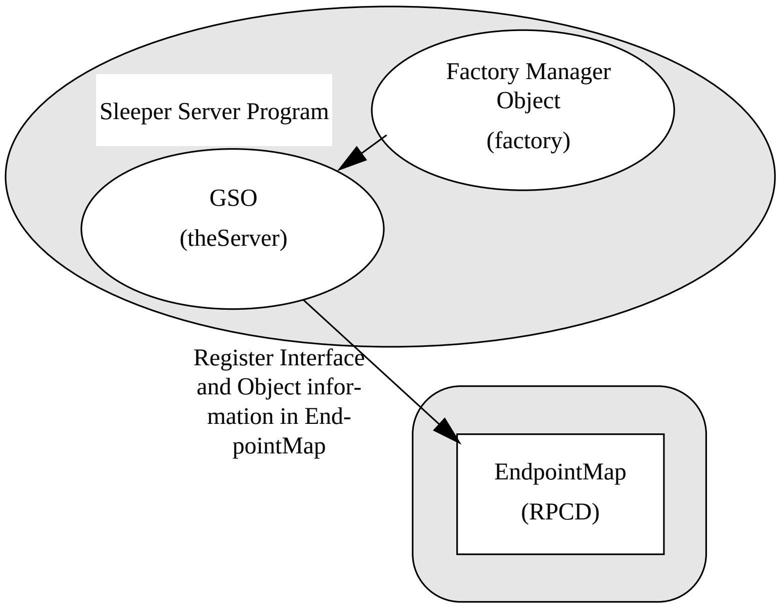

Figure 8-1. Initial State for Sleeper Server with Factory

Example Implementation of a Factory Object

The client proxy classes for the factory interface are generated by the **`idl++`** compiler. The concrete manager class must be implemented to provide the required functionality for the operations defined in the IDL specification (**`Create`** and **`Delete`**). The following code fragment shows the implementation of the factory manager class.

```
DCEObjRefT* Factory_1_0_Mgr::Create()
{
     sleeper_1_0_Mgr* newObject;

     // Create new instance of sleeper object
     newObject = new sleeper_1_0_Mgr;

     // Register sleeper object with Server Object
     theServer->RegisterObject(*newObject);

     // Return object reference to new sleeper object
     return(newObject->GetObjectReference());
}
```

This code creates a new **`sleeper`** manager object and then registers the manager object with the GSO before returning a reference to the newly created object.

```
void Factory_1_0_Mgr::Delete(uuid_t id)
{
     // Remove object represented by id
     // from server object
     theServer->UnregisterObject(&id);
}
```

This code deletes a **`sleeper`** manager object from the GSO and prevents it from being used further by clients. The DCE object UUID for the **`sleeper`** manager object is passed as the only parameter to this function.

The server main function and the factory implementation can now be used to dynamically create **`sleeper`** objects for clients.

Creating Server Manager Objects from a Client Program

This section discusses the client side of the new **`sleeper`** example. The policy specified earlier is to have a **`sleeper`** manager object created on the server when the proxy constructor is invoked, and then have that **`sleeper`** manager object deleted when the client object's destructor is invoked. To do this, a new class on the client is created as follows:

```
class Sleeper_Fac: public sleeper_1_0 {
private:
     Factory_1_0* _factory;

public:
     Sleeper_Fac(char*, char*);
     ~Sleeper_Fac();
};
```

This class inherits from the **`sleeper`** client object and includes a private pointer to the factory client object. A constructor and destructor have been declared and deal with the initialization of the factory client object and in turn the creation of the server-based **`sleeper`** manager object via the factory client. The following is the code fragment for the constructor.

```
// Declare String form of UUID for the
// sleeper interface
char* sleeper_interface_uuid_str =
     “7395E26E-5BA4-11CC-988B-080009253B97”;

Sleeper_Fac::Sleeper_Fac(char* hostaddr, char* protocol)
{
     // Create a factory client object that is used to
     // access a factory manager object at a server.
     // Sleeper manager objects are created via this
     // client object.
     // Note that the DCE object UUID for the
     // factory is equivalent to the interface UUID
     // declared in sleeper.idl

          _factory = new Factory_1_0(
               (unsigned char*)netaddr,
               (unsigned char*)protseq,
          sleeper_interface_uuid_str);

     // Create the sleeper manager object at the server
     // use the Create member function of the factory
// client object to perform creation
            DCEObjRefT* ref = _factory->Create(object);

  //Initialize the binding for this client object
  //from the object reference returned by create
            SetBinding(
            (rpc_binding_handle_t)GetBindingHandle(ref));
  }
```

This code fragment locates and initializes the factory client object that can be used to create **sleeper** manager objects on a server. The factory manager object (that actually creates the **sleeper** manager objects) is located based on an object identifier. This is the same as the interface UUID specified in **sleeper.idl**. In this case, it is not important which instance of a factory is located as long as it is one that can create **sleeper** manager objects.

The server main code for this example of a **sleeper** server program hardcodes the object UUID of the factory to be that of the **sleeper** interface UUID. Once a **sleeper** factory manager object is located, its **Create** member function is called (via the factory client object) to create a **sleeper** manager object at the server. **Create** returns an object reference that is used to set the binding handle for the **Sleeper_Fac** client class using the **SetBinding** member function. The call to **SetBinding** calls **sleeper_1_0::SetBinding** and initializes the binding information for the client object to refer to the **sleeper** manager object just created by the **Create** factory call. The following figure shows the objects in the server program after the client constructor is called.

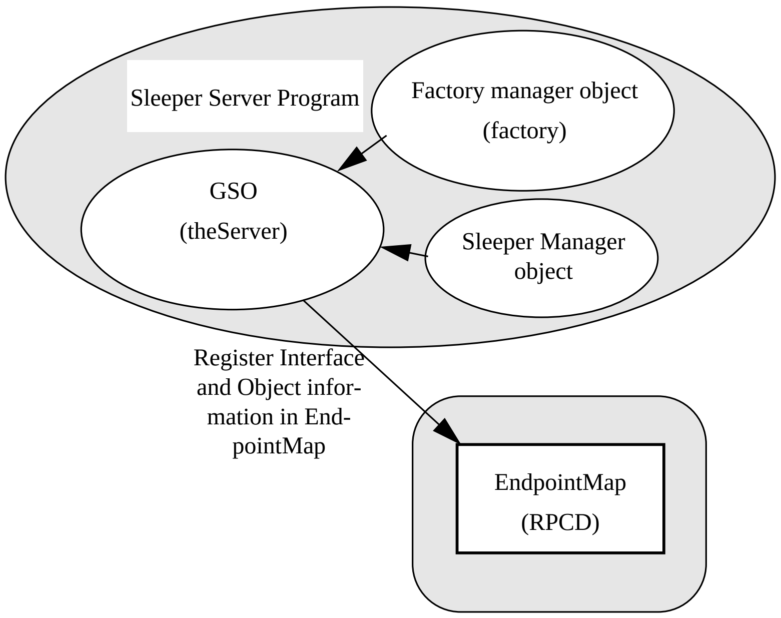

Figure 8-2. Objects In Server Program After Client Constructor Is Called

The following code is the destructor for **`Sleeper_Fac.`**

```
Sleeper_Fac::~Sleeper_Fac()
{
     if(_factory){
          _factory->Delete(_object);
          delete _factory;
     }
}
```

This destructor checks that the factory client object has been initialized and then calls the local **`Delete`** member function to delete the **`sleeper`** manager object at the server. The work is done by the **`Delete`** member function of the factory manager object using the implementation shown in the section Example Implementation of a Factory Object earlier in this chapter. The object UUID passed to **`Delete`** is stored in the protected state of the **`DCEInterface`** class, from which **`Sleeper_Fac`** is derived. The following figure shows the objects in the server program after the destructor is called.

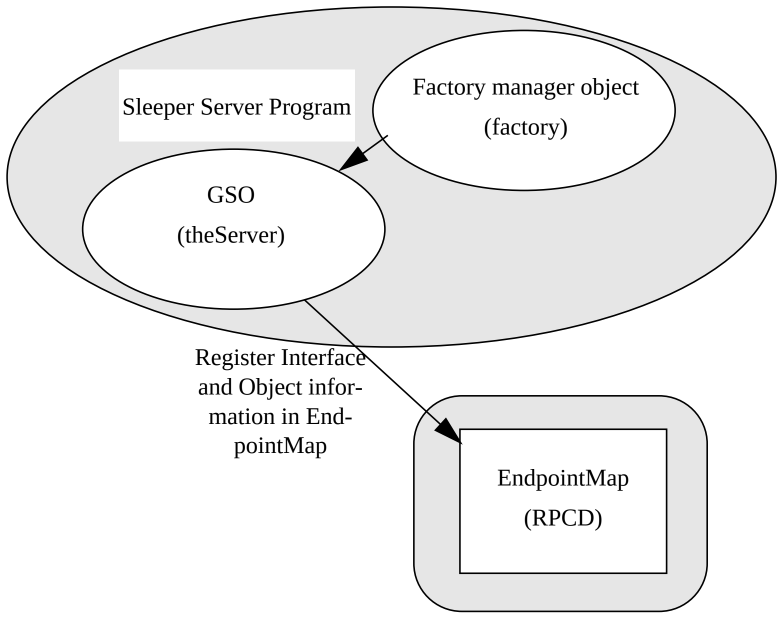

Figure 8-3. Objects In Server Program After Destructor Is Called.

Example Client Program that Dynamically Creates Manager Objects

The client object, when constructed, creates a **`sleeper`** object on a remote server and when destructed deletes that object. The following is a client application that uses this code.

```
main(int argc, char** argv)
{
// Construct Sleeper_Fac object.
// This causes a sleeper object to be created on a
// server. The constructor takes a host address and
// protocol sequence.
     Sleeper_Fac objectClient = Sleeper_Fac(argv[1],
“ip”);

// Call Sleep member function on sleeper object
// that was created
     objectClient.Sleep(10);

// objectClient destructor will be called when function
// exits. This will delete the sleeper manager object on
// the server.
}
```

Running this code creates a **`sleeper`** manager object at a server using a factory to create the object. **`Sleep`** is then called on the newly created manager object. When the client exits, the **`sleeper`** manager object is deleted using the factory object.

Object Activation

If a server program manages a large number of DCE objects, it can be impractical to have all of its objects active at the same time. Activation is different from creation because it assumes that the object has already been created and initialized but is not currently active or available for clients to use (that is, the object is in a passive state). Object activation allows a server designer to choose when it is appropriate to activate an object (normally when a client needs to use it) and do this independent of when a server program is started.

Activation is normally associated with persistent storage. When an object is passive, it usually means that its state has been saved in long term storage (e.g., a file system or a database). OODCE does not provide support for persistent storage of an object's state, but can work with persistent store implementations (such as object databases) to achieve the semantics of object activation.

For an object to be activated, information is required that identifies the object to be activated. In OODCE, this information is represented by the DCE object UUID. Since the semantics of object activation are closely tied to the object's implementation, there is no way for OODCE to provide a general activation mechanism. OODCE does, however, allow you to support activation.

OODCE defines an abstract class called **`Activation`** that provides an abstraction onto the activation of manager objects. You need to derive a concrete class from the **`Activation`** class and provide an implementation that performs the activation for a specific object class or set of classes. Typically, this implementation performs the following steps.

1. Locate the object state based on the DCE Object UUID.
2. Create new manager object for the class associated with the Object UUID.
3. Initialize the new manager object with the state.
4. Register the manager object with the GSO.

To demonstrate activation, a few design changes to the **`SleeperStat`** server example can be made. The **`SleeperStat`** server program will now have the following characteristics. A single **`SleeperStat`** manager object, which supports both the **`sleeper`** and the **`stat`** interfaces, is created by the server program and registered with the GSO.

This manager object continuously checkpoints itself after a set period of time. If there have been no client requests on the manager object between two checkpoints, its state (in this case statistical information) is stored in the UNIX file system and the manager object is unregistered from the server program. If a client request is made on this manager object, it is reactivated from the stored state on the file system.

The **`SleeperStat`** client program is modified such that it calls the **`Sleep`** member function of a **`SleeperStat`** manager object. It then waits for a period that exceeds twice the checkpoint period of the manager object before making a second sleep call. The second sleep call requires that the manager object be reactivated at the server before the call can complete.

Modifications to SleeperStat Manager Object to Support Activation

The original **`SleeperStat`** manager object did not support persistence. It must be modified to save and restore its state from the UNIX file system. The modification to support persistence is done in the **`SleeperStat`** object constructor and through an additional member function that supports checkpointing. These modifications are shown in the following code fragments.

```
sleeper_stat::sleeper_stat(uuid_t* obj):
 DCEObj(obj),
 sleeper_1_0_ABS(obj, (uuid_t*)(0)),
 stat_1_0_ABS(obj, (uuid_t*)(0))
{
     struct stat tmp;

     ////
     // Check to see if this is a persistent object
     ////
     if(stat((char*)(this->GetId()), &tmp) == -1) {
     ////
     // Initialize state to Zero
     ////
     max = 0;
     min = 0;
```

```
        calls = 0;
        total = 0;
        state = fopen((char*)(this->GetId()), “w”);
        fprintf(state,
        “%d %d %d %d”, max, min, calls, total);
        fclose(state);

        } else {
        ////
        // Read in the state from file
        ////
        state = fopen((char*)(this->GetId()), “r”);
        fscanf(state,
        “%d %d %d %d”, &max, &min, &calls, &total);
        fclose(state);
        }
        ////
        // Start the checkpoint thread
        ////
        checkpoint = new DCEPthread(checkpointhandler,
         (void*)this);
   }
```

The **`SleeperStat`** constructor is passed an object UUID. This UUID is used to check if this object has been created previously. This is achieved by using the string form of the object UUID as the UNIX file name for where the object state is stored. If the file exists, the state of the object is read in from the file, otherwise, the **`SleeperStat`** object’s state is initialized to zero. Before the constructor exits, it creates a checkpoint thread that is used to checkpoint the objects state every five minutes.

The following code fragment shows the implementation of the checkpoint member function declared for the **SleeperStat** class. This member function is called by the checkpoint thread created by the constructor. When it is called, it writes the object’s state to the UNIX file system. If there has been no change in the state since the last checkpoint call, the member function returns a boolean indicating that the **SleeperStat** object can be deactivated or deleted from the GSO of the server program.

```
boolean32 sleeper_stat::CheckpointAndRemove(
     unsigned32* numcalls)
{
     ////
     // Check if the object has been accessed
     // since last checkpoint
     ////
     if(*numcalls == calls) {
     ////
     // No change since last checkpoint.
     // Save the current state and exit
     ////
     state = fopen((char*)(this->GetId()), “w”);
     fprintf(state,
     “%d %d %d %d”, max, min, calls, total);
     fclose(state);
     return true; // Caller can delete
     } else{
     ////
     // Save current state
     ///
     *numcalls = calls;
     state = fopen((char*)(this->GetId()), “w”);
     fprintf(state,
          “%d %d %d %d”, max, min, calls, total);
     fclose(state);
     return false;
     }
}
```

Example Implementation of Activation Code for `SleeperStat`

A new class needs to be derived from the base **`DCEActivation`** abstract class. This new class must be implemented to support the activation of **`SleeperStat`** objects. The following is the activation class for the **`SleeperStat`** example:

```
class SleepStatActivator: public DCEActivation {
public:
 DCEActivationResultT *ActivateObject(DCEUuid& object);
};
```

In general, a single member function needs to be implemented called **`ActivateObject`**. The **`ActivateObject`** member function takes a UUID as the single parameter and returns a pointer to a **`DCEActivationResultT`** structure, as follows.

```
DCEActivationResultT
     *SleepStatActivator::ActivateObject(DCEUuid& object)
{
     DCEActivationResultT* ret_act = 0;
     struct stattmp;
     sleeper_stat* obj;

     ////
     // Check and see if object state is stored
     // in the file system
     ////
     if(stat((char*)(object), &tmp) == -1)
     // Object never existed return NULL
     return ret_act;

     ////
     // Initialize activation result
     ////
     ret_act = new DCEActivationResultT;
     ret_act->referral = 0;
     ret_act->object_active = false;

     ////
```

```
        // Create new sleeper manager object pass
        // object UUID to constructor.
        // Constructor will initialize state from
        // file system.
        ////
        obj = new sleeper_stat(object);

        ////
        // Registered sleeper manager object with GSO
        ////
        theServer->RegisterObject(*obj);
        ret_act->object_active = true;

        ////
        // Return activation result
        ////
        return ret_act;
   }
```

Registering An Activation Object

To support activation, a server program must register an activation object implementation with the GSO. This is done in the server program’s main function, as follows.

```
void main()
{
     // Create instance of sleeper_stat manager object
     sleeper_stat* sleeperStat = new sleeper_stat(0);

     // Activate cleanup thread
     DCEPthread*exit = new DCEPthread(
          DCEServer::ServerCleanup, (void *)(0));

     // Create activator object for sleeper_stat
     SleepStatActivator activator;

     // Create a UUID object for the sleeper interface
     // UUID. see sleeper.idl
     DCEUuidsleeper_if(
     “7395E26E-5BA4-11CC-988B-080009253B97”);

     // Register manager object
     theServer->RegisterObject((DCEObj&)(*sleeperStat));

     // Set activation object. sleeper_if indicates
     // that activation is done through sleeper
     // interface of the sleeper stat object
     theServer->SetActivationObject(&activator,
          sleeper_if);

     // Call listen on the DCEServer
     theServer->Listen();
}
```

When this code executes, it creates a **`SleeperStat`** server with a single manager object. An activation object is registered with the GSO using the **`SetActivationObject`** member function. This member function is called with the activation object and an interface UUID. Activation is done when a client makes a call on an interface. The GSO needs to know which interfaces support activation. In this example, activation only occurs when a client object makes a call to the server program using the **`sleeper`** interface. No activation is done for the **`stat`** interface.

NOTE **`SetActivationObject`** can be called more than once for different interfaces or a vector of interface UUIDs can be passed to the member function instead of a single UUID used in the example.

The manager object checkpoints itself by storing its statistical state in the UNIX file system. If there has been no activity on the manager object between two checkpoints, the object is unregistered and deleted from the server program (that is, the server will have no manager objects). Subsequent calls by clients to the server program activate the object.

Example Client That Activates a Manager Object at the Server.

Once the server program has been started and the initial manager object has been registered in the endpoint map, any **`sleeper`** client program described so far will activate that manger object if necessary. The following example creates a **`sleeper`** client object and calls its **`Sleep`** member function. The client program then waits ten minutes before making a call to **`Sleep`** on the same object again. The second call to **`Sleep`** requires that the server program activate the object.

```
main()
{
     // Create instance of sleeper client object
     sleeper_1_0 sleepObj;

     // Call remote sleep operation
     // of sleeper_stat manager object
     sleepObj.Sleep(10);

     // wait for object to checkpoint
     sleep(600);

     // Call remote sleep operation
     // of sleeper_stat manager object
     // This will cause the object to
     // get activated
     sleepObj.Sleep(10)
}
```

Creating and Using Object References

It is often convenient to pass a reference to a manager object within a server to another program in a distributed system. This allows multiple programs to collaborate with a single manager object. An object reference uniquely refers to a specific DCE object. It can be passed as an RPC parameter and then converted into a client object reference that can be used to access the server manager object it refers to.

An example of object references is shown in the factory example in the section “Example Implementation of a Factory Object” earlier in this chapter. In this case, a manager object is created in a server program and a reference to that object is passed back to the client. The client used this reference to create a client object to access the newly created manager object.

OODCE ensures that object references are type safe. You can only create a client object from an object reference that is of a class that is compatible with the manager class of the manager object associated with the reference. Checks are made at the server to verify that the referenced object supports the requested interface of the client object.

A reference to a manager object can be obtained by calling the **`GetObjectReference`** member function as shown in the implementation of the factory example in the section “Example Implementation of a Factory Object” earlier in this chapter.

There is an OODCE data type called **DCEObjRefT**. This data type has been declared using IDL. It can be used as parameter type in other IDL specifications by using the IDL **`import`** statement in the IDL file. The following code shows the IDL specification for the **`Factory`** interface. The data type specification can be found in the **`ObjRef.idl`** file in the OODCE include directory.

```
[uuid(6bf99034-97f7-11cc-9cb1-080009627155),
 version(1.0)]
interface Factory
{
import “ObjRef.idl”;

DCEObjRefT* Create(
      [in] handle_th,
      );

void Delete (
      [in] handle_th,
      [in] uuid_t theobject
     );
}
```

9

Advanced ACL Management

How to Write Your Own Database Implementation

Writing your own ACL database implementation involves creating an implementation class that inherits from the **`DCEAclDb`** abstract class. You must implement all of the functions in the abstract class, but you can add more. For example, to create an ACL database with persistence, you would write the constructor to find the persistent data and do the necessary conversions needed to make it accessible to the program. If you keep an in-memory copy of the database that is distinct from its persistent representation, you must design a strategy to keep these data stores in synchronization and to minimize the loss of data if a program exits or a system crashes. One way to do this is to have the constructor create a checkpoint thread that periodically ensures that the data stores contain the same information. Another way to do this is to have all manager routines that write the database write both the in-memory and persistent copies.

`DCEAclDb`, **`DCEAcl`**, and **`DCEModifyableAcl`** work closely together to manage the ACL storage and retrieval. Therefore, when reimplementing one of these, the design of the others must also be considered. Where to store locks, where to store owner information, and how and when to free resources needs to be agreed upon among all three.

How to Instantiate an Existing (Persistent) Database

The library does not support persistence directly, but it does let you provide your own persistent implementation of **`DCEAclDb`**. If an implementation called **`AclDbPersist`** inherits from the **`DCEAclDb`** abstract class, you must create the database by invoking the constructor and registering the **`AclDbPersist`** object with the **`DCEAclStorageManager`** object.

An existing data store must have some record of the database schema such that an **`DCEAclSchema`** object can be created as a result of constructing an **`AclDbPersist`**. The following is an example of the code that is needed to create the database:

```
DCEAclDb *the_database; // assume sharing across all
objects


the_database = AclDbPersist(database_name,
                             persistent_file_name...)
 num_slices = the_database->GetSchema()->Num_Slices();
 acl_storage_manager.Register(the_database, num_slices);
```

Multiple Refmon's

You might want to create application management operations within the application interface that manipulates the ACLs instead of the application objects. A client of such an interface would be required to pass the general security check provided by the **`DCERefMon`** object registered with this application interface. However, there might be a different security check imposed on **`rdacl`** interface requests.

If your application manager that implements one of these ACL management operations wants the client also to pass the security check imposed on the **`rdacl`** interface, the ACL manager version of **`IsAuth`** should be used, as follows:

```
acl_manager.IsAuth(the_database,
                   object_name,
                   desired_perms);
```

The other versions of **`IsAuth`** do not invoke the **`rdacl`** **`DCERefMon`** security check because it is assumed that the appropriate DCERefMon has been invoked prior to calling **`IsAuth`**. The ACL manager version of **`IsAuth`** is special because it calls the **`DCERefMon`** object before getting an authorization decision. You may not need to use this version of **`IsAuth`** very often.

One way to get around needing it is to separate management operations into a different application interface, and register the same **`DCERefMon`** object with it as you pass to the **`DefineAclMgr`** macro.

Implementing ACL Management for a Factory Object

Normally a server would only use ACL management when there is state to which it wants to restrict access on a per user and per operation basis. A factory does not maintain state, but might want to use ACL management to implement a multi-tiered authorization policy that would be cumbersome to implement using a reference monitor. For example, you might want to allow the owner and certain special users of the factory to create and delete objects and all members of a group to create new objects. The application server implementing both the factory and the objects created by the factory must decide how to manage the ACL(s) for both the factory and the objects.

One option is to develop a schema that includes both the factory permissions (create and delete) as well as the permission relevant to the objects created by the factory. If the factory creates work order lists, the permissions might be add, flush, and dispatch. If one schema is used for both, the factory ACL can reside in the work list ACL database provided it is given a name that is guaranteed not to not to be the same as the name of a real work list.

Another option is to develop a separate schema and database for the factory ACL. The factory schema would define create and delete permissions, and a separate work order schema would define add, flush, and dispatch. Using different databases allows the factory ACL name to be anything. However, both the factory and the set of application objects would need to register **`DCEAclMgr`** bindings into the namespace in order for **`acl_edit`** to be able to bind to the appropriate database.

10

Advanced Threads Programming

Using DCEPthread Attributes

A Pthread has a number of attributes that determine how it behaves. The most important of these is its scheduling priority. A thread's priority may be set at a higher or lower level. Maximum and minimum levels are available but should only be used by preemptive or idle threads respectively.

Pthreads also have a scheduling policy that controls time slicing policy and the interpretation of low-priority. With the foreground (fg) (default) policy and background (bg) policy, low priority threads eventually get CPU time. Pthread_fg has a medium priority range, and while giving preference to high-priority threads, sees that all threads eventually get some processing time. Pthread_bg has the lowest priority, but is otherwise similar to Pthread_fg. Pthread_fifo (first-in-first-out) and Pthread_rr (round robin) have priority ranges higher than the other algorithms, and equal to each other. Pthread_fifo does not do time slicing among threads with equal priority; consequently it can starve low priority threads. Pthread_rr does time slicing; consequently, low-priority threads cannot starve. The stack size attribute establishes the size of the stack available for the thread.

The **`DCEPthread`** class represents a thread, but it is not itself a thread. A Pthread may be created without actually starting a thread execution, and it may be deleted either before or after the thread terminates. The **`DCEPthread`** class contains attribute values before the thread is started, and provides for control communications with other threads. In the following sections, it is important to recognize the difference between a Pthread and the thread that it represents.

The termination attribute determines the behavior of the Pthread when it is deleted, either by exiting the block in which it is declared or by issuing an explicit call to the **`C++ delete`** function. The default action deletes the block and eliminates the possibility of communicating with the thread. The thread may continue to run. The option **`Pthread_join_on_delete`** waits for the thread to complete before terminating. This can be important if the process terminates or the next program step assumes that the work being done by the thread has already completed.

Setting the Attributes

Attributes may be set and interrogated through DCEPthread operations. Instead of starting the thread when the DCEPthread is created, a separate start operation is performed, as follows:

```
{ DCEPthread checkpoint_thread;

checkpoint_thread.Scheduling(Pthread_fifo);
checkpoint_thread.Priority(Pthread_pri_hi);
checkpoint_thread.Termination(Pthread_join_on_delete);

// checkpoint_thread will now execute at a high priority
// using fifo scheduling (higher precedence than the
// default fg), and on termination will wait for the
// thread itself to complete.

checkpoint_thread.Start(checkpoint, NULL);

} // This deletes checkpoint_thread, waiting for the
  // checkpoint itself to complete.
```

A thread may return a result when it exits. If that result is required, an explicit **`Join`** must be performed on the thread:

```
DCEPthreadResult res = checkpoint_thread.Join();
```

The value can only be returned after the thread terminates. If this is not yet the case, this operation waits for thread termination.

If priority is the only attribute that needs to be specified, there is a shorter way to specify it:

```
DCEPthread checkpoint_thread(Pthread_pri_hi, checkpoint,
                  NULL);
```

Attributes for Multiple Threads

When a number of threads are to be created that have the same attribute settings, a **`DCEPthreadAttr`** can be used, as follows:

```
{ DCEPthreadAttr ckpt_attr;

ckpt_attr.Scheduling(Pthread_fifo);
ckpt_attr.Priority(Pthread_pri_hi);
ckpt_attr.Termination(Pthread_join_on_delete);

DCEPthread checkpoint_1(ckpt_attr, checkpoint, NULL);
DCEPthread checkpoint_2(ckpt_attr, checkpoint, NULL);
DCEPthread checkpoint_3(ckpt_attr, checkpoint, NULL);
DCEPthread checkpoint_4(ckpt_attr, checkpoint, NULL);

} // This deletes checkpoint_thread, waiting for all
  // checkpoint threads to complete.
```

Changing and Interrogating Thread Attributes

All attributes of a DCEPthread can be changed until the thread is started. After that time, all attributes other than the stack size can be changed.

```
checkpoint_thread.Priority(Pthread_pri_hi);
```

If the thread has not started, this changes an attribute value within the **`DCEPthread`** object. Once the thread starts, the priority change happens within the pthread software, locating the pthread and changing its priority.

Attributes can also be interrogated. The same operation name, but without a parameter, is used to access the attribute:

```
DCEPthreadPrio x = checkpoint_thread.Priority();
```

Before the thread starts, this command removes the intended attribute value. After the **`Start`** command, it accesses the process and returns its priority.

When a **`DCEPthread`**'s constructor specifies a **`DCEPthreadAttr`**, the values set in that attribute determine those used for the thread after it is started. Once the thread starts, **`DCEPthreadAttr`** has no effect on the running thread.

```
{ DCEPthreadAttr ckpt_attr;

ckpt_attr.Scheduling(Pthread_fifo);
ckpt_attr.Priority(Pthread_pri_hi);
ckpt_attr.Termination(Pthread_join_on_delete);

DCEPthread checkpoint_1(ckpt_attr);
DCEPthread checkpoint_2(ckpt_attr);
DCEPthread checkpoint_3(ckpt_attr);
DCEPthread checkpoint_4(ckpt_attr);

checkpoint_thread_1.Start(checkpoint, NULL);
checkpoint_thread_3.Priority(Pthread_pri_min);

ckpt_attr.Priority(Pthread_pri_mid);        //statement A
checkpoint_thread_2.Start(checkpoint, NULL);

checkpoint_thread_4.Scheduling(Pthread_rr); //statement B
ckpt_attr.Priority(Pthread_pri_low);        //statement C
checkpoint_thread_3.Start(checkpoint, NULL);

checkpoint_thread_4.Start(checkpoint, NULL);

} // This deletes checkpoint_thread, waiting for all
  // checkpoint threads to complete.
```

This codes executes as follows:

1. checkpoint_thread_1 is started with hi priority. The next statement changes its priority to min, but does not affect other threads or ckpt_attr.

2. Statement A changes the priority of ckpt_attr. The next statement starts checkpoint_thread_2 with a mid priority.
3. Statement C changes the priority of ckpt_attr, so checkpoint_thread_3 is started with low priority. Statement B has no effect on checkpoint_thread_3 since it was directed at a single process.
4. checkpoint_thread_4 is started at priority mid since statement B isolates it from ckpt_attr. As soon as an attribute of a process changes, the values of the parameter attribute are captured and all linkage with that attribute is broken.

Using Thread-Specific Storage

Often, a thread wants to maintain a state of its own that is not shared with other threads. While this storage may be located in a heap, there needs to be an access to the pointer to the data. This is similar to having state data in a class. The **`DCEThreadSpecific`** class provides a way of doing this.

A **`DCEThreadSpecific`** data element is generally declared as a state variable to a Manager class, although it can also be declared globally. **`DCEThreadSpecific`** is a template, so a type name must be specified to provide proper type checking. For most cases, the **`DCEThreadSpecificPtr`** template should be used. This calls a destructor to delete your allocated memory when the thread terminates. **`DCEThreadSpecific`** may be used for managing your own memory or for storing a single integer data item (up to 32 bits).

Suppose the thread-specific information is represented as a class **`MyClass`**. The thread-specific data is created by the following declaration:

```
DCEThreadSpecificPtr <MyClass> data;
```

Inside a thread procedure, establish the value for this thread. For example, the **`checkpoint`** procedure could begin with a statement:

```
void * checkpoint (void *)
{
   data.Set(new MyClass);
}
```

Elsewhere in the thread, perhaps after receiving a new work request, the thread may retrieve the data in the following way:

```
MyClass * foo = data.Get();
```

When the thread terminates, the destructor for **`MyClass`** is called.

Using Condition Variables

Some applications use threads as servers. Threads receive requests to perform work, and wait for new requests when all have been serviced. This is a typical use of condition variables.

A condition variable always has an associated mutex. These are declared as follows:

```
DCEPthreadMutex cond_mutex;
DCEPthreadCond cond(cond_mutex);
```

NOTE When a **`DCEPthreadCond`** is declared as a data member of a class, the initialization parameter must be placed in the constructor.

In addition to these declarations, there is normally a data structure that receives the requests. The data structure may be a **`C++`** template queue that maintains a list of requests, allowing a new request to be added to the end, and allowing requests to be removed from the front of the queue.

```
Queue<Request> request_list;
unsigned32 accepting_requests = 1;
```

The request list is the protected data guarded by **`cond_mutex`**. It must only be touched by a thread that has acquired a lock on the mutex. The **`accepting_requests`** flag indicates that no additional requests can be placed on the list. It is initially TRUE.

The server procedure monitors the **`request_list`** for work to be done as shown in the following code.

```
cond_mutex.Lock();
while(request_list.Empty() && accepting_requests)
    cond.Wait();
Request * current_request = request_list.Pop();
cond_mutex.Unlock();
```

If there are no messages waiting in the request list, this routine waits on the condition variable cond. It is important to recheck the condition upon return from the Wait operation to be certain that the condition has really become true.

Work requests are placed on the queue in the following manner, assuming that the **`accepting_requests`** flag is true, and that there is a request called **`req`**:

```
cond_mutex.Lock();
request_list.Add(req);
cond.Signal();
cond_mutex.Unlock();
```

The **`Signal`** operation wakes up any thread waiting on the condition variable. If there are none, the next available thread discovers the request before waiting on condition.

A

OODCE Glossary of Terms

base class
The class from which another class is derived.

client object
An object that locates and accesses manager objects implemented by server programs.

DCE interface
A set of related operations that can be applied to any object of the class. See also DCE object.

DCE object
An entity that is manipulated by well-defined operations. Every DCE object has a class, which specifies the type or category of the object. All DCE objects of a class are manipulated using a specific set of one or more interfaces. See also DCE interface.

derived class
A class that inherits its members from another class.

endpoint vector
A vector that contains endpoints, which are addresses of specific server instances on a host system.

entrypoint vector
The **`idl++`** generated server entrypoint vector (EPV) is used to interface OODCE with the DCE server stubs generated by the idl compiler. The server-side EVP implements a generic DCE EPV for a DCE interface. The entrypoint vector is different for each interface specified in an IDL file. The entrypoint vector handles object location and activation within the server and provides the first level security check.

exception

A value defined in IDL that is returned by a request whose execution encounters abnormal conditions, such as invalid input parameters. An exception can also occur if an abnormal condition occurs during the performance of a request, in which case the OODCE returns an exception to the client. Exception handling provides a language-level facility for the uniform handling of program anomalies.

extension classes

Highest level of the OODCE class library structure. They provide default implementations of abstract classes that exist in the framework and provide an implementation of the ACL manager subsystem. See also framework classes.

factory

Executable code that creates objects upon request. Factories have detailed knowledge of the objects they create.

framework classes

OODCE class library classes that represent abstractions onto the DCE C-based API. They are used to implement the DCE object model. They use the Utility classes to manage and manipulate DCE location and identification information.

generated classes

Application specific classes of the OODCE class library. They are generated by the **`idl++`** compiler.

global server object

DCE allows only a single **`rpc_server_listen`** call per process. Therefore, a process can have only one instance of a server object. For convenience, the OODCE library contains a Global Server Object (GSO) called **`theServer`** that can be used by server programs. The implementation of the **`DCEServer`** class is thread safe; therefore, the global server object can be safely accessed from any thread within the server process.

idl++

An instrumented compiler that processes the RPC interface definition specified in the Interface Definition Language (IDL). The **`idl++`** compiler generates the client and server stub files, and the files that specify the abstract and concrete manager classes, the entrypoint vector, and the server and client functionality.

manager object

An object that implements remote operations defined in the IDL file.

marshalling

RPC: The process by which a stub converts local arguments into network data and packages the network data for transmission.

OODCE class

Specifies the type or category of a DCE object and the set of operations that can be applied to the object. See also DCE object.

OODCE Class Library

A C++ class library for the Open Software Foundations's Distributed Computing Environment (DCE).

Pthread

An API (application programming interface) standard for threads functionality, specified by POSIX in 1003.4a, Draft 4.

reference monitor

Code that controls access to an object. Servers control access to the objects they maintain; and for a given object, the ACL Manager associated with that object makes authorization decisions concerning the object.

server object

An object that provides a response to a request for a service. An object can be a client for some requests and a server for other requests.

stub

RPC: A code module specific to an RPC interface that is generated by the DCE **`idl++`** compiler to support remote procedure calls for the interface. RPC stubs are linked with client and server applications and hide the intricacies of remote procedure calls from the application code.

surrogate object

A language-mapping-specific object created by the **`idl++`** compiler that corresponds to a single interface operation. A client issues a static request by calling the surrogate object, which transmits the request to the corresponding method.

B

Basic Application Development Summary

This appendix presents s a step-by-step guide to application development. This appendix refers to previously described details.

Use the following steps to develop an application:

1. Define the client server interface using the DCE Interface Definition Language (IDL). Specify remote procedures that must be implemented to provide the required functionality.
2. Compile the IDL file with the OODCE **idl++** compiler, which generates the following files:

   <**IDL File Name**>**S.H** — Contains **C++** manager class definitions. These are described further in “Manager Classes” in Chapter 2.

   <**IDL File Name**>**E.C** — Implements entry point manager code for the interface. This is described further in “Entry Point Vector and Code” in Chapter 2.

   <**IDL File Name**>**C.H** and <**IDL File Name**>**C.C** — Contain the class declaration and implementation of the client object for the interface. This is described further in “Client Class” in Chapter 2.

   <IDL File Name>**_sstub.c** and <**IDL File Name**>**_cstub.c** — Contain the RPC communication stub code generated by the **idl** compiler. These files are described further in “DCE Stub Files” in Chapter 2.
3. Implement member functions of the manager class called <**InterfaceName**>**_**<**Major Version**>**_**<**Minor Version**>**_Mgr** declared in the <**IDL File Name**>**S.H** file. “Implementing Manager Objects” in Chapter 2 contains an example of this manager class.

4. Implement the server main function for the server program. This includes the following:
   - Construct manager objects implemented in Step 3 and accessed by the server program. The section “Construction of Manager Objects” in Chapter 2 shows an example of this.
   - Create and activate signal handling thread to perform cleanup if the server program is sent a signal. The section “Creating a Signal Handling Thread” in Chapter 2 shows an example of this.
   - Register manager objects with the Global Server Object (GSO). The section “Registering Manager Objects With the Server Object” in Chapter 2 shows an example of this.
   - Select communications protocols for the server program to use when listening for client requests. Chapter 6, Basic Security, describes this further. (Optional.)
   - Set the CDS naming preferences for the server program allowing clients to locate server manager objects via the CDS. Chapter 3, Using the Cell Directory Service, describes this further. (Optional.)
   - Set security preferences for the server program. Chapter 5, Basic Pthreads, describes this further. (Optional.)
   - Call the **`Listen`** member function on the GSO to register manager object information in the CDS and the endpoint map. The section “Listening for Clients” in Chapter 2 shows an example of this.

5. Compile and link source files for the server program. These include the source files containing the server main function and the implementation of the manager class as well as the following:

6. <**`IDL File Name`**>**`E.C`**
   <**`IDL File Name`**>**`_sstub.c`**

   These files must be linked with the OODCE library.

7. Implement the client main function to access manager objects at the server program. This includes the following:
   - Set up exception handling to catch remote errors. The section Communicating OODCE Exceptions from Server to Client in Chapter 4 describes this further. (Optional.)
   - Create a local client class object to locate and access the server manager object. The section “Client Class Example” in Chapter 2 shows an example of this activity.
   - Set client security preferences. The section “How to Set Client Security Preferences” in Chapter 6 describes this further. (Optional.)
   - Make member function calls on the local client class. The section “Client Class Example” in Chapter 2 shows an example of this activity.

8. Compile and link source files for the client program. These files include the client main function written in step 6 and the following files:

   ```
   <IDL File Name>C.C
   <IDL File Name>_cstub.c
   ```

   These files must be linked with the OODCE library.

Index