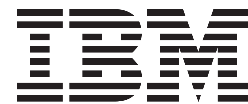
OSF ® DCE Application Development Guide— Introduction and Style Guide Release 1.2.2

December 8, 1998

Open Software Foundation
11 Cambridge Center
Cambridge, MA 02142



OSF ®DCE Application Development Guide– Introduction and Style Guide

Release 1.2.2

The information contained within this document is subject to change without notice.

OSF MAKES NO WARRANTY OF ANY KIND WITH REGARD TO THIS MATERIAL, INCLUDING BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

OSF shall not be liable for errors contained herein, or for any direct or indirect, incidental, special or consequential damages in connection with the furnishing, performance, or use of this material.

Copyright © 1995, 1996 Open Software Foundation, Inc.

This documentation and the software to which it relates are derived in part from materials supplied by the following:

Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1996 Digital Equipment Corporation

Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1996 Hewlett-Packard Company

Copyright © 1989, 1990, 1991, 1992, 1993, 1994, 1995, 1996 Transarc Corporation

Copyright © 1990, 1991 Siemens Nixdorf Informationssysteme AG

Copyright © 1990, 1991, 1992, 1993, 1994, 1995, 1996 International Business Machines

Copyright © 1988, 1989, 1995 Massachusetts Institute of Technology

Copyright © 1979, 1980, 1983, 1986, 1988, 1989, 1991, 1992, 1993, 1994 The Regents of the University of California

Copyright © 1995, 1996 Hitachi, Ltd.

All Rights Reserved

Printed in U.S.A.

THIS DOCUMENT AND THE SOFTWARE DESCRIBED HEREIN ARE FURNISHED UNDER A LICENSE, AND MAY BE USED AND COPIED ONLY IN ACCORDANCE WITH THE TERMS OF SUCH LICENSE AND WITH THE INCLUSION OF THE ABOVE COPYRIGHT NOTICE. TITLE TO AND OWNERSHIP OF THE DOCUMENT AND SOFTWARE REMAIN WITH OSF OR ITS LICENSORS.

Open Software Foundation, OSF, the OSF logo, OSF/1, OSFMotif, and Motif are registered trademarks of the Open Software Foundation, Inc.

X/Open is a registered trademark, and the X device is a trademark, of the X/Open Company Limited.

The Open Group is a trademark of the Open Software Foundation, Inc. and X/Open Company Limited.

UNIX is a registered trademark in the US and other countries, licensed exclusively through X/Open Company Limited.

DEC, DIGITAL, and ULTRIX are registered trademarks of Digital Equipment Corporation.

DECstation 3100 and DECnet are trademarks of Digital Equipment Corporation.

HP, Hewlett-Packard, and LaserJet are trademarks of Hewlett-Packard Company.

Network Computing System and PasswdEtc are registered trademarks of Hewlett-Packard Company.

AFS, Episode, and Transarc are registered trademarks of the Transarc Corporation.

DFS is a trademark of the Transarc Corporation.

Episode is a registered trademark of the Transarc Corporation.

Ethernet is a registered trademark of Xerox Corporation.

AIX and RISC System/6000 are registered trademarks of International Business Machines Corporation.

IBM is a registered trademark of International Business Machines Corporation.

DIR-X is a trademark of Siemens Nixdorf Informationssysteme AG.

MX300i is a trademark of Siemens Nixdorf Informationssysteme AG.

NFS, Network File System, SunOS and Sun Microsystems are trademarks of Sun Microsystems, Inc.

PostScript is a trademark of Adobe Systems Incorporated.

Microsoft, MS-DOS, and Windows are registered trademarks of Microsoft Corp.

NetWare is a registered trademark of Novell, Inc.

FOR U.S. GOVERNMENT CUSTOMERS REGARDING THIS DOCUMENTATION AND THE ASSOCIATED SOFTWARE

These notices shall be marked on any reproduction of this data, in whole or in part.

NOTICE: Not withstanding any other lease or license that may pertain to, or accompany the delivery of, this computer software, the rights of the Government regarding its use, reproduction and disclosure are as set forth in Section 52.227-19 of the FARS Computer Software-Restricted Rights clause.

RESTRICTED RIGHTS NOTICE: Use, duplication, or disclosure by the Government is subject to the restrictions as set forth in subparagraph (c)(1)(ii) of the Rights in Technical Data and Computer Software clause at DFARS 52.227-7013.

RESTRICTED RIGHTS LEGEND: Use, duplication or disclosure by the Government is subject to restrictions as set forth in paragraph (b)(3)(B) of the rights in Technical Data and Computer Software clause in DAR 7-104.9(a). This computer software is submitted with ″restricted rights.″ Use, duplication or disclosure is subject to the restrictions as set forth in NASA FAR SUP 18-52.227-79 (April 1985) ″Commercial Computer Software-Restricted Rights (April 1985).″ If the contract contains the Clause at 18-52.227-74 ″Rights in Data General″ then the ″Alternate III″ clause applies.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract.

Unpublished - All rights reserved under the Copyright Laws of the United States.

This notice shall be marked on any reproduction of this data, in whole or in part.

Contents

Figures ix

Tables xi

Preface xiii
Audience. xiii
Applicability xiii
Purpose xiii
Document Usage. xiii
Related Documents xiii
Typographic and Keying Conventions xiv
Problem Reporting xiv
Pathnames of Directories and Files in DCE Documentation xiv

Chapter 1. Introduction to DCE Application Programming 1
Development Overview 1
Overview of DCE Application Development Steps 3
DCE Application Development Tools. 4
The DCE UUID Generator 4
DCE Interface Definition Language 5
The DCE IDL Compiler 6
The Attribute Configuration File 6
The DCE Host Daemon 6
The DCE API 7
The DCE Control Program 7
The Interface Definition 8
Generating the Interface UUID 8
Writing the Interface Definition File 9
Writing the Attribute Configuration File 10
Processing the Files with the IDL Compiler 10
Server Initialization 11
Setting Up for Serviceability 12
Setting Up the Server's Objects 13
Setting Up Security 15
Defining the Manager Entry Point Vectors for Each Set of Operations . . . 16
Registering the Server 17
Specifying Multithreadedness 19
Listening for Incoming Service Requests 20
Cleaning Up Code When the Server Terminates 21
The Client Binding and RPC Invocation 22
Importing the Binding Information from the Namespace. 22
Annotating the Binding Handle for Security 23
Invoking Remote Procedure Calls 25
The Server's Manager of RPC Requests 26
Getting the Client's Credentials 26
Getting the Object's ACL 27
Making the Authorization Decision 28
Servicing the RPC Request 28
Returning the Results and Resuming Listening. 29
About DCE Programming Style 29
Mechanism, Policy, and Style 30
Policy and Style Issues 31
General Policies 32

Chapter 2. Threads . . . 35
Thread Use Policy . . . 36
Choosing to Thread . . . 36
Specifying the Number of Threads . . . 37
Scheduling Policies . . . 38
Thread Safety . . . 38
Threads Programming Topics . . . 40
Thread Handles . . . 41
Storage for Thread Specific Data . . . 41
Canceling Threads . . . 42
Signals . . . 46
Forking in a Threaded Application . . . 49
RPC Threads and RPC Cancel Semantics . . . 49

Chapter 3. Security . . . 51
The Basic Security Model . . . 51
Application Roles . . . 53
Authentication Model . . . 53
The DCE Authentication Model . . . 54
Application-Level Authentication . . . 55
Obtaining an Authentication Identity . . . 56
The Authenticated RPC Call . . . 57
Managing Keys . . . 58
Default Server Authentication Steps . . . 59
Default Client Authentication Steps . . . 63
Authorization . . . 65
Client Credentials . . . 66
Access Control Lists . . . 67
ACL Managers . . . 69

Chapter 4. Binding . . . 91
The Binding Model . . . 91
Server Binding Model . . . 92
Client Binding Model . . . 95
Call Routing . . . 95
Routing Policy . . . 96
Binding Handles . . . 97
Binding Methods . . . 98

Chapter 5. Using the DCE Name Service . . . 103
Introduction to Using NSI . . . 103
The UUID . . . 104
Object UUIDs . . . 104
Interface UUIDs . . . 104
Summary: Names and UUIDs . . . 105
Binding to an Object . . . 105
Junctions . . . 106
A Junction Example . . . 106
Junctions and the ACL Editor . . . 107
Name Service Terminology . . . 108
CDS Entries . . . 108
CDS Entry Attributes . . . 109
Binding . . . 110
Importing and Exporting Bindings . . . 110
Summary . . . 111
Partial Binding and the Endpoint Mapper . . . 112

Interface Ambiguity and Partial Bindings 113
Using Object UUIDs to Avoid Binding Ambiguity 114
An Object-Oriented Namespace 116
Setting Up an Object-Oriented Namespace 117
Groups and Profiles. 120
Group Entries 120
Profiles 120
Summary of Namespace Entry Types 121
Three Models for Accessing Binding Information 121
Access By Services. 121
Access By Servers 122
Access By Objects 122
Summary of Binding Models 122
Models Based on Non-CDS Databases 123
Example of a Privately Managed Database 124
Combining Models 124
An Object-Oriented Model with Grouped Binding Information. 125
Server and Client Steps 125
Server Export 126
Client Import 127
Global Organization of the Namespace 129

Chapter 6. RPC Parameters 131
Execution Semantics 131
Parameter Semantics 132
Parameter Memory Management 133
Client Side Allocation 134
Server Side Allocation 134
RPC Data Types 135
IDL to C Type Mappings 135
Character Handling 137
Pointers 138
Context Handles 145
Arrays. 145
Structures and Unions 149
Pipes 152
The transmit_as Attribute 152

Chapter 7. Errors and Messaging 155
Error Handling. 155
Messaging Facilities 156
DCE Errors and DCE Messages 156
DCE Application Message APIs 157
Serviceability and Logging 157
Sample Code 159

Chapter 8. Object-Oriented Applications with Distributed Objects 165
Kinds of Objects 165
Reference Counting: How Objects Keep Track of Multiple Clients 166
Using Interface Definitions to Design Classes 167
Using Static Functions in Interface Design 168
Adding an Interface Rather than Changing One 168
Binding to Distributed Objects Rather than Servers 169
Clients Manipulate Objects Maintained on Servers 169
Naming Objects 170

Chapter 9. Server Management 173
Application Support for Server Management 174
Manager Initialization 174

Chapter 10. A Sample Application 177
The Generic Server 177
Object Bind Interface 228
Manager and Client Illustrations 231
Message (sams) File 249
Makefile 252

Index 257

Figures

1. The Combined Effect of IDL and the RPC Runtime 2
2. Information Required to Complete an RPC 92
3. Server Binding Relationships 94
4. Methods of Binding Management 99
5. How a Name Turns into an Object 105
6. A Namespace Junction. 107
7. Client and Server Use of the Name Service 111
8. The Endpoint Mapper Service Completes a Binding 113
9. Print Server Entries in Namespace 114
10. Print Server Name Entries with Object UUIDs 115
11. Separate Printer Name Entries 116
12. Object-Oriented Namespace Organization. 119
13. The Export Operation in a Model with Grouped Bindings 127
14. Importing from a Model That Uses Grouped Bindings 129
15. The RPC Class Hierarchy. 167
16. Distributed Objects and a Remote Procedure Call 170
17. Server and Object Names in the Name Service. 171
18. Managing a Server with a Control Client 173

Tables

1. Thread-Safe Calls 39
2. Cancelability State 42
3. Authentication 52
4. Binding Semantics 101
5. Some Examples of Objects 116
6. Parameter Semantics 133
7. IDL/NDR/C Type Mappings: Part 1 136
8. IDL/NDR/C Type Mappings: Part 2 136

Preface

The *OSF DCE Application Development Guide* provides information about how to program the application programming interfaces (APIs) provided for each OSF® Distributed Computing Environment (DCE) component.

Audience

This guide is written for application programmers with UNIX operating system and C language experience who want to develop and write applications to run on DCE.

Applicability

This revision applies to the OSF® DCE Release 1.2.2 offering and related updates. See your software license for details.

Purpose

The purpose of this guide is to assist programmers in developing applications that use DCE. After reading this guide, you should be able to program the Application Programming Interfaces provided for each DCE component.

Document Usage

The *OSF DCE Application Development Guide* consists of three books, as follows:

- *OSF DCE Application Development Guide—Introduction and Style Guide*
- *OSF DCE Application Development Guide—Core Components*
  - Part 1. DCE Facilities
  - Part 2. DCE Threads
  - Part 3. DCE Remote Procedure Call
  - Part 4. DCE Distributed Time Service
  - Part 5. DCE Security Service
- *OSF DCE Application Development Guide—Directory Services*
  - Part 1. DCE Directory Service
  - Part 2. CDS Application Programming
  - Part 3. GDS Application Programming
  - Part 4. XDS/XOM Supplementary Information

Related Documents

For additional information about the Distributed Computing Environment, refer to the following documents:

- *Introduction to OSF DCE*
- *OSF DCE Administration Commands Reference*
- *OSF DCE Application Development Reference*
- *OSF DCE Administration Guide*
- *OSF DCE DFS Administration Guide and Reference*

- *OSF DCE GDS Administration Guide and Reference*
- *OSF DCE/File-Access Administration Guide and Reference*
- *OSF DCE/File-Access User's Guide*
- *OSF DCE Problem Determination Guide*
- *OSF DCE Testing Guide*
- *OSF DCE/File-Access FVT User's Guide*
- *Application Environment Specification/Distributed Computing*
- *OSF DCE Technical Supplement*
- *OSF DCE Release Notes*

Typographic and Keying Conventions

This guide uses the following typographic conventions:

Bold
: **Bold** words or characters represent system elements that you must use literally, such as commands, options, and pathnames.

Italic
: *Italic* words or characters represent variable values that you must supply. *Italic* type is also used to introduce a new DCE term.

`Constant width`
: Examples and information that the system displays appear in `constant width` typeface.

[ ]
: Brackets enclose optional items in format and syntax descriptions.

{ }
: Braces enclose a list from which you must choose an item in format and syntax descriptions.

|
: A vertical bar separates items in a list of choices.

< >
: Angle brackets enclose the name of a key on the keyboard.

...
: Horizontal ellipsis points indicate that you can repeat the preceding item one or more times.

This guide uses the following keying conventions:

<Ctrl-x> or ˆ x
: The notation **<Ctrl-x>** or **ˆ x** followed by the name of a key indicates a control character sequence. For example, **<Ctrl-c>** means that you hold down the control key while pressing **<c>**.

Return
: The **Return** notation refers to the key on your terminal or workstation that is labeled with the word Return or Enter, or with a left arrow.

Problem Reporting

If you have any problems with the software or documentation, please contact your software vendor's customer service department.

Pathnames of Directories and Files in DCE Documentation

For a list of the pathnames for directories and files referred to in this guide, see the *OSF DCE Administration Guide—Introduction* and *OSF DCE Testing Guide*.

Chapter 1. Introduction to DCE Application Programming

The majority of this first chapter consists of a fairly detailed overview of each of the separate steps that a developer usually has to perform (or have the application perform) from the beginning of coding to the end of execution of a successful DCE application.

Before you begin a serious study of the contents of any part of this guide, or indeed of any other book in the DCE documentation set, you should read the *Introduction to OSF DCE*. It contains clear and comprehensive overviews, with illustrations, of all the DCE components and of the integrated DCE as a whole; many concepts and details are explained there that are necessary to a full understanding of what is described here.

If you do not find information about topics you are interested in either in this guide or in the *OSF DCE Application Development Reference*, you should also look in the *OSF DCE Administration Guide* and the *OSF DCE Administration Commands Reference*. For example, the DCE Cell Directory Service (CDS) is not accessed directly by applications (except through DCE RPC NSI or through XDS) so most of the discussion of CDS as a separate component is found in the administration documentation. Although the DCE Security Service is documented in the development books, certain aspects of it important to application developers (for example, adding new principals to the security registry database) are found only in the administration books.

Several key methods underlie the successful development of DCE applications programs. These methods, explained in this chapter, are as follows:

- A set of tools for distinguishing the component applications programs, for describing how they work together, and for manipulating and managing DCE components both locally and remotely.
- A method for establishing the interface between the component parts.
- Methods to install and register a server, so that clients can use it.
- Methods to set up clients so they can use servers.

Development Overview

Most of the effort of developing a DCE application usually lies in the familiar steps of planning, writing and compiling the necessary C code, linking the result with the DCE library and other modules, and executing it (perhaps repeatedly). However, there is an important preliminary task which must be performed before you write any other code. Before you can implement the application's client and server, you must write and compile an interface definition file in which you define the application's client/server interface.

This interface, defined in the DCE Interface Definition Language (IDL), consists of a set of *prototypes* for the remote procedure calls your client(s) will be requesting your server(s) to execute. After you have written this file, you compile it with the DCE IDL compiler. The final output of IDL compilation is a pair of object files, one for the server module and one for the client, which you must later link with the compiled output of your server and client implementation code. These two IDL output files contain the server and client stub code, where all the details of remote execution, data transfer, and so on, are managed (in conjunction with the DCE runtime).

The IDL compiler also generates a header file for inclusion in the server and client source files. It contains all the declarations that result from the IDL file definitions. Among these are, for example, the interface specification identifier, which will be used at runtime to describe the interface being defined in the programs.

Once you have linked the stub files (and the DCE library) to their respective client and server modules, the IDL-generated stubs make the client and server seem to communicate directly through the operation signatures you defined in the original **.idl** file, although in actuality client/server communications pass back and forth through layers of stub and runtime processing, which are necessary to send and receive the data over the network. Figure 1 illustrates how the combination of IDL (by means of the stubs it generates) and the RPC runtime routines shields both client and server from the details of network communications.

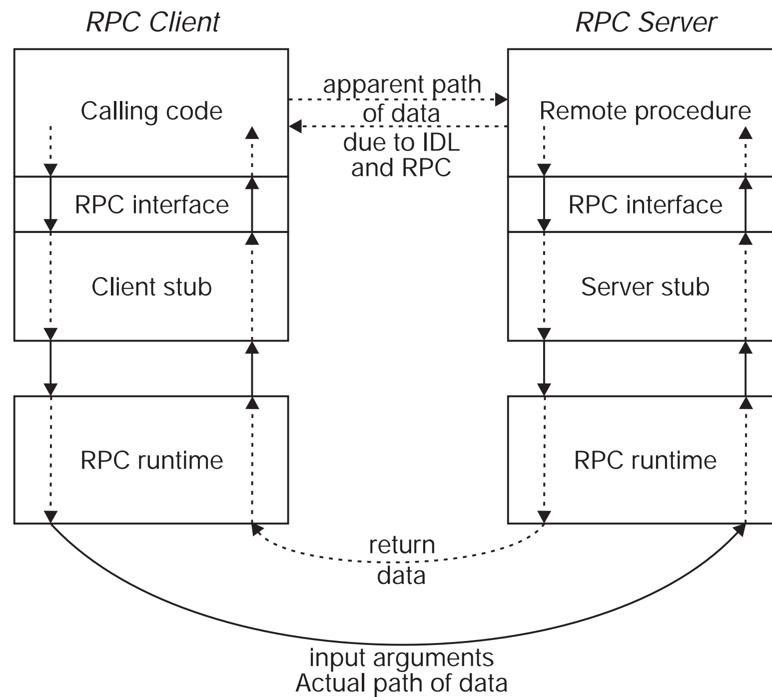


Figure 1. The Combined Effect of IDL and the RPC Runtime

Once the work of defining an interface has been completed, the task of implementing the interface (that is, coding the operations, along with the rest of the necessary initialization and management routines, in some programming language) begins. The rest of this chapter consists of detailed explanations of the DCE application development steps from start to finish.For a practical example of the result of such a process, refer to the code for the DCE sample application, which is reprinted in full in the appendix of this book.

Each of the DCE components (with the exception of CDS, which is accessed through the RPC NS API) is discussed in depth in separate parts of this guide. You should also refer often to the *OSF DCE Application Development Reference*, which contains reference pages for all of the DCE library routines mentioned in the following sections.

Overview of DCE Application Development Steps

The rest of this chapter consists of a step-by-step checklist of every single one of the decisions that a programmer must make in developing a typical DCE application. Each set of decisions or choices is combined into one step. The combination of all these steps takes you from the initial coding stages into and through the normal course of execution of the application itself. The underlying intention of this arrangement is to give you a useful mental model of the overall code development process.

The four basic phases of DCE application development are as follows:

A. **CLIENT and SERVER:** Define the IDL interface [Steps **A1** to **A4**]

B. **SERVER:** Set up and listen [Steps **B1** to **B8**]

C. **CLIENT:** Bind to and invoke the server [Steps **C1** to **C4**]

D. **SERVER:** Service request(s) [Steps **D1** to **D5**]

Following is an overview list of all 21 steps, separated into the four main phases previously described. Each step's numeral is followed by a **/** (slash) and the terms **Client** and/or **Server** to indicate whether it applies to the application's server or client, or both.

A. **CLIENT and SERVER:** Define the IDL interface

- **A1/Client and Server:**
  Generate the interface UUID
- **A2/Client and Server:**
  Write the **.idl** file
- **A3/Client and Server:**
  Write the **.acf** file (optional)
- **A4/Client and Server:**
  Process the files with the IDL compiler

B. **SERVER:** Initialization

- **B1/Server:**
  Set up for serviceability
- **B2/Server:**
  Set up the server's objects
- **B3/Server:**
  Set up security
- **B4/Server:**
  Define the manager entry point vectors
- **B5/Server:**
  Register the server
- **B6/Server:**
  Specify multithreadedness
- **B7/Server:**
  Listen for incoming service requests
- **B8/Server:**
  Clean up when server terminates

C. **CLIENT:** Bind to and invoke the server

C1/Client:
Multithreaded client design

C2/Client:
Import the binding information from the namespace (CDS)

C3/Client:
Annotate the binding handle for security

C4/Client:
Invoke an RPC interface operation

D. **SERVER:** Service the request

D1/Server:
Get the client's credentials

D2/Server:
Get the object's access control list

D3/Server:
Make the authorization decision

D4/Server:
Service the request

D5/Server:
Return the results to the client and resume listening

DCE Application Development Tools

The following DCE tools allow developers to define and manage a set of programs intended to run in a DCE environment.

- Unique identification

  Because DCE involves the interaction of many distinct programs, operating on several processors that may be quite remote from each other, every entity (such as programs, interface definitions, and so forth) needs a unique identifier. This identifier is provided by the UUID generator.

- Interface definition language

  Applications programs that are to work within DCE can be written in any of several programming languages. The two halves of a client/server pair need not be in the same language. In order to permit this flexibility, each application's client/server interface uses a common language, called IDL. It is supported by an IDL Compiler.

- Attribute configuration language

  To allow developers to control the interface between local applications code and the RPC interface, there is an optional attribute configuration language supported by the IDL compiler.

- Remote DCE management

  A host daemon (**dced**) and a control program (**dcecp**) provide capabilities for management of a host and its servers.

The DCE UUID Generator

The UUID generator **uuidgen** is an interactive utility that creates UUIDs (universal unique identifiers). A UUID is a hexadecimal number that contains information that

makes it unique from all other UUIDs. Applications use UUIDs to identify many kinds of entities, including interface definitions. Consequently, application developers typically use the UUID generator when they are creating their interface definition files.

To run the UUID generator, issue the **uuidgen** command. This command offers several options, including an option to create a template interface definition file (an **.idl** file) containing a newly generated interface UUID. For complete information about generating UUIDs and template interface definition files, see the *OSF DCE Application Development Guide—Core Components*. See this volume also for a discussion of UUIDs and their use in DCE applications. Refer to the **uuidgen(1rpc)** reference page for a description of the **uuidgen** utility and its options.

DCE Interface Definition Language

As was mentioned earlier in this chapter, developing a DCE application involves writing and compiling an interface definition, which defines the application's client/server interface. Application developers use IDL to write the interface definition. IDL is a high-level descriptive language whose syntax resembles that of ANSI C. IDL is a declarative, not a procedural, language. Some of the important attributes specified with IDL are the following:

- For interfaces

  uuid Specifies the interface's UUID.

  version
  Specifies the interface major and minor version number.

- For parameters

  in Signifies a parameter whose value is passed from the client to the server.

  out Signifies a parameter whose value is passed from the server to the client.

- For data types

  handle
  Specifies a customized binding handle. Chapter 4 (on "Binding Handles" on page 97) discusses binding handles and binding methods in more detail.

  context_handle

  Specifies a context handle, which is a pointer to state information that the server uses and which is maintained across RPC invocations. An example of a context handle is a file pointer. For more information about context handles, see the *OSF DCE Application Development Guide—Core Components*.

IDL's operation attributes include specifiers for execution semantics: whether the operation can be safely executed more than once, whether a response is expected, and so on. The default is that operations can be executed at-most-once. Parameters (the arguments supplied by the client when it makes the remote call) can be specified as input to the server, output to the client, or both. See the *OSF DCE Application Development Guide—Core Components* for a complete description of IDL syntax and usage.

The DCE IDL Compiler

The DCE IDL compiler **idl** processes interface definitions written in IDL and generates header files and stub object code. (The compiler generates source code for the stubs in ANSI C.) The code generated from an interface definition by the compiler includes client and server stubs.

The compiler also generates a data structure called the interface specification, which contains identifying and descriptive information about the compiled interface, and creates a companion global variable, the interface handle, which is a reference to the interface specification. Each header file generated by the IDL compiler contains the reference the application code needs to access the interface handle. The interface handle allows the application code to refer to the interface specification in calls to the RPC runtime. Runtime operations obtain required information about the interface, such as its UUID and version numbers, directly from the interface specification.

You run the IDL compiler by issuing the **idl** command. See the **idl(1rpc)** reference page for a description of the **idl** command and its options.

The Attribute Configuration File

Application developers can use an optional attribute configuration file to tailor how an RPC interface appears to local application code and how the local application code interacts with the RPC interface. The attribute configuration file is written in the attribute configuration language, which is a companion language to IDL. When the IDL compiler is invoked, it searches for an attribute configuration file in addition to processing the interface definition file.

An attribute configuration file modifies how the IDL compiler interprets an interface definition. For example, an attribute configuration file can specify a subset of operations declarations for a client stub so that the client stub contains declarations for only the operations that the client application code needs for its remote procedure calls. Limiting the client's access to the remote procedures offered by servers reduces the size of the client stub. Another action you can control with an attribute configuration file is defining how a client establishes a binding with a server that implements the called interface.

For complete information on the set of attribute configuration file attributes, see the *OSF DCE Application Development Guide—Core Components*.

The DCE Host Daemon

Each DCE host runs a DCE host daemon (**dced**) to provide remote DCE management services for a host and its servers. The **dced** provides remote management of DCE-related host and server data, it provides remote control of a host's servers, and it maintains host-specific state for DCE such as the host's login identity. From the server's perspective, **dced** is a central point where all servers can consistently inform their host about themselves. From the host's perspective, **dced** gives clients, management applications, and DCE administrators (via **dcecp**) a focal point from which to find out about (and even control) servers.

The most important feature of the **dced** is that it provides the endpoint mapper service. This service maintains the host's local endpoint map for local RPC servers and looks up endpoints for RPC clients. An endpoint is the address of a specific

instance of a server that is executing in a particular address space on a given host. Each endpoint can be used on a host by only one server at a time. The endpoint map is the system-specific database on each host, in which servers register their endpoints and associated addressing information (information about communication protocols, objects, and so on). A server registers separate endpoints for each of its RPC interfaces and any objects the server offers with the interface.

If a client makes a remote procedure call to a host without providing an endpoint, the **dced** searches its endpoint map for the endpoint of a compatible server. Upon finding a suitable endpoint, the endpoint mapper service (depending on the protocols used) forwards the call to that endpoint or returns the endpoint to the client's runtime, which sends the call to the server at that endpoint.

Other remote services of **dced** include host data management, server control, security validation, and key table management. These are described in detail in the *OSF DCE Application Development Guide—Core Components*.

The DCE API

DCE provides a wide range of application programming interface routines. All of the following are available:

- A set of general DCE routines routines provide the means for configuration, handling messages, using the backing store, and managing the DCE daemon, as well as other purposes.
- The DCE thread routines provide thread control, including thread creation, conditional waiting, priorities, and locks.
- The DCE remote procedure call routines provide tools to establish and manage servers, and also include utilities for use by clients and by servers.
- The DCE directory service routines are a set of X/OPEN directory service routines that provide access to the Global Directory Service (GDS) and CDS.
- The DCE distributed time service routines obtain timestamps, translate between timestamp formats, and perform time calculations. The routines can be used from server or clerk systems to determine event sequencing, duration, and scheduling.
- The DCE security service routines allow developers to create network services with complete access to all the authentication and authorization capabilities of DCE Security Service and facilities.

The DCE Control Program

Although the DCE control program, **dcecp**, is intended as an administrator's tool, developers will find it invaluable for examining and modifing many aspects of the DCE environment. It can be used in constructing installation scripts, as in the following examples:

- For exporting binding information to a namespace, instead of putting C code in your application to call the NSI routines, **rpc_ns_ *( )**, you could write a **dcecp** script that calls **rpcentry export** and its related commands.
- For installation, you might need to create a principal name and/or set an access control list (ACL) on it. Instead of writing C code in your application's initialization section to call **sec_rgy_pgo_ *( )** and **sec_acl_ *( )**, you could ship a **dcecp** script that includes the following:

```
principal create ...
acl mod /.:/sec/principal/...
```

- It is recommended that you have **dced** start your application by using server configuration information. It is generally better to do this by writing a **dcecp** script that sets up the server configuration information (the arguments to start the executable) rather than doing it with C code that calls the **dced_server_create()** API.

In general, **dcecp** scripts for server configuration allow better flexibility than embedded C code. Furthermore, unlike embedded code, the script does not persist after configuration is done.

The DCE control program can also be useful for debugging, as the following examples show:

- You can check exported infoformation in the namespace with **rpcentry show** or **rpcgroup, rpcprofile show**.
- You can use **server ping** to see if your server is running and receiving requests.
- If your server was set up to be started by **dced**, you can start it by using the **server start** command and can view the startup parameters by using **server show -executing**.

The Interface Definition

Once you have designed your DCE application and have decided which procedures are needed, and which will be remote procedures, the next step in developing the application is to write one or more interface definitions that describe the remote procedures your application's clients will be requesting your application's servers to run.

To create an interface definition, use the following steps:

1. Generate an interface UUID and a skeleton **.idl** file with the **uuidgen** utility.
2. Write your interface operation declarations in IDL, using the skeleton **.idl** file you generated with **uuidgen** as a base.
3. Write the attribute configuration file. This is an optional step that you take only if you want to alter the IDL output in various ways.
4. Compile the completed interface definition file with the IDL compiler.

The following sections describe these steps in more detail.

Generating the Interface UUID

Interfaces, like most other objects and entities in DCE, are identified by associating each one with a 128-bit universal unique identifier (UUID). An interface's UUID serves to identify it far and wide throughout DCE. Every interface in a DCE application must have a UUID assigned to it.

When you define a new interface, you must generate a UUID for it. Consequently, the first step in developing an interface definition is to run the **uuidgen** utility to generate a UUID for the interface.

Typically, you run the **uuidgen** command with the **-i** option when generating an interface UUID. The command line has the following syntax:

```
uuidgen -i > your_interface_name.idl
```

where *your_interface_name* is the name you have given your interface, and **.idl** is the suffix that all interface definitions use by convention. The **uuidgen** utility generates a file named *your_interface_name***.idl**, that contains a skeleton of an interface definition and includes the newly generated UUID for the interface. See the *OSF DCE Application Development Guide—Core Components* for more information about the contents of this skeleton file. Refer to the *OSF DCE Application Development Reference* for a complete description of **uuidgen**.

Writing the Interface Definition File

The **.idl** file is where the set of remote operations that constitute the interface are defined. The **.idl** file defines and characterizes the interfaces to the server implementations of the remote operations (which you write, in C source code, then compile and link to the stub code output by the IDL compiler). Thus, an **.idl** file's contents is like a set of *network prototypes* for a set of operations. The IDL definitions in the interface definition file determine not only how the operations "look" to client and server (that is, the operations' call signatures, parameter types, and so on), but also what the data looks like when it is transmitted back and forth between clients and servers in a distributed application.

An interface definition file consists of the following two basic components:

- An interface header

  An interface header contains an interface UUID, interface version numbers, and an interface name. An interface name is an easy-to-read local name that is not guaranteed to be unique; it is merely a convenience. It is helpful if the interface name reflects the nature or purpose of the interface.
- An interface body

  An interface body declares any application-specific data types and constants, and contains directives for including data types and constants from other interfaces. The interface body also contains the operation declaration of each remote procedure to be accessed through the interface. An operation declaration identifies the parameters of a procedure in terms of their data types, access method, and call order, and declares the data type of the return value (if any).

The skeletal interface definition produced by the **uuidgen** utility provides an interface header that contains the newly generated UUID for the interface, a version number, and a dummy string **INTERFACENAME**. Replace this dummy string with the name of your interface, then add any additional interface header attributes your application requires. (See the *OSF DCE Application Development Guide—Core Components* for a complete description of interface header attributes).

The skeletal interface definition file also provides an interface body, which consists solely of **{ }**, that is, an empty pair of braces. You fill in the space between the braces with your RPC interface's import, constant, type, and operation declarations, written in IDL. The *OSF DCE Application Development Guide—Core Components* explains this process in more detail. In addition, the same volume for a complete description of the IDL syntax for specifying import, constant, type, and operation declarations.

Note that a server can implement more than one interface. In this case, you define each interface in a separate **.idl** file and compile it separately with the IDL compiler. You then link the implemented interface operations in various source code files with the IDL output.

Writing the Attribute Configuration File

The attribute configuration file (**.acf**) is an optional additional input file to the IDL compiler, that, if present, affects the IDL compiler's output in various ways. The difference between the purpose of the **.idl** and an **.acf** file is that while the **.idl** file defines how the network communications between the client and server are handled, the **.acf** file, if one is present, affects only the interaction between the stub code modules and the developer code that they support. In other words, changing the contents of an **.acf** file has no effect on the network communications between the client and server.

Nevertheless, some of the features offered by an **.acf** file are very important, and they cannot be obtained by any other means. For example, The **comm_status** attribute configuration file attribute allows the status code of a communications failure that occurs in an RPC to be stored as a parameter or returned as a result, rather than being raised to the caller code as an exception. This attribute can only be declared in an **.acf** file; it cannot be declared in an **.idl** file. Another very important function of the **.acf** file is the specification of a binding method to be used by remote clients of the application. Three methods are available, as follows:

- **auto_handle**
- **implicit_handle**
- **explicit_handle** (the default)

These binding methods are described in "Chapter 4. Binding" on page 91 of this guide. The binding method you choose determines how much attention your server's clients will have to devote to the upkeep of their binding handles.

See the *OSF DCE Application Development Guide—Core Components* for a description of the attribute configuration file attributes available for use in attribute configuration files.

Processing the Files with the IDL Compiler

IDL's input is an *xxx***.idl** and (optionally) an *xxx***.acf** file. Its default output is a header (*xxx***.h**) file, that contains definitions and declarations derived from the input for general use in the development source code, and two stub files, one for the client and one for the server, which contain runtime code for marshalling and unmarshalling, message handling, and all the other details of managing network communications. The stub files are output as object code (*xxx*_**cstub.o** and *xxx*_**sstub.o**) suitable for linking with the developer's compiled code. The IDL compiler generates C source code as an intermediate step in the compilation process, and the output of this step can also be saved in a pair of files (*xxx*_**cstub.c** and *xxx*_**sstub.c**).

In order for a pair of client and server stubs to interoperate, they should be generated from the same interface definition (**.idl**) file, but they do *not* have to be generated with the same attribute configuration file (**.acf**). The compatibility rules for interface version numbers also apply (see the *OSF DCE Application Development Guide—Core Components*).

For further information on the IDL compiler, see the **idl(1rp)** reference page.

Server Initialization

Servers must initialize some data and notify various DCE services about themselves prior to servicing RPC requests. At a minimum, servers must register with DCE and then go into a wait state listening for remote procedure calls. In addition to these minimum tasks, your application may first parse the input arguments, obtain information about how it was started using **dced** API calls, and establish the proper message tables and locale for internationalization.

DCE applications should be started in such a way that they can be controlled by **dced**. When the server is installed, the **dcecp server create** operation (or a custom made server management application) is commonly used to establish the server's configuration with its host **dced**. This configuration data includes among other things the program name and its arguments, the CDS entry name to use for exporting to the name service, and the valid starting methods. Installing your servers in this way does not compromise their security because **dced** operations are protected with ACLs, and the major advantages include the following:

- You do not have write any complex management code for each server
- Your servers are like other DCE servers in that they can all be managed consistently

Depending on how the server is configured, the **dced** can start it in the following ways:

- At boot time when the DCE daemon itself starts
- Explicitly via the **dcecp server start** operation (or from another application that called **dced_server_start( )**)
- Automatically when a remote procedure call comes in for the server
- After a failure of the server it can be restarted

If **dced** did not start the server, it cannot control it. Therefore, one of the first things your server should do is to verify that **dced** started it by obtaining the configuration information, as in the following:

```
server_t  *server_conf;
 .
 .
 .
dce_server_inq_server(&server_conf, &status);
if(status != error_status_ok) {
    .
    .
    .
```

Additional routines, such as **dce_server_inq_uuids( )** and **dce_server_inq_attr( )**, are also useful for obtaining information from **dced** about the running server.

Robust servers usually perform some or all of the following initialization tasks:

- Set up for serviceability which includes establishing message routing, debug levels, and internal message tables.
- Set up the server's objects. This includes creating and storing UUIDs for all necessary objects and object types, and grouping objects by type.

- Set up the security environment which includes setting authentication information, establishing the server's principal identity, and creating ACL managers for each type of ACL object.
- Define manager entry point vectors (EPVs) for each set of interface operations.
- Register the server with DCE. This includes the following: registering the interfaces and the associated EPVs for the operations, establishing the network protocol sequences and endpoints on which the server will listen, registering endpoints and other binding information in the endpoint mapper service, and exporting binding information to the CDS namespace.
- Specify how the server will be multithreaded.
- Listen for incoming requests for remote procedure calls.
- Clean up the program state and environment affected by the server prior to the server's termination.

Setting Up for Serviceability

Serviceability standardizes the server messages displayed or logged. It acts on a set of standard message catalogs and application-specific catalogs generated from the **sams** utility. Some of the obvious advantages the serviceability facility gives servers over using the standard C library routines such as **printf( )** and **fprintf( )** include the following:

- Messages do not need to be hard-coded into applications
- Message routing can be better controlled

The following routine shows how a server can report a status code returned from an API routine:

```
void
print_server_error(
char *caller,          /* Routine that received the error.        */
error_status_t status)  /* Status we want to print the message for. */
{
    dce_error_string_t error_string;
    int print_status;

    dce_error_inq_text(status, error_string, &print_status);
    dce_svc_printf(SERVER_ERROR_MSG, caller, error_string);
}
```

The **dce_error_inq_text( )** routine looks up the status number in a standard table and returns a string of text that describes the error status. The serviceability routine **dce_svc_printf( )** then displays the message, logs it to one or more files, or both.

The following code shows some typical tasks when setting up the server for serviceability:

```
/* The following calls set up default routing of serviceability     */
/*  messages.                                                       */
for (i = 0, route_error = FALSE; (i < MAX_DEFAULT_ROUTES)
&& (!route_error); i++)
{
    printf("Setting default route %s ...\n", default_routes[i]);
    dce_svc_routing(default_routes[i], &status);
    if (status != svc_s_ok)
    {
        print_server_error("dce_svc_routing(default_routes[i])", status);
    }
}
```

```
/* Get serviceability handle...                                   */
smp_svc_handle = dce_svc_register(smp_svc_table,
(idl_char*)"smp", &status);
if (status != error_status_ok)
{
    print_server_error("dce_svc_register()", status);
    exit(1);
}

/* Set the default serviceability debug level and route...        */
dce_svc_debug_routing(default_debug_route, &status);

/* Set up in-memory serviceability message table...               */
dce_msg_define_msg_table(smp__table,
sizeof smp__table / sizeof smp__table[0],
&status);
if (status != error_status_ok)
{
    print_server_error("dce_msg_define_msg_table()", status);
    exit(1);
}

dce_svc_printf(SIGN_ON_MSG);
    .
    .
    .
DCE_SVC_DEBUG((smp_svc_handle,
smp_s_server,
svc_c_debug4,
"Calling dce_server_sec_begin()");
```

Setting Up the Server's Objects

The term object is a very general term that has meaning specific to each application. DCE uses object UUIDs to uniquely identify any object. The creation of object UUIDs, the determination of what (if anything) constitutes an object for a server application, and the association of these objects' UUIDs into collective types are all your application design decisions.

Object UUIDs have a double use in the routing of RPCs, and you may at first find this a bit confusing. One use of object UUIDs is in the DCE RPC binding mechanism so that clients can distinguish between specific resources, and another use of object UUIDs in routing involves grouping objects into types so that a server can support different implementations of the same interface. (DCE servers also use type UUIDs to associate objects for each ACL manager.)

If an application makes use of object UUIDs in bindings, it makes them accessible to clients by exporting them with its bindings when a server registers with DCE.

The following shows sample code to create UUIDs for server objects and how to store them using the backing store API:

```
 .
 .
 .
/* A "well-known" residual name for the management "object":        */
#define MGMT_OBJ_NAME "server_mgmt"
/*                                                                  */
/* A residual name for a sample object:                             */
#define SAMPLE_OBJECT_NAME "sample_object"
 .
```

```
 .
 .
/* These are the backing store database handles:                    */
dce_db_handle_t db_acl, db_object, db_name;
 .
 .
 .
/* A UUID for a sample object:                                       */
uuid_t sample_object_uuid = {/* 00415371-f29a-1d3d-b8c8-0000c0d4de56 */
    0x00415371, 0xf29a, 0x1d3d, 0xb8, 0xc8, 0x00, 0x00, 0xc0, 0xd4,
0xde, 0x56 };
    .
    .
    .
    uuid_create(&server_uuid, &status);
    ..
    ..
    ..
    dce_db_store_by_uuid(db_object, object_uuid, (void *)&sample_data,
status);
    if (*status != error_status_ok)
    {
print_server_error("dce_db_store_by_uuid()", *status);
return;
    }

    /* Finally, store the object UUID keyed by the object */
    /* ("residual") name...                               */

    dce_db_store_by_name(db_name, (char *)object_name, object_uuid,
status);
    .
    .
    .
```

Names are established so that applications can refer to objects in a way other than through the cumbersome UUID. Object UUIDs are generated in the following two ways:

- The **uuidgen -s** command generates the C-structure form of a UUID that can then be hard-coded into applications
- The **uuid_create( )** routine generates a UUID “on-the-fly.”

After creating backing store headers (if desired) and opening the backing store databases, UUIDs are stored by calling the **dce_db_store_by_uuid( )** routine. To store names associated with the UUIDs, call the **dce_db_store_by_name( )** routine.

Object UUIDs in Bindings

Object UUIDs are often used in the DCE RPC binding mechanism. The details of RPC binding are explained in “Registering the Server” on page 17 and more thoroughly in “Chapter 4. Binding” on page 91. It all comes down to this: clients import only *partial* bindings from the namespace. These will carry them only as far as the endpoint mapper service of the **dced** on the destination server’s host; it is **dced**’s job to resolve the binding with a dynamic endpoint.

This means that some registration of bindings must be done by a server with the endpoint mapper. The minimum two items that have to be registered are interface UUIDs and bindings (the latter of which contains the server’s dynamically allocated endpoints). With this information available, the endpoint mapper can inspect the incoming RPCs interface UUIDs, select one of the endpoints that was registered

under them, and resolve the partial bindings. In addition, a server can register its object UUIDs with its endpoint mapper. This allows lookups of endpoints by object UUID rather than interface UUID; the advantage is that object UUIDs are much more specific than interface UUIDs, which may be registered by multiple servers at the same host.

Making Object-UUID/Type-UUID Associations

To group together objects into types, the server makes an RPC library call repeatedly to associate whatever objects it expects will appear in incoming RPCs with a type UUID. The association is made between each of the expected incoming object UUIDs and the type UUID. The following is an example:

```
rpc_object_set_type(obj_uuid, type_uuid, &status);
```

A type UUID is nothing but a special kind of object UUID. *Type* in this context refers to a group of ordinary object UUIDs that have all been associated with another specially generated common object UUID, which can then be used to identify that group of objects collectively.

The type UUIDs in turn are associated with the entry points of manager modules in the server when the server registers with DCE. An incoming RPC with a *typed* object UUID in its binding will be automatically vectored by the server's runtime to the appropriate associated type manager.

Note that it is not necessary to call **rpc_object_set_type( )** at all if you intend to register only one set of manager routine implementations per interface.

Summary of Mechanisms that Rely on Object UUIDs

The type UUIDs and the type manager vectoring mechanism have nothing to do with the use of the object UUIDs themselves as lookups for the host endpoint mapper. The type manager vectoring occurs after object UUID binding happens, at the server. Note also that object UUID binding happens only once in an uninterrupted client/server session; after the partial binding is completed, communications proceed directly between the client and server. Type manager vectoring, on the other hand, occurs every time an incoming RPC contains an object UUID.

The very different nature of the two mechanisms just discussed is somewhat obscured by the order in which they are initialized in the steps in this chapter. The following list shows the relevant server steps, with an indication in each instance to which mechanism they are related:

1. When setting up the server's objects, groups of object UUIDs are associated under type UUIDs in the RPC runtime related to the type vectoring mechanism.
2. When defining the manager EPVs, each type UUID is associated with a manager EPV (in the RPC runtime) related to the type vectoring mechanism.
3. When registering the server, object UUIDs and server endpoints are registered with the server's endpoint mapper and the server bindings (containing the object UUIDs) are exported into the namespace. These are related to the endpoint mapping mechanism.

Setting Up Security

To set up the security environment, the server makes the following DCE library call:

```
dce_server_sec_begin(dce_server_c_login | \
dce_server_c_manage_key, &status);
```

The flags in the first parameter represent the following security issues:

- Establish the server principal identity

  When first invoked, a server process uses the login context of the user who invoked it, until it assumes its own identity by accessing its secret key, which is analogous to a user's password, and using it to get its own login context. Of course, it is possible for a server to simply continue using its inherited login context. In that case, all it needs to do is use the security login routines to obtain its principal name and explicitly get its login context.
- Manage the server key

  When a server has its own identity, it takes on responsibility for the upkeep of its password using the security key management routines.

The decision whether or not to use authenticated remote procedure calls is something of a cooperative matter between the client and the server. When the client calls **rpc_binding_set_auth_info( )** , it registers its preferences about the same things. The client's and server's choices are not required to agree in order for the client to successfully reach the server. If the client's authentication and authorization choices do not agree with what the server expects, it is up to the server to decide whether or not to go ahead with the operations, and how far to cooperate with client requests.

To control access to the server's objects, ACL managers are also set up.

Defining the Manager Entry Point Vectors for Each Set of Operations

Manager is the DCE term for the part of a server that actually implements a set of interface operations (the remote procedures), as distinguished from the more or less generic server initialization code described here.(see **sample_manager.c** for an example of manager code). A manager EPV is the data structure in which is recorded the entry addresses of the application routines that implement the server's operations, as offered through an interface. The server's stub code uses the EPV to dispatch incoming RPCs to the requested operations. For each interface the server supports, a default manager EPV is generated automatically by the IDL compiler. In order for the RPC runtime to properly dispatch remote procedure calls to the correct procedure, the server initialization code must declare the default EPVs and then register them with the runtime, as shown in the following example:

```
extern rdaclif_v1_0_epv_t dce_acl_v1_0_epv;
extern sample_bind_v1_0_epv_t sample_bind_epv;
```

We will later describe registering the EPVs with the RPC runtime.

If more than one version of the same interface is to be supported by the same server, another EPV is needed for each additional interface version. Interface version numbers are specified by the **version** attribute in the **.idl** file. Additional EPVs are also required if the application implements the procedures in more than one way. For example, some applications invoke the same remote procedure to operate on different types of objects. Different objects would likely require different implementations, and thus more than one manager procedure would be coded. The type manager RPC runtime mechanism, properly utilized, allows a server to declare

multiple EPVs under the same interface, and to have the RPC runtime vector (direct) the incoming remote calls to the correct implementation code.

Registering the Server

To register the server with DCE, the server calls the following:

```
dce_server_register(
    dce_server_c_ns_export,   /* flag says register */
    server_conf,              /* server with CDS    */
    &register_data,
    &server_handle,
    &status
);
```

The **dce_server_register( )** routine affects a number of components and services in DCE including the RPC runtime, the local endpoint mapper service, and if the **dce_server_c_ns_export** flag is set, even the CDS namespace. The **server_conf** structure is obtained with a call to the **dce_server_inq_server( )** routine and represents the configuration **dced** used to start the server. This contains information needed to register the server too. The **register_data** structure contains data about the server's interfaces, entry point vectors, and type UUIDs.

The following subsections describe the details about what happens when you register a server.

Registering the Interface, Type UUID, and EPV with RPC Runtime

Earlier we described how to establish an EPV for each set of operations provided by interfaces. Remember that an EPV is a list of pointers to procedures. The first affect of registering the server is to register the services offered (represented by IDL interfaces) and the associated EPVs with the RPC runtime. Registering interfaces with their associated EPVs allow the RPC runtime to use the EPVs to direct an incoming remote procedure call to the correct procedure implemented in the server's manager code.

We also described earlier the type manager mechanism which uses a type UUID to group together object UUIDs. With this mechanism, a different EPV can be associated with each type UUID so that different manager code can be called, depending on an object's type UUID. After these EPVs are registered with the runtime, incoming RPC binding that contain a typed object can be routed by the runtime to the correct manager code.

The data structure the server uses to establish its services is of type **dce_server_register_data_t**. This data structure is initialized prior to the **dce_server_register( )** routine call as in the following example:

```
dce_server_register_data_t  register_data[2];
.
.
.
register_data.ifhandle[0]  = rdaclif_v1_0_s_ifspec;
register_data.epv[0]       = NULL;   /* use the default epv */
register_data[0].num_types = 0;
register_data[0].types     = NULL;
register_data.ifhandle[1]  = sample_bind_v1_0_s_ifspec;
register_data.epv[1]       = NULL;   /* use the default epv */
register_data[1].num_types = 0;
register_data[1].types     = NULL;
```

The **dce_server_register( )** routine usually establishes all the services for a server at once. This is a reasonable approach for most applications, but some interfaces for services may have dependencies on the order in which they are enabled. After the server calls **dce_server_register( )**, it can use a series of calls to **dce_server_disable_service( )** and **dce_server_enable_service( )** to disable and then later reenable any interface offered by the server.

Telling RPC Runtime What Protocol Sequences to Use

The second thing registering the server does is it obtains a set of endpoints and associates them with the desired protocol sequences. Endpoints are the host's address numbers on which the server can receive incoming calls. This begins the process of actually setting up the information that the server's clients will need in order to bind to it.

The endpoints are usually dynamically generated each time the server starts. However, some applications may use well-known endpoints that are the same every time the server starts. If well-known endpoints are used, they are typically defined in the interface definition with the **endpoint** attribute.

In the default case, all valid protocol sequences are used when the **dce_server_register( )** routine is called. The **dce_server_c_no_protseq** flag can be passed in the first argument to the routine in cases where dynamic assignment of endpoints is not desired; for example, when well-known endpoints (specified in the IDL definition) are being used.

Registering the Binding Information with the Endpoint Mapper Service

After server registration obtains the endpoints, the endpoints, protocol sequences, and object UUIDs are registered with the endpoint mapper service of the local host's **dced**.

Typically the server has received a certain number of endpoints dynamically allocated on its host machine. However, when prospective clients import binding information from the namespace, they get partial bindings. When they first try to contact their server, the partial binding will get them only as far as the server's endpoint mapper service. The purpose of registering endpoints is to let the endpoint mapper know what endpoints belong to the server so that it can fill in the partial bindings as they arrive and route the incoming remote calls on their proper ways. Subsequent remote calls executed with the same bindings will go straight to the server, since the bindings are now complete.

The purpose of registering endpoints together with object UUIDs is to account for all possible incoming object UUIDs (that is, object UUIDs that could appear in incoming partial bindings arriving at the endpoint mapper), and to associate with each of them one of the server's allocated endpoints. Then the endpoint mapper can simply look up the object UUID, find an endpoint, insert it into the binding, and send the RPC on to its destination.

An incoming RPC *always* has an interface UUID associated with it; therefore, if a server registers all of its endpoints with the interface it is offering, this will usually be sufficient for the endpoint mapper to send the incoming requests to one of the servers that offer the desired interface, even if there is more than one such server active on the machine. However, if the application is designed in such a way that the binding operation should not be generalized to the interface but must be made

more specific (in other words, this server's clients should always bind *to this server and no other*, even if some other server happens to offer the same interface), then object UUIDs must be used to accomplish this. *Generic* interfaces offered by an application (such as the remote ACL or the DCE serviceability interface) require an object UUID in order to distinguish the application's *instance* of them; unique interfaces, however, do not require an object UUID.

Of course, the server's interface UUID must also be included in each object UUID/endpoint mapping, since no RPC will pass the endpoint mapper if it does not have a matching interface UUID for its destination server. Therefore, the endpoint mapper takes either two or three types of item to be registered, namely

- Endpoints
- Interface UUID
- Object UUIDs (optionally)

It then generates a cross-product table of all possible combinations of all values of the items. This allows it to find a valid endpoint for every possible valid object UUID/interface UUID combination.

The endpoint mapper is the first point of decision for an incoming RPC with a partial binding. The mapper makes its decision *solely* on the basis of the contents of its endpoint map. The object/type and manager EPV registrations that were done earlier have no effect on the endpoint mapper. Only after a client request arrives at the server does the server's runtime routines dispatch the request among multiple managers, if type managers have been registered by the server. The endpoint mapper knows nothing about registered object types.

Exporting the Binding Information to the Namespace

The final task of server registration (if the **dce_server_c_ns_export** flag is set in the **dce_server_register( )** call) is to export the binding information to the namespace. In the usual case, where the server's endpoints have been dynamically allocated to it, the endpoint information will not be included in the exported handles. Instead, this information will be filled in by the host's endpoint mapper as the partially bound handles arrive at the host in incoming RPCs. However, if the endpoints are well-known, they will be included in the exported binding handles, and clients will thus import fully bound handles.

If you wish, you can use the lower level RPC routine **rpc_ns_binding_export( )** to export individual services to the namespace, but in this case you should first be sure the flag **dce_server_c_ns_export** is not set in the **dce_server_register( )** routine.

As a final note, a client must have a binding handle in order to reach a server, but it does not have to get the handle from the name service. However, the name service is the recommended way for clients and servers to find each other because it is a convenient and easy to use service built into DCE.

Specifying Multithreadedness

The application may also spawn an additional thread for a signal handler. An example follows:

```
if (pthread_create(&sigcatcher,
        pthread_attr_default,
```

```
        (pthread_startroutine_t)signal_handler,
        (void*)0))
{
    dce_svc_printf(NO_SIGNAL_CATCHER_MSG);
    exit(1);
}
```

The *max_calls_exec* parameter to the **rpc_server_listen( )** routine specifies the number of operations that the server can perform concurrently in response to client requests. The *max_calls_exec* parameter is also used to derive the size of a buffer (the call request buffer) for incoming client requests that cannot be immediately executed. *max_calls_exec* specifies the upper limit for the number of RPC threads that will be spawned by the RPC runtime to handle incoming remote procedure calls. Thus, an important side effect of **rpc_server_listen( )**, when the specified concurrency is greater than 1, is to create multiple threads of execution in the server.

The threads are automatically spawned to handle whatever operation is requested by the client. If the maximum number of manager threads is already active and more incoming calls arrive, the RPC runtime buffers them in a call request buffer. The size of the call request buffer depends on the *max_calls_exec* parameter; the larger the parameter, the bigger the buffer. Incoming calls beyond the call request buffer capacity are rejected (with an error code) by the RPC runtime.

Although the execution threads are automatically managed by the RPC runtime, the developer is responsible for coding the manager routines according to thread-safe guidelines so that the threads will execute properly. For further information on thread-safe programming practices, see “Chapter 2. Threads” on page 35.

Listening for Incoming Service Requests

In order to begin listening for incoming remote procedure calls, the server calls the following RPC library routine:

```
rpc_server_listen(max_calls_exec, &status);
```

The *max_calls_exec* parameter specifies the number of concurrent remote procedure calls the server can execute. This call normally begins a ”semi-infinite” loop, execution of which is terminated only by one of the following events:

- One of the server’s manager routines calls **rpc_mgmt_stop_server_listening( )**
- One of the server’s clients makes a remote call using the routine **rpc_mgmt_stop_server_listening( )**. (Note that the server can intercept such a remote call and either allow or prevent it by installing a function with **rpc_mgmt_set_authorization_fn( )**).
- A management application makes a remote procedure call using the routine **dced_server_stop( )**
- An administrator (or administrative script) uses the **dcecp server stop** *server_name* operation
- A signal or exception occurs

From the point of view of the server, the call to **rpc_server_listen( )** blocks until the **rpc_mgmt_stop_server_listening( )** routine is called. When this happens, the RPC runtime stops accepting incoming client requests to the server, and when all the currently executing operations are completed, the call to **rpc_server_listen( )** returns.

Server operations can also be terminated by an exception or signal. DCE Threads defines all exceptions as *terminating*, which means that execution must be caught by an exception handler (if one exists) and then be resumed there, or the process will be terminated. Certain signals are defined by DCE Threads as exceptions, which means that these signals have the same general characteristics as exceptions. For more information on the DCE Threads exception handling interface, see "Chapter 2. Threads" on page 35.

Cleaning Up Code When the Server Terminates

If (or when) the server terminates execution, it should undo its initialization that affected other facilities and services of DCE. Facilities affected include the CDS namespace, the endpoint mapper service, and backing store databases such as those used for ACL managers. For the most part, API routines that cause these kinds of effects have a corresponding API routine to undo them. The following sections describe the series of routines typically used to clean up after an application.

Unregistering the Server

Two important aspects of registering the server is that it registered the interfaces and EPVs with the RPC runtime, and it established the endpoints (or addresses) on which the server listened for requests. If the endpoint map contains *stale* data, it can create for a client a fully bound binding that is not valid. Even though the endpoint mapper service does its own housecleaning periodically, there is the possibility that these invalid bindings could be created and used. Therefore, it is a good idea to call the following routine:

```
dce_server_unregister(server_handle, &status);
```

In addition to unregistering the server's address information from the local endpoint mapper's database, this routine unregisters all the services (interfaces and EPVs) from the RPC runtime as well.

If your application requires a partial shutdown or a particular order to the shutdown of services, you can use more specific routines such as **rpc_ep_unregister( )** and **dce_server_disable_service( )**.

Unexporting from the Namespace

If the server is going to be out of service for an extended period, it should unexport any information it previously caused to be placed in the namespace. This will prevent future prospective clients from being misled into attempting to reach the server when it does not exist, and also will help to conserve resources in the namespace.

Unexporting is automatic when **dce_server_unregister( )** is called if the **dce_server_c_ns_export** flag was set when the corresponding **dce_server_register( )** was called. For more specific control, an individual service previously exported is removed from the namespace with the following routine:

```
rpc_ns_binding_unexport(entry_name_syntax, \
entry_name, if_handle, obj_uuid_vector, &status);
```

The CDS namespace is designed to store location data for extended periods of time.

Cleaning Up Security Information

A call to the **dce_server_sec_begin( )** routine should have a corresponding call to the **dce_server_sec_done( )** routine to release resources allocated. In addition, your code should close any backing store databases used for ACL management.

The Client Binding and RPC Invocation

To use RPC, a client must first establish a binding to the server. The following steps cover bindings and binding handles.

The programmer designing clients must decide whether or not to use threads, and should have an understanding of multithreaded clients. DCE provides a set of tools for multithreaded programming; these are described in “Chapter 1. Introduction to DCE Application Programming” on page 1.

Importing the Binding Information from the Namespace

The first important thing that the client does is to acquire a binding to the server it wants to request services from. From the client’s point of view, there are several binding choices to be made.

The first choice is in regard to the binding method to be used; however, this is determined and implemented as part of the development coding process (the **.acf** file). The binding method chosen has an effect both on what the client has to do in the present step to acquire bindings, and subsequently on what it must do to maintain them. In this step, it will be assumed that either the explicit or implicit method was chosen. If auto-binding were chosen, there would be no need for a discussion, since the client would then have nothing to do.

Getting a Handle

The second choice involves how to get a binding handle. Again, this is a choice that is at least partially dependent on decisions already made. The client can always generate a binding handle for itself; the problem is where to get the information that belongs in it. There are two general solutions, as follows:

- The client imports from the namespace binding handles that already contain the necessary information, or
- The client receives the information in string form from user input, from a file, from another server, or from any other source. It then converts the string into a binding by calling **rpc_binding_from_string_binding( )**.

The normal way for a server to make its location known to clients is to export its binding information into the namespace. The client can then call the following RPC name service library routines to import one or more bindings from the specified namespace entry:

```
rpc_ns_binding_import_begin(entry_name_syntax, entry_name, \
if_handle, obj_uuid, &import_context, &status);

rpc_ns_binding_import_next(import_context, &binding_handle, \
&status);

rpc_ns_binding_import_done(import_context, &status);
```

The name service sees to it that only compatible bindings exported under the specified interface, with the optionally specified object UUID, will be returned to the client. (Note that the interface specification is *not* contained in the binding, although it is exported to the namespace entry where it is used by the name service for matching entries to prospective importers.) The object UUID specified by *obj_uuid* is contained in the binding, if it is present. This is the object UUID that was (optionally) registered under a type UUID in an earlier step. Even if *obj_uuid* is not specified in the import call, it will be returned in the binding handle(s) if it was exported by the server.

Determing the Entry Name

To determine how the client knows the entry name to import from, the simplest method is to have the user type it in on the command line.

Binding Compatibility

The protocol sequence used must be supported by both the RPC runtime and the operating system on the client's machine. However, the RPC runtime implicitly takes care of binding compatibility when it returns bindings to importing clients; only compatible bindings are returned.

The **rpc_network_inq_protseqs( )** and **rpc_network_is_protseq_valid( )** routines can be used to return all supported protocol sequences and to determine whether a specified protocol is supported, respectively.

To find out what protocol sequence is used in a binding handle, make the following series of calls:

```
rpc_binding_to_string_binding(binding, &string_binding, &status);

rpc_string_binding_parse(string_binding, NULL, &protseq, \
NULL, NULL, NULL, &status);
```

Annotating the Binding Handle for Security

Now that the client has a binding, it is almost ready to begin RPC operations. One last preliminary task remains; namely, to specify various security-related parameters to the RPC runtime, which will govern the (security) conduct of the ensuing client/server relationship. If the client does not require authentication, it can skip this step completely. The result will be that no authentication will take place between the client and server. It will then be up to the server to decide how far to go with an unauthenticated client.

Preparation

What the client essentially wants to do now is call the routine **rpc_binding_set_auth_info( )** in order to specify all the necessary security parameters. However, when it does this, it should be able to specify its server's principal name, so that the server it binds to can be authenticated *to the client*. (The server's principal name is the name by which the server is known to the DCE Security Service.) The client must also supply a handle to its own login context when it calls **rpc_binding_set_auth_info( )**.

There are several ways to determine the server's principal name, as follows:

- The server's principal name could be hardcoded in the client. This is not recommended practice for reasons of robustness and flexibility.

- The client can be handed the name as input from the command line when it is invoked.
- The principal name can be the same as the name entry (binding information) name.
- The client can query the server's principal name by calling **rpc_mgmt_inq_princ_name( )**. It can then check group membership by calling **sec_rgy_pgo_is_member( )**, using a known tested group.

The reason for checking group membership has to do with authorization-related decisions that the client may need to consider. It is not necessarily enough to know that a server has a certain identity; it may also be necessary that it belong to a certain group in order for it to be fully authorized, from the client's point of view, to receive the data that the client will send. In other words, the client may need to make a decision about the server similar in nature to that which the server makes about the client, when it checks the client's authorization, via ACLs, to do the things it wants to do. Security can be just as important for the client as for the server; this is the justification for having to make the extra calls described here.

The client retrieves its login context with the following security library routine:

```
sec_login_get_current_context(&login_context, &status);
```

However, this is not usually necessary. The client can, by passing a NULL value to **rpc_binding_set_auth_info( )**, simply use its default login context.

In any case, note that this login context already exists; the client merely retrieves it. (The client inherited its login context from the user principal who executed it.) The client can now set up for authenticated RPC.

Setting Up for Authenticated RPC

The client makes the following call in order to set up the security characteristics of the communications it is about to enter into with the server:

```
rpc_binding_set_auth_info(binding, \
server_princ_name, protect_level, authn_svc, login_context, \
authz_svc, &status);
```

The security parameters specified here include *protect_level* for level of protection performed (for example, authenticate only at the beginning of each RPC, or authenticate everything received by the server), *authn_svc* for the authentication service (including "none"), and *authz_svc* for the type of client authorization information that will be supplied to the server.

The usual practice is to pass NULL for *login_context* here, and thus use the default context.

Note that it is the client who chooses whether or not to use authenticated RPC, as well as the level of authentication, and how much authorization information about itself to send. It is then up to the server to accept this arrangement or reject it, or to allow some limited operation with the client, or whatever else it might decide. The server decides which authentication to use. The client also specifies an authentication service (in *authn_svc*), but if this differs from what the server specified, the call to **rpc_binding_set_auth_info( )** will fail and an error will be returned to the client.

There is an important difference between the rationales of authentication and authorization. Authentication is performed by the RPC runtime and is only indirectly felt by client and server; authorization, however, is for the most part implemented explicitly in the server code if it is implemented at all. This difference is the reason for the larger number of authentication-related arguments that have to be specified in this step.

For further information about authenticated RPC, see the *OSF DCE Application Development Guide—Core Components*.

Invoking Remote Procedure Calls

This step is the culmination of all the foregoing steps; here the client makes its first remote call to the server. This call, which will obviously be application specific (its definition was specified in the application's **.idl** file, and possibly modified by the **.acf** file), will look something like the following:

```
my_rpc_op(binding_handle, arg1, arg2, arg3);
```

Note that the presence of the binding handle as a parameter means that explicit binding handles are being used.

Note also that after all the preceding talk about interfaces, no interface handle appears in the parameter list. The RPC runtime takes care internally of making sure that the interface offered by the server exactly matches what the client expects. The **my_rpc_op( )** routine was (or should have been) defined as part of the application's interface. When the client calls **my_rpc_op( )** in the present step, the client stub code (which was generated during the IDL compilation step) will include the correct UUID for the interface the routine is associated with in the data sent out on the network. The RPC runtime uses the interface specification included with each RPC as a "fingerprint" to ensure that the operation being requested of a server is in fact implemented by that server. This ensures that interface compatibility is never dependent on the vagaries of application code.

The Possibility of Binding Failure

Perhaps the most important thing to mention about this step is that it may not at first succeed. Remember that the client imported a *partial* binding to the server. Completion of the binding, and therefore of the remote call, depends on the endpoint mapper's being able to successfully complete the incoming binding with a good endpoint for either the specified server (if one is specified) or for one of its own choosing. This in turn depends on the up-to-dateness of the host's endpoint database, and that depends on such things as other servers' being conscientious about unregistering themselves when terminating, and so on. Even the target host specified may not be valid when the call is made because of any one of the various network problems that can occur.

In other words, the client should regard an unused binding not as a firm promise that comes directly from the server, but rather as a well-meant expression of intent passed on by the name service and based on circumstances not entirely under anyone's control. This is the reason for the series of binding import calls described earlier. The prudent thing for a client to do after importing a binding is, therefore, to assume that it will have to perform one or more times a series of steps something like the contents of the following loop:

1. Annotate the binding handle for security.

2. Try it out: attempt a remote call with it.
3. If the call succeeds, discard the binding import context and proceed to step 5 in this loop.
4. Otherwise, if the call fails, import the next binding and return to step 1 in this loop.
5. Proceed with remote operations until finished.

If all imported bindings happen to fail, this could be because the client's cache of bindings has become stale. The client could then try calling **rpc_ns_mgmt_handle_set_exp_age( )** with a low timeout value, and then retry the previous loop. A last resort could be to allow the user to type in a string binding.

Note that if you are using the auto-binding method and the binding becomes unusable for some reason, the RPC runtime will rebind under most conditions.

The Result of Successful Binding

If **my_rpc_op( )** or its equivalent does succeed, the binding will as a result be complete (even if it was partial before), and the information in it can be regarded with much more assurance from then on. Subsequent remote procedure calls by the client to the same server will go straight to the bound-to server.

The Server's Manager of RPC Requests

As was explained, server threads are automatically spawned by the RPC runtime in the server manager to handle incoming remote procedure calls from clients. The number of calls that can be concurrently handled depends on the value of the *max_calls_exec* parameter specified in the call to **rpc_server_listen( )**. The thread is created by the RPC runtime and begins execution in the operation requested. When the operation is completed, the thread is automatically terminated (by the RPC runtime).

See also the *OSF DCE Application Development Guide—Core Components* and the *OSF DCE Application Development Reference* for a comprehensive discussion of DCE threads.

Getting the Client's Credentials

As mentioned in the previous step, authentication, if it was specified by the client, has already occurred if the client's request is received by the server manager. If the client fails to authenticate itself to the server runtime, its remote procedure call fails before reaching the server's RPC code.

Authentication, if specified by the client and offered by the server, is performed by the RPC runtime; it is not a responsibility of the application code. However, it is up to the application to formulate its own security policy with regard to the client, based on the following:

- The level at which the client has been authenticated.
- The client's authorization; that is, whether the client should be allowed to access resources it may request.

In order to find out the client's authentication and authorization information, the server calls the following RPC library routine:

```
rpc_binding_inq_auth_caller(binding_handle,
privs, server_princ_name, \
    protect_level, authn_svc, authz_svc, &status);
```

The parameters in this call are analogous to the similarly named parameters in the registration routines. The server can learn what level of authentication, what authentication service, and what server principal name the client specified. Of most interest, however, are the *privs* and *authz_svc* parameters. The *privs* parameter is a pointer to whatever information the client is willing to let the server know about its privilege attributes; *authz_svc* tells what this information is. It could be any one of the following:

- The client's privilege attribute certificate (PAC), containing the client's principal and group UUIDs. These can be used to look up the client's privilege attributes in ACLs, whose entries are keyed by principal and group UUID.
- The client's principal name (a string). This also can be used to look through ACLs, provided that the lists have been annotated with such name strings.
- Nothing. The client chooses not to provide any authorization information.

From now on, it is the server's decision, as implemented by the developer, how to respond to the client's requests for services and resources, depending on the security information the server has learned about it. A non-ACL-based strategy may be implemented using the client's principal name string for lookups. The ACL-based strategy, which is supported by a DCE interface, is described further in the next step.

Getting the Object's ACL

This step is reached if the client requests access to any object, resource, or service that is managed by the server, to which ACLs are attached. As previously mentioned, the application must implement its own ACL manager if it wants to use ACLs to control access to its resources. For further details on how to go about creating an ACL manager, see "ACL Managers" on page 69.

In order to allow applications to as easily as possible offer an ACL interface that is uniform with that used by the DCE components themselves, the remote ACL interface has been built into the DCE library, and client applications can perform operations on ACLs through another interface, also part of the DCE library, which calls through the remote interface to the appropriate manager. The remote interface, consisting of **rdacl_ *( )** calls, must be implemented by the server application; clients execute the local **sec_acl_ *( )** routines, which are linked to every DCE application as part of **libdce**.

For the client, all that is necessary is to possess a binding to the object whose ACL is to be operated on. As long as the application exposes the resources it manages as accessible objects (via the namespace), then the DCE ACL interface provides for a client's being able to bind to the object by calling **sec_acl_bind( )**. (In fact, this kind of object-oriented binding model can be very useful, and is discussed in further detail in "Chapter 4. Binding" on page 91.) Note that the **sec_acl_ * ( )** routines use an ACL handle to specify the object whose ACL is to be accessed, so **sec_acl_bind( )** must always be called to obtain this handle, even if the client is already bound to the object's server.

There is a user interface into the ACL operations, embodied in the **acl_edit** command. For further information, see the *OSF DCE Administration Commands Reference*.

Server applications can use the DCE ACL library routines to implement ACL managers. The DCE ACL library is an implementation of the remote ACL (**rdacl**) interface, designed in such a way as to allow any DCE application to use it instead of having to implement the interface itself. In DCE 1.0, applications that wished to use the DCE ACL functionality had to implement the full remote interface themselves; in DCE 1.1 this is no longer necessary. For further information, see “Chapter 3. Security” on page 51.

Making the Authorization Decision

In this step, the server’s ACL manager inspects the ACL of the resource (object) under question, determines whether the client is authorized for the requested access, and takes the appropriate action.

The application may choose to implement more than one type of ACL (reflecting the different kinds of objects and resources to be protected), thus resulting in several ACL *type managers*.

Although it is up to the application to implement its own ACL storage, testing algorithms and manager types, there are certain DCE-wide design conventions that should be kept in mind and departed from only for good reason. Among these are the following:

- Standard DCE ACL entry types: the kinds of entry that can occur in an ACL (for example, **user**, *group*, and so on).
- Standard privileges: the kinds of access that a principal can have to a protected object (for example, read, write, and so on).
- Standard inheritance rules: these rules govern the default characteristics of ACLs created for newly created objects.
- Standard access algorithm: the order in which a client’s credentials are matched against the various possible entry types.

Information about these topics for application developers designing their own ACL model can be found in the *OSF DCE Application Development Guide—Core Components*, in which all the DCE authorization conventions are described in detail.

Servicing the RPC Request

If the client’s request is determined to be properly authorized, then the requested operation can proceed.

Note that this step and steps D3 and D4 (as discussed in “Overview of DCE Application Development Steps” on page 3) are somewhat intertwined. Something like the following could occur:

1. The server wakes up in some routine defined in its manager code. For example, if the client executed the call **my_rpc_op( )**, then the server will wake up in the routine that implements this remote call.

2. Execution of the **my_rpc_op( )** routine requires the **insert** privilege for the application's database **my_database**. So **my_rpc_op( )** begins by checking the client's relevant privilege attribute by making an internal call to the application's ACL manager.
3. If the client is found to have the requisite privilege, **my_rpc_op( )** proceeds.

The remote procedure executed in this step is written by the application developer.

Returning the Results and Resuming Listening

At the completion of the operation, the RPC thread that was automatically spawned to execute it is terminated by the RPC runtime. As far as the server is concerned, it is still blocking on the call to **rpc_server_listen( )** which was made earlier. If *max_calls_exec* was specified to be greater than 1 in that call, other threads may still be executing at this time in response to other requests that have been received from other clients. In any case, the call to **rpc_server_listen( )** will not return until one of the server's own management routines, or a client, makes a successful call to **rpc_mgmt_stop_server_listening( )**. If this happens, the RPC runtime will stop accepting incoming client requests to the server. When all the currently executing operations have been completed, the call to **rpc_server_listen( )** will return.

The other way that execution can be thrown out of the **rpc_server_listen( )** call is as a result of a signal or exception.

From the server's point of view, the result of completing the remotely called routine is that it reenters the *listen* loop, waiting for further remote calls. The server's runtime handles all the communications details of actually sending any requested data to the client.

From the client's point of view, the server's return at the end of its remotely called routine results in the client's returning from a seemingly locally executed routine.

Continuing

The client now goes on about its business, which may include performing other remote procedure calls.

Note that there is no housekeeping burden placed on the client with regard to the termination of the relationship with a server. However, a long-lived client might want to make use of the **rpc_binding_free( )** routine to free memory that was allocated for no-longer-used handles. The client should also call **rpc_ns_binding_import_done( )** to clean up the resources used by the NSI routines. If another binding handle will be needed later on, then **rpc_ns_binding_import_begin( )** will be recalled.

About DCE Programming Style

The *OSF DCE Application Development Guide—Introduction and Style Guide* (hereafter, the *Style Guide*) attempts to bridge a gap. On one side stands the tutorial and reference material provided by the rest of the *OSF DCE Application Development Guide* and by the *OSF DCE Application Development Reference*. In theory, this material provides complete documentation of the *mechanisms* of DCE application programming. In particular, it documents the syntax and semantics of every DCE API interface and IDL construct and provides a service-by-service guide to their use.

On the other side stands the formal application portability specification provided by the *AES/DC*. This provides a *policy* guide of a specific kind: if applications wish to be portable among DCE implementations, they need to follow the AES guidelines.

Between these two poles of DCE documentation, there is still a great deal of room to maneuver. The DCE application programming facilities provide such a large number of mechanisms, so many possible ways of doing things, that it is often difficult for the programmer to decide among them. The guidelines provided by the *AES/DC* are limited to only one (albeit an important one) policy issue: portability. The DCE programmer is still left with many decisions about issues that do not arise in the typical local programming environment: how to use the name services, which security services to employ, how many threads to use, and so on.

The *Style Guide* attempts to answer many of these questions or at least to provide the grounds upon which an application programmer can base decisions. Of course, the coverage in these relatively few pages in not exhaustive. The number of implementation issues raised by the available DCE application programming mechanisms is potentially unlimited. The *Style Guide* attempts to cover the major issues that are likely to confront most programmers at some stage in DCE application design and development.

Aside from attempting to anticipate your questions, the *Style Guide* may also raise issues that you may not even have considered. DCE covers a great deal of ground that is probably unfamiliar to most application developers, such as multithreading and distributed security. When moving in such unfamiliar territory, it is easy to overlook potential problems. The *Style Guide* attempts to alert you to major stumbling blocks in each area.

Mechanism, Policy, and Style

The *Style Guide* is based on what is, to some degree, a fiction: that application development issues can be nicely divided between *mechanism* on one hand and *policy and style* on the other. In theory, the *mechanisms* of DCE programming refer to the syntax and semantics required by APIs, IDL constructs, services, and the like. These are the things about which the programmer has no choice: they must either be done according to the documentation or not done at all. *Policy* and *style*, on the other hand, are supposed to refer to the things about which the programmer can make a choice: specifically, which mechanisms to use in given circumstances.

In practice, the distinction between mechanism and policy/style is often vague. The other parts of the DCE application development documentation set contain much that could be considered policy and style guidance. And, for reasons discussed in some detail in the next section, the *Style Guide* often contains descriptions of the mechanisms of DCE programming.

Nevertheless, the *Style Guide* does attempt to keep to the ground of policy and style issues. It assumes that you already know what mechanisms are available and attempts to provide guidance about the choices you have in using those mechanisms. One result is that the *Style Guide* is not a tutorial; it often assumes knowlege of terms and concepts that are explained elsewhere in the programmer's documentation.

On the other hand, the *Style Guide* does in many cases provide high-level discussions of the organization and principals of DCE services, such as the security

services. The assumption is that you may already know many of the details but may lack an overall framework. Often, such a general model is just what you need to be able to make rational policy decisions.

The distinction between policy and style is itself somewhat vague. In general, *policy* refers to the things you *should* do in an application program. You can usually identify a policy recommendation because the words "should," "must" or "recommended" appear. Style is a more general term that includes policy (hence "*Style Guide*"), but that also covers a variety of other suggestions about how you might do things. Much of the sample code included in the *Style Guide* embodies not only the recommended policies, but also provides illustrations of possible styles of usage. Such suggestions are intended to be helpful, but unless they are couched in the language of policy, should be considered entirely optional.

Policy and Style Issues

Remote application programming, using DCE, imposes some special requirements on applications that are not relevant to most local applications. A DCE application is a multicomponent system in which the various components interact dynamically as the program operates. Obviously, the application developer is concerned with creating two major types of components, servers and clients, but these application specific components also enter into relationships with other DCE components. For example, most applications will be clients of naming and security services. Server applications that provide ACL managers may act, in turn, as servers to **dcecp** ACL commands. Many similar client/server relationships may be created during the operation of a distributed application.

Furthermore, even components that do not communicate directly share common resources, such as directory and security services. Components use these services to exchange specific kinds of data, such as bindings, and such exchanges can succeed only when they are made according to the correct protocols. For example, a server needs to organize the way it exports bindings to a name service so that clients can succeed in finding them. Similarly, clients and servers can only succeed at authenticated communications if the correct registry and ACL data has been created and if each follows the correct incantations to make use of this data.

A particular constraint on DCE applications is that they must take into account the administrative overhead of a distributed system. Servers need to consider such issues as the location and availability of the services they need, the structure of the namespace into which they export their bindings, the DCE identity and privileges under which the server must run, and many similar issues. A successful server will be one that interacts correctly with other components while imposing a minimal load on the DCE environment and, most important, can be successfully and easily administered.

To meet these requirements, application components must interact with each other and with other DCE components in a consistent and well-behaved manner. In this context, one can think of DCE applications as having to meet application-level and administrative interoperability requirements. The *Style Guide* is, in part, a guide to such requirements. Given the enormous variety of programming and administrative mechanisms that DCE makes available to the programmer, the *Style Guide* provides a set of policy recommendations for the use of those mechanisms that will maximize the application-level and administrative interoperability of DCE applications.

In addition to being complex, DCE application programming involves elements that are likely to be unfamiliar to many programmers, such as remote parameter passing, name services, and distributed security services. Another goal of the *Style Guide* is to suggest wise uses for these tools, since many of the familiar local programming models are inadequate. These recommended policies are especially important in the area of security: an application that fails to follow them is likely to be insecure. Recommended policies in some other areas, such as execution semantics and locking, may also fundamentally affect the integrity of a distributed application and should not be lightly ignored. Other policies, such as those relating to parameter passing affect mainly appliction performance.

The simple unfamiliarity of many of the concepts can make the actual coding of an RPC application a daunting task. In traditional C programming you can usually begin with familiar models—often, with existing code—but with RPC you are unlikely to have such starting points. Therefore, this guide also provides extensive examples that illustrate the basic uses of many important elements. For example, in developing an ACL manager, you may well be able to use the sample ACL manager as a starting point.

The sample code is intended to suggest certain styles of usage that will probably prove useful in many situations. Obviously, these styles are only sugggestions: you will certainly develop your own DCE programming style as you develop DCE applications.

General Policies

The *Style Guide* embodies a variety of basic assumptions. These form the basis for a set of high-level policy recommendations that cross the boundaries of the specific services discussed in later chapters. These are as follows:

- Servers are generalized providers of the services specified by their published (IDL) interfaces. That is, servers should encapsulate the services they provide in such a way that naive clients, with no knowledge of the specifics of server implementation, can successfully make use of these services via the remote interfaces. In this sense, servers are much like libraries. One should not assume that clients will be written by someone with knowledge of server internals. Where appropriate, define wrapper routines for the IDL operations to shield developers from binding handles and other RPC peculiarities.
- Servers should make their resources known to clients using standard mechanisms. In particular, they should export their bindings according to the recommended service models, use name and endpoint services rather than fixed bindings and well-known endpoints, and associate exported objects with UUIDs.
- Clients and servers should be portable, using DCE provided mechanisms instead of operating system and transport-dependent mechanisms. For example, data streams should be communicated via the RPC pipe mechanism rather than socket calls. The AES/DC is the definitive guide to application portability using the DCE mechanisms.
- Distributed applications make greater administrative demands than nondistributed ones. Clients and servers need to be written with an eye to minimizing and simplifying administrative tasks. This means, for example, that
  - Applications need to be as configuration and location independent as possible. In particular, this means giving careful thought to the use of name services for advertizing and finding resources.

  - Applications require both local and DCE identities and privileges. They should follow the recommended models for acquiring and maintaining these privileges and identities.
  - Servers should be administratively interoperable; that is, they should behave like the standard DCE servers, exporting the recommended management interfaces, exporting ACL managers, logging errors and messages, and providing for the standard startup and shutdown mechanisms.
- Distributed security is inherently more complex than local system security (you can't just "lock the door"). Applications should follow the recommended security policies rigorously.
- Clients and servers should follow the recommended internationalization guidelines to ensure character set interoperability.

Chapter 2. Threads

Threads as used specifically in DCE applications raise several obvious policy issues which may be summarized, roughly, as follows:

- When to use multiple threads
- How many threads to use
- What scheduling and priority attributes to apply

These issues are covered in “Thread Use Policy” on page 36.

Beyond these obvious policy questions, however, threads raise a tricky issue for a programming policy guide because it is not always clear where the line between mechanism and policy lies. Multithreaded programming in general requires a number of practices that are likely to be unfamiliar and unintuitive to many programmers, and errors arising from failure to follow these practices can be obscure, infrequent, and difficult to reproduce. One result is that an incorrect program can easily appear to be correct.

A typical case is a program that performs the following sequence of steps:

```
pthread_create(&thread . . .);
pthread_setprio(thread . . .);
```

From the point of view of a single thread, this may seem like a logical sequence of steps, yet it contains a fundamental error: the spawned thread may well have begun to execute, or even have terminated, by the time the call to **pthread_setprio( )** occurs. The result is a program whose behavior is indeterminate, and which may fail unpredictably. The correct procedure is to use a thread attributes object to set the thread’s priority when it is created.

Strictly speaking, this is really a programming *mechanism* issue, since the failure to follow the rule results in an incorrect program. However, errors of this type can be obscure: in fact, the resulting program might never fail due to this error. There are many such error possibilities in a multithreaded program that can result in all kinds of deadlocks, race conditions, and data corruption. Yet these errors can sometimes be so obscure as to be extremely difficult to analyze *a priori*, and failures may occur so rarely as to be virtually unreproducible.

As a result, correct use of threads mechanisms requires following a set of general rules designed to avoid errors that may or may not occur in specific cases. For example, locks must be taken and released in the same strict order. Rules like this are in not enforced by the thread programming mechanisms, and failure to follow them will not always result in program failures. In fact, failure to obey these rules may not always be a programming error: depending on the program, it is certainly possible that there is no possible execution path where failure to follow a rule would result in an error (although this might be difficult to establish *a priori*).

As a result, such rules have in some sense the flavor of policy recommendations: they are a set of disciplines for avoiding certain classes of problems which threads programmers can assume to exist, in general, even though they might not arise in specific cases. Because of this, and because these rules may be unfamiliar to many programmers, it seems wise to repeat them in summary form in this policy guide. Moreover, because DCE client and server applications are implicitly multithreaded,

even when the application itself makes no thread related calls, it is also important to identify when application code must be thread safe. These issues are covered in "Thread Safety" on page 38.

The remaining sections of this chapter cover a variety of specific policy and usage issues relating to DCE threads. Thread handles and thread-private data are discussed in "Thread Handles" on page 41 and "Storage for Thread Specific Data" on page 41.

Cancels and signals introduce a number of specific semantic issues that applications must be aware of when programming in a multithreaded environment. These are covered in "Canceling Threads" on page 42 and "Signals" on page 46 respectively. Finally, DCE introduces the concept of an RPC thread. This is intended to extend the semantics of a local thread of execution across two address spaces in the course of an RPC. However, the extension is not entirely transparent, and applications need to be aware of the semantic peculiarities of RPC threads. These are covered in "RPC Threads and RPC Cancel Semantics" on page 49.

Thread Use Policy

Thread use policy questions arise in the following two ways:

- Server manager code is multithreaded by default, and applications can specify the degree of multithreading.
- Client code can be made multithreaded by making threads API calls.

Choosing to Thread

The choice of multithreading is really a question of specific application design, and only general guidelines can be supplied here. Application programmers need to be aware that, depending on the threads implementation and the underlying hardware, concurrency may be more apparent than real for many applications. If threads are being time-sliced on a single processor, nonblocking activites will not go any faster because they are multithreaded. In fact, given the extra overhead of a given threads implementation, they may be slower. Even on a multiprocessor, with the DCE user-space threads implementation, all threads in a single process contend for the same processor.

On the other hand, if multiple threads are carrying out activities that may block—and this includes making RPCs to remote hosts—then multithreading will probably be beneficial. For example, multiple concurrent RPCs to several hosts may allow a local client to achieve true parallelism. Note however, that concurrent RPCs to a single server instance may not be any more efficient if the server itself cannot get any real benefit from multithreading of the manager code.

RPC servers are multithreaded by default, since multithreading is an obvious way for servers to simultaneously handle multiple calls. Even if the manager code and underlying implementation do not permit true parallelism, manager multithreading may at least allow a fairer distribution of processing time among competing clients. For example, a client that makes a call that can complete in a short time may not have to wait for a client that is using a lot of processor time to complete. For this to occur, threads must make use of one of the time-sliced scheduling policies (including the default policy). On the other hand, if all calls make use of

approximately similar resources, then multithreading may become simply an additional, possibly expensive, form of queueing unless the application or the environment permits real parallelism.

In summary, the developer must consider the following questions in order to decide whether an application will benefit from multithreading:

- Are the threaded operations likely to block, for example, because they make blocking I/O calls or RPCs? If so, then multithreading is likely to be beneficial in any implementation or hardware environment.
- Can the underlying hardware and RPC implementation support threads on more than one processor within a single process? If not, then multithreading cannot achieve real parallelism for processor intensive operations. The DCE user-space threads implementation restricts all threads of a single process to contend for a single processor and so cannot provide real parallelism for processor intensive operations.
- Even if the answer to both of the first two questions is yes, will the use of a time-slicing thread scheduling policy permit fairer distribution of server resources among contending clients? If so, then server manager multithreading may be beneficial.

Even if, according to these criteria, multithreading is likely to benefit an application, the programmer still needs to consider the cost, in terms of additional complexity, of writing multithreaded code. In general, most server manager code will probably benefit from multithreading, which is provided by default by DCE. Most server applications will therefore choose to be multithreaded and incur the extra costs of creating thread-safe code. Whether client code will find the extra complexity of multithreading worthwhile really depends on a careful assessment of the listed criteria for each program design. There is no way to predict what a “typical” client will do.

Specifying the Number of Threads

The RPC runtime allows server applications to specify the number of manager threads available to handle concurrent RPCs via the *max_calls_exec* parameter of the **rpc_server_listen( )** routine. The runtime also allows applications to specify the number of unhandled calls that can be queued via the *max_call_requests* parameters of the **rpc_server_use_ * protseq *( )** routines. In theory, these two values should be set in conjunction, but in practice, the interpretation of the *max_calls_requests* parameter is highly dependent on protocol and implementation.

For example, in a connection-oriented protocol based on Berkeley sockets, the socket backlog—the number of connections which may be queued on a socket pending acceptance—typically has a value of five.

Portable applications should therefore not rely on *max_calls_requests* as anything more than a hint to the runtime about the number of queued calls desired. Note well that the *max_call_requests* parameter *does not set* the number of calls that can be handled concurrently. That is strictly a function of the number of call threads, as specified by *max_calls_exec*. The *max_call_requests* parameter simply specifies (as a hint) the number of calls that can be queued prior to being picked up by call threads.

Scheduling Policies

The default thread scheduling policy provides round robin time-slicing and guarantees that even low priority threads will get to run. For servers, this policy will provide at least the benefit of fair access to server processing time for multiple callers, even when no real parallelism is provided by multiple threads of execution.

Thread Safety

Thread safety involves two issues. The first is *blocking behavior*. Blocking I/O should block just the thread doing the I/O, not the entire process. The following scenario illustrates the kind of problem that can occur when an application fails to observe this rule:

1. The client side of the application executes a blocking I/O call such as a **read( )** from the keyboard.
2. The **read( )** sleeps for an indeterminate amount of time. All threads in the client process are blocked.
3. A timer thread in the client RPC runtime, which manages the client side of the RPC protocol, is among the blocked threads. Eventually the server side times the connection out, even though the client application is still running.

The second thread safety issue is *reentrancy*. Routines that operate on shared objects must have appropriate locking in place. A typical reentrancy problem is as follows:

1. The application invokes a nonreentrant **malloc( )**.
2. DCE threads interrupts the **malloc( )** and the interrupt handler executes a properly reentrant **malloc( )**. The reentrant **malloc( )** examines a lock and incorrectly infers that nobody else is currently doing a **malloc( )**.
3. Global data governing memory allocation for the process becomes corrupted.

These thread safety issues arise in the following two contexts for DCE applications:

- Even when application code is not itself multithreaded (for example, client code that does not make any explicit **pthread** API calls), both client and server applications are still multithreaded as a result of threads created by the RPC runtime. While such single-threaded application code need not itself be reentrant, it must still avoid blocking the entire process, and it must take care that any library routines that it calls, which may also be called by runtime-created threads, are reentrant.
- When application code is itself multithreaded (which is the default for server manager code), it must, in addition to obeying the rules above, also be reentrant; all access to shared objects must be protected by locks.

Obviously, providing for the second condition in explicitly multithreaded code is the application's responsibility. The **pthread** API provides a set of facilities that can be used for this purpose. To provide for the first condition, which applies to all application code, DCE implementations provide a mechanism to make system and library calls thread-safe. This may be implemented either by providing a set of *wrappers* for unsafe calls or by providing reentrant libraries and a nonblocking kernel threads implementation. Applications must always be built using either the appropriate wrapped calls or linked to the appropriate reentrant libraries.

DCE implementations provide, at the least, via wrappers or some other mechanism, the set of thread-safe calls shown in Table 1 on page 39.

Applications should not assume that a call to any routine not on this list is necessarily thread-safe. Whether other routines are safe to call from a DCE applications depends on the following factors:

- Application code that is single threaded (that has not explicitly created any application threads via calls to the **pthread** API) need not concern itself about reentrancy of routines not on this list, since all library and system calls made by RPC created threads are included in this list. However, such application code must still take care that no calls it makes will block the entire process.
- Application code that is multithreaded must exercise caution when making any call not on this list. Non-reentrant library calls may be wrapped by the application using **pthread_lock_global_np( )**, although this practice is discouraged since this call is not portable. The global lock can be used only in limited circumstances; the approach will work only if all threads in an application follow the same rule. Failure to observe these restrictions can lead to deadlocks. Note also that this approach will not work with any call that could block the whole process, for example by making a blocking I/O call.

Table 1. Thread-Safe Calls

_sleep( )	**accept( )**	**atfork( )**	**calloc( )**
catclose( )	**catgets( )**	**catopen( )**	**cfree( )**
close( )	**connect( )**	**creat( )**	**ctermid( )**
cuserid( )	**dup( )**	**dup2( )**	**fclose( )**
fcntl( )	**fdopen( )**	**fflush( )**	**fgetc( )**
fgets( )	**fopen( )**	**fork( )**	**fprintf( )**
fputc( )	**fputs( )**	**fread( )**	**free( )**
freopen( )	**fscanf( )**	**fseek( )**	**ftell( )**
fwrite( )	**getc( )**	**getchar( )**	**gets( )**
getw( )	**isatty( )**	**malloc( )**	**mktemp( )**
open( )	**pclose( )**	**pipe( )**	**popen( )**
printf( )	**putc( )**	**putchar( )**	**puts( )**
putw( )	**read( )**	**readv( )**	**realloc( )**
recv( )	**recvfrom( )**	**recvmsg( )**	**rewind( )**
scanf( )	**select( )**	**send( )**	**sendmsg( )**
sendto( )	**setbuf( )**	**setbuffer( )**	**setlinebuf( )**
setvbuf( )	**sigaction( )**	**sigwait( )**	**sleep( )**
socket( )	**socketpair( )**	**sprintf( )**	**sscanf( )**
system( )	**tempnam( )**	**tmpfile( )**	**tmpnam( )**
ttyname( )	**ttyslot( )**	**vfprintf( )**	**vprintf( )**
vsprintf( )	**write( )**	**writev( )**	

What follows is a summary of the thread-safety rules that should be followed when using the **pthread** facilities. The list is by no means comprehensive; it describes the places where multithreaded applications most frequently go wrong.

- Access to all shared objects should be protected by the appropriate synchronization mechnisms. The pthread global lock is not appropriate for such synchronization.
- Mutexes should be used only to protect resources held for a short period of time. In particular, note that **pthread_mutex_lock( )** is *not* a cancellation point.

Resources needing to be held exclusively for a long time should be protected by condition variables rather than mutexes, as this will not inhibit cancelability (see “Cancellation Points” on page 43).

- A shared object should be protected by only one mutex.
- Be sure to use the available thread-safe library calls. These may be available as wrapped routines, via the **pthread.h** header file, or your implementation may supply reentrant libraries which must be linked with DCE applications.
- Avoid nonwrapped process-blocking system calls, such as **wait( )**.
- When threads need to acquire more than one mutex at a time, create a locking sequence and require that all threads follow the sequence.
- Do not make any assumptions about the atomicity of operations, as these are unlikely to be portable.
- In general, to avoid priority inversion, when three or more threads of different priorities access a lock, associate a priority with the lock and force any thread to raise its priority to the lock priority before acquiring the lock. Note that the default scheduling policy (**SCHED_OTHER**) mitigates the effects of priority inversion by giving low-priority threads a chance to execute (and thus release held locks) even when higher-priority threads are eligible to run.
- You may be able to use the global locking call **pthread_lock_global_np( )** when calling into libraries not known to be thread safe.
- Use the **atfork( )** routine to keep the state of mutexes consistent across calls to **fork( )**. Note, however, that this routine is not considered portable. Try to create threads rather than processes whenever possible.
- Call **pthread_cond_wait( )** from within a predicate loop, as in the following example:

```
while (test_condition)
    pthread_cond_wait();
```

- Set thread attributes via a **Athread attributes object** before thread creation. Changes to a thread attribute object after a thread has been created will not affect the thread’s attributes. A thread can straightforwardly change its own scheduling attributes by calling **pthread_set_scheduler( )** and **pthread_set_prio( )**, but cannot reliably change the attributes of another thread once it has been created.

See “Canceling Threads” on page 42 and “Signals” on page 46 for specific guidelines relating to cancels and signals.

Threads Programming Topics

The subsections that follow contain discussions of the following aspects of multithreaded DCE application development:

- Thread handles and their use
- Storage for thread specific data
- Canceling threads
- Signals

Thread Handles

The **pthread** package provides thread handles to identify threads; these are returned as the *thread* argument to **pthread_create( )**. Applications supply thread handles as thread identifiers to the **routines pthread_join( )**, **pthread_detach( )**, and **pthread_cancel( )**. Thread handles should be treated as opaque data; they may be compared by calling **pthread_equal( )**, but any other operations on thread handles are likely to be nonportable and are thus discouraged.

Storage for Thread Specific Data

The **pthread** package provides the ability to allocate per-thread global storage using per-thread data keys. That is, an application can create storage that has global scope within a thread but which is private to each instance of that thread. To do this, the application creates a global data key by calling **pthread_keycreate( )**. Each thread then typically allocates storage of the required type and associates this instance with the global key by calling **pthread_setspecific( )**. Routines that need to access the per-thread storage do so by calling **pthread_getspecific( )**, which returns the address of the thread's private instance.

The following code fragments show a sample model of per-thread-data key use:

```
/* Declare global data key storage */

pthread_key_t key;

main()
{
          .
          .
          .

    /* Create exactly one instance of the key.  You could also use  */
    /*   a pthread_once() routine...                                */

    status = pthread_keycreate(&key, (pthread_destructor_t) destroy);
          .
          .
          .
    /* Start some threads...                                        */
          .
          .
          .
}

/* The following routines are called in each of the threads.       */
/*  They access the thread's private instance of the "global"       */
/*  value.                                                          */

/* The following routine sets the value to a thread-specific        */
/*  value...                                                        */

void write_global(mytype value)
{

    mytype *global_var;

    global_var = (mytype*) malloc(sizeof(mytype));
    pthread_setspecific(key, (pthread_addr_t)global_var);
    *global_var = value;
}
/* The following routine returns the thread-specific value ...      */
```

```
mytype read_global()
{

    mytype *global_var;

    /* Note the extra indirection; pthread_getspecific() returns    */
    /*  the address of the thread's private instance of the         */
    /*  storage...                                                   */

    pthread_getspecific(key, (pthread_addr_t*)&global_var);
    return (*global_var);
}
```

Canceling Threads

In order to program correctly for cancels, applications must be aware of the precise semantics of cancels in a DCE threads environment. The DCE threads package provides for per thread cancellation. Thread cancellation allows a thread to attempt to terminate a thread in the same process in an orderly manner. The basic model is that a cancel is generated for a thread at an unpredictable time as a result of some external event (typically, another thread calling **pthread_cancel( )**). Whether and when the canceled thread acts on a generated cancel depends on the thread's cancelability state, which may be one of the following:

disabled
: No cancellation takes place.

deferred
: Cancellation is deferred to cancellation points.

asynchronous
: Cancellation may occur at any time.

The default action for DCE threads on cancellation is that the thread calls any cancel cleanup routines that have been established and then terminates. In DCE threads a canceled thread receives a cancel as an exception, so a thread may establish a nondefault action by providing an exception handler.

However, this behavior is not recommended for two reasons. First, the exception handling mechanism is not itself portable. Second, the cancel mechanism is intended to provide for orderly thread termination. It is not designed as a generalized thread synchronization mechanism. (There is, for example, only one kind of cancel.) Threads should use condition variables for this purpose. (For the same reason, the use of **pthread_signal_to_cancel_np( )** is not recommended.)

Cancelability State

A thread's cancelability state is determined by the combination of two substates: *general cancelability* and *asynchronous cancelability*. These substates can be set to either **CANCEL_ON** or **CANCEL_OFF** by calls to the routines **pthread_setcancel( )** and **pthread_setasynccancel( )** respectively. A thread's cancelability state is determined by its general and asynchronous cancelability substates, as shown in Table 2 .

Table 2. Cancelability State

General	Asynchronous	Cancelability
Cancelability	**Cancelability**	**State**

Table 2. Cancelability State (continued)

General	Asynchronous	Cancelability
CANCEL_OFF	**CANCEL_OFF**	**disabled**
CANCEL_OFF	**CANCEL_ON**	**disabled**
CANCEL_ON	**CANCEL_OFF**	**deferred**
CANCEL_ON	**CANCEL_ON**	**asynchronous**

One awkwardness introduced by this mechanism for setting cancelability state is that threads cannot easily determine their current cancelability state, although **pthread_setcancel( )** and **pthread_setasynccancel( )** return the previous substates. When a thread is created, the default cancelability state is **deferred** (general cancelability set to **CANCEL_ON**, asynchronous cancelability set to **CANCEL_OFF**). A thread that needs to discover its current cancelability state should explicitly maintain this state in some place where it can be easily queried.

Cancellation Points

Applications need to be aware of where cancellation may actually occur when cancelability state is set to **deferred**. Cancellation points are points inside certain functions where a thread must act upon any pending cancellation request when cancelability state is **deferred** if the function would block indefinitely. If cancelability state is **asynchronous**, then every point is a cancellation point; that is, the thread may be canceled at any time.

If cancelability state is **deferred** then cancellation may occur at the following points:

- While waiting on a condition variable; that is, within **pthread_cond_wait( )** or **pthread_cond_timedwait( )**.
- While awaiting the termination of another thread (within **pthread_join( )**.)
- When **pthread_testcancel( )** is called.
- When **sigwait( )** is called.
- When a thread is waiting within **pthread_delay_np( )** (not a portable routine).
- During the timeslice interruption.
- Within the DCE threads I/O wrappers for system calls that block. These are as follows:
  - **read( )**
  - **readv( )**
  - **select( )**
  - **write( )**
  - **writev( )**
  - **accept( )**
  - **connect( )**
  - **recv( )**
  - **recvmsg( )**
  - **recvfrom( )**
  - **send( )**
  - **sendmsg( )**
  - **sendto( )**
- When **pthread_setasynccancel( )** is called, and either of the following apply:

– It has set the cancelability state to **asynchronous** (general cancelability and asynchronous cancelability are both enabled), it hasn't yet returned, and a cancel is pending.
– It was called to disable **asynchronous** cancelability state, but hasn't yet done so, and a cancellation request has been asynchronously delivered.

One important blocking routine that is not a cancellation point is **pthread_mutex_lock( )**, as this would create a domino effect so that every routine calling it would also become a cancellation point. Thus, mutexes should be used only to protect resources held for a short period of time so that noncancelability will not be a problem. Resources needing to be held exclusively should be protected by condition variables rather than mutexes, as this will not inhibit cancelability.

If a thread has not set **disabled** cancelability state, a cancellation request has been made to that thread, and the thread executes **pthread_testcancel( )**, the cancellation request must be acted upon. Similarly, if a thread has not set **disabled** cancelability state, a cancellation request has been made to that thread, and the thread is blocked at a cancellation point waiting for an event to occcur, then that thread must act upon the cancellation request. However, if a thread is suspended at a cancellation point and the event for which it is waiting has completed before a cancellation request is received and acted upon, the thread may resume normal execution and the cancellation request remains pending.

Cancellation Side Effects

Cancellation ordinarily involves cleanup in order to leave resources in an orderly state. Any side effects of acting upon a cancellation request occur before the first cleanup routine is called.

There are no side effects of acting upon a cancellation request while executing **pthread_join( )**.

The side effects of acting upon a cancellation request while in a condition variable wait are as follows:

- The mutex is reacquired before calling the first cleanup routine.
- In addition, while the thread is no longer considered to be waiting for the condition, no signals directed at the condition variable are consumed by the target thread if there are other threads blocked on the condition variable.

Using* pthread_cancel( ) *to Terminate a Thread: The **pthread_cancel( )** routine allows a thread to cancel itself or another thread. The routine is fully described in the **pthread_cancel(3thr)** reference page. Its use is straightforward, but if you use it to cancel a thread that makes use of mutexes or condition variables, you should keep in mind the following aspect of its operation.

The canceled thread receives the cancel in the form of an exception. If the thread has not disabled its cancelability by a call to **pthread_setcancel( )**, its effect is to immediately terminate the thread. However, if the thread happens to have acquired a mutex (including the global lock) when it is canceled, the mutex will remain in its locked state and no other thread will be able to acquire it. Moreover, the data that was protected by the mutex may be in an inconsistent state as a result of the thread's having been canceled in the middle of its operation on the data.

The easiest way to prevent this is simply to disable cancels before entering code for which access has been restricted by a mutex. If this is undesirable, you can explicitly handle a cancel by coding an exception-handling block.

This same possibility exists with condition variables, since the variable is protected by a mutex. An example of handling a cancel (or any other exception) while using a condition variable follows.

```
#include <pthread_exc.h>

    <...>

/* First, lock the mutex that protects the condition variable  */
/*      and the predicate...                                   */
pthread_mutex_lock(some_object.mutex);

/* Add this thread to the total number of threads waiting for  */
/*      the condition...                                       */
some_object.num_waiters = some_object.num_waiters + 1;

/* Enter the exception handling block...                       */
TRY

    /* Test the predicate condition...                         */
    while (! some_object.data_available)

/* If the desired condition is not yet true, wait for  */
/*    it to become true.  This next call also auto-    */
/*    matically releases the mutex...                  */
pthread_cond_wait(some_object.condition, some_object.mutex);

    /* Code to access data_available goes here */

    <...>

/* If a "cancel" exception occurs during the call to           */
/* pthread_cond_wait(), the thread will resume         */
/* execution in the FINALLY block following...         */
FINALLY

    /* Remove this thread from the total number of threads     */
    /*    waiting for the condition...                         */
    some_object.num_waiters = some_object.num_waiters - 1;

    /* Release the mutex, and then continue with the           */
    /*    exception --that is, cancel ...                      */
    pthread_mutex_unlock(some_object.mutex);
ENDTRY
```

Note that in order to handle the cancel as an exception, you must **#include** the **pthread_exc.h** header file rather than **pthread.h**; this allows you to use the DCE Threads exception interface.

Thread Cleanup

Each thread maintains a list of cleanup routines (handlers). The routines are placed on and removed from the list by the **pthread_cleanup_push( )** and **pthread_cleanup_pop( )** functions, respectively. These functions must appear as statements and in pairs within the same lexical scope.

When a cancellation request is acted upon, the routines on the list are invoked in the last in, first out (LIFO) order with cancellation disabled (cancelability state of **deferred**) until the last cleanup routine returns. When the last cleanup routine returns, thread execution is terminated. If other routines are joining with the target of the cancellation, a status of **(void*)** -1 is made available to them.

Cleanup routines are also invoked when the thread calls **pthread_exit( )**. Cleanup routines should never exit via **longjmp( )** or **siglongjmp( )**.

Asynchronous Cancel Safety

A function is said to be *asynchronous cancel safe* if it is written in such a way that entering the function with the cancelability state of **asynchronous** will not cause any invariants to be violated if cancellation should occur at any (arbitrary) instruction. Such functions are often written in such a manner that they need acquire no resources, and variables which they write that are visible outside their process are strictly limited.

Any routines that acquire a resource can not be made asynchronous safe. This unfortunately includes most routines that do useful work. The only function that is guaranteed to be asysnchronous cancel safe is **pthread_cancel( )**. In general, no other library functions should be called with cancelability state set to asynchronous.

Cancel Rules Summary

The following summarizes a set of cancel-related rules that should always be adhered to when programming with cancels:

- Applications should not use cancels as a synchronization mechanism. Condition variables should be used instead.
- **pthread_mutex_lock( )** is *not* a cancellation point. Resources needing to be held exclusively for a long time should be protected by condition variables rather than mutexes, as this will not inhibit cancelability.
- A condition wait (via **pthread_cond_wait( )** or **pthread_cond_timedwait( )**) is a cancellation point. A side effect of acting on a cancellation request while in a condition wait is that the mutex is (in effect) reacquired. The effect is as if the thread were unblocked, allowed to execute up to the point of returning from the wait, but at that point notices the cancellation request and handles it instead of returning.
- In general, most library calls cannot be assumed to be asynchronous cancel safe, and hence must *not* be called with cancelability state set to **asynchronous**.
- Cleanup routines should never exit via **longjmp( )** or **siglongjmp( )**.

In addition to the material covered in this section, “RPC Threads and RPC Cancel Semantics” on page 49 covers the additional semantics of cancels as applied to RPC threads.

Signals

Application developers must be aware of significant differences in the handling of signals between DCE threads and typical single-threaded environments. In DCE threads, some signals are handled on a per-process basis, and some are handled on a per-thread basis. This section explains the semantic details of DCE thread signal handling.

A signal is said to be *generated* for a process or thread when the event that causes the signal first occurs. Each process or thread has an action to be taken in response to each signal supported by the system or implementation. A signal is said to be *delivered* when the appropriate signal action for the process or thread is taken. A signal can be *blocked* (or *masked*) by a thread or process by establishing a *signal mask* containing the signal to be blocked.

The delivery of a blocked signal is deferred until it is unblocked. (Note that if the action specified for a signal is to ignore it, the signal effectively remains blocked.) During the time between its generation and delivery a signal is said to be *pending*.

Signals can be classified into the following two types, with differing semantics:

- *Synchronous signals* are generated by a specific thread and delivered to the same thread. Threads can establish nondefault per-thread signal handlers for sychronous signals by calling **sigaction( )**. Synchronous signals can be blocked on a per-thread basis by establishing per-thread signal masks.
- *Asynchronous signals* are generated by external events, not identifiable with a single thread. Asynchronous signals are handled on a per-process basis. An asynchronous signal is delivered exactly once to some thread in a process. All threads in a process share the same signal mask. Per-process handling of asynchronous signals can be established by calling **sigwait( )**.

DCE threads applications must handle synchronous and asynchronous signals differently.

Signal Masking

Signal masks can be examined and changed with the **sigprocmask( )** function. When a synchronous signal is masked via a call to **sigprocmask( )** it is masked for the calling thread. When an asynchronous signal is masked via a call to **sigprocmask( )** it is masked for the entire process.

Care must be taken when a thread unblocks an asynchronous signal. If another thread has blocked and is, or is will be, waiting for the same signal, the results can be unpredictable and may result in the other thread waiting forever. This problem can be avoided by having all handling of asynchronous signals occur in a single thread, as described in “Asynchronous Signal Handling” .

Synchronous Signal Handling

Threads should call **sigaction( )** to establish per-thread handlers for synchronous signals. The DCE Threads **sigaction( )** function only modifies the signal action behavior for the calling thread and only works for synchronous signals. Threads must not use **sigaction( )** for asynchronous signals.

Signal handlers should be careful in the actions they perform. In general, synchronous signal handlers should attempt to clean up and allow the thread to terminate. It is not advisable to attempt to continue after errors such as a segment violation, illegal instruction, and the like.

In general, the threads routines cannot safely be called within a signal handler. Furthermore, runtime libraries cannot reliably be used in signal handlers.

Asynchronous Signal Handling

Applications should handle asynchronous signals by having one thread (or possibly a few specific threads) call **sigwait( )**. The waited-for signals must be blocked before waiting. The recommended procedure is to establish a “signal catcher” thread that calls **sigprocmask( )** to establish the per-process mask for asynchronous signals and then calls **sigwait( )** to wait for the set of blocked signals. The following code fragment shows an example of a signal catcher thread start routine:

```
/*
 *  This is run by the signal catcher thread to handle async signals.
 *  We don't use sigaction() here because it won't work with
 *  async signals.  Note that signals must be blocked prior to being
 *  waited for.
 */

void signal_catcher(char *arg)
{
    sigset_t signals;
    int sig;

    sigemptyset(&signals);
    /* In this sample, we'll catch only SIGINT...                  */

    sigaddset(&signals, SIGINT);
    sigprocmask(SIG_BLOCK, &signals, NULL);
    while(1)
    {
        sig = sigwait(&signals);
        switch(sig)
        {
            case SIGINT:

                /* SIGINT specific actions here.                   */
                             .
                             .
                             .
                break;
            default:
                /* Not reached.  If we were waiting on other        */
                /*  signals. this would establish a default action   */
                /*  to exit ...                                      */
           continue;
        }
        break;
    }
    sigprocmask(SIG_UNBLOCK, &signals, NULL);

    /* Do termination clean up here.                               */
                .
                .
                .
    exit(1);
}
```

Signal Rules

The following rules summarize correct signal handing practices for multithreaded programs.

- Signals must be blocked prior to being waited for. The **sigwait( )** routine waits for blocked (masked) signals.
- In order to avoid unpredictable behavior, all asynchronous signal handling should be confined to one signal catcher thread. This may be extended to a set of signal catcher threads.
- The **pthread_cond_signal( )** routine cannot safely be used in a signal handler that is invoked asynchronously. In general, mutexes and condition variables are not suitable for releasing a waiting thread in response to a signal handler. When a thread must wait for an asynchronous signal, use **sigwait( )** instead.
- Signal handlers should not call the **pthread** routines. In general, runtime libraries cannot reliably be used in signal handlers.

Forking in a Threaded Application

The **fork( )** system call causes the creation of an exact clone of the caller's address space, resulting in the execution by two address spaces of the same code. In order to avoid the problems that would arise in a threaded environment when one thread, possibly without the others' knowledge, executes a **fork( )**, the POSIX model defines **fork( )** to result in the propagation of the calling thread only. Any other active threads are immediately terminated without notice.

The abrupt destruction of the other threads means that any mutexes they may have been holding at the time of the **fork( )** will persist in the locked (and therefore unacquirable) state. On the other hand, assuming that the call to **fork( )** is followed by a call to **exec( )**, then the outstanding mutexes will remain so only until **exec( )** is called, when the new process space will be reinitialized.

Thus, "out-of-state" mutexes are a problem for the forked thread only in the interval between the **fork( )** and the **exec( )**. Even so, as long as no calls occur here to routines outside the application, you can determine whether the thread is going to encounter any mutexes that could have been locked by the destroyed threads. However, it is impossible to be sure of this if calls into other libraries, which may have hidden interdependencies, occur in this interval.

Aside from these considerations, there is also the question of what happens when **exec( )** fails and execution returns to the original forking (and now lone) thread, which is left with an address space that may contain out-of-state mutexes (as well as an inconsistent state in the data protected by the mutexes) as a result of the **fork( )**.

DCE does not support the "simple" **fork( )**; it supports only the **fork( )** and **exec( )** sequence. For cases where forking in the presence of threads is felt to be necessary, DCE threads provides a mechanism, the **atfork( )** call, which allows you to install "fork handler" routines for an application or a library. These routines will be automatically run as follows:

- A routine that will be run just prior to the fork in the parent process; that is, just before all of the other threads are terminated
- A routine that will be run in the child process just after the fork occurs; that is, just after all the other threads are terminated
- A routine that will be run in the parent process just after the fork occurs; that is, just before the parent (forking) thread resumes execution

RPC Threads and RPC Cancel Semantics

Each RPC occurs in the context of a thread. A thread is a single sequential flow of control with one point of execution at any instant. When an application thread extends across client and server execution contexts via the DCE RPC mechanism, the local execution contexts are joined by an abstraction known as an *RPC thread*. The RPC thread attempts to extend local thread semantics to the situation in which execution is extended over two or more local contexts. Specifically, the RPC mechanism tries to make RPC cancels look to the application as much like local cancels as possible.

The semantics of cancels across RPCs are slightly different from the semantics across local (procedure) calls. The differences can be summed up as follows:

1. If the cancel state is **disabled** when an RPC is made, then, regardless of what is done to the cancellation state on the remote procedure, no cancels will be seen by the remote procedure.

   This is because a cancel must be noticed in the client-side runtime in order for it to be forwarded to the server. However, if the cancellation state has been set to **disabled** when an RPC is issued, then since the client-side runtime does not enable cancels, the client-side runtime will never notice if a cancel has been issued against the calling thread; subsequently, the cancel remains pending and unnoticed by the client-side runtime, even if the server side has changed the cancellation state (for instance, to **deferred**).

   Furthermore, since lexical scoping of changes to the cancellation state is enforced by RPC, the cancellation state in effect at the time of the RPC call is restored upon completion of the call. Thus, any state changes made on the server side of the call are lost. Any issued cancels remain pending as the server-side state change is “undone” by the client-side runtime prior to returning to the calling thread. In this instance, if a cancel arrives after the callee returns, the cancel will not be acted upon.

   This behavior contrasts with the local procedure call case: if cancel state is **disabled** when a local procedure call is made, and the callee sets the cancellation state to **deferred**, then if a cancel arrives and the callee hits a cancellation point, the cancel will be acted upon. Furthermore, if the cancel arrives after the callee returns, the cancel will be acted upon when a cancellation point is arrived at in the caller.
2. If cancelability state is **deferred**, then cancellation requests will be sent to the server where they will be handled according to the server’s setting of the cancelability state for the application thread extension (that is, the call thread) in the server. If ignored at the server, the client side would then effect the cancel upon return from the RPC, so the cancel would not be lost or incorrectly handled. In particular, the timeslice interrupt (context switch) is a cancellation point in DCE threads, so that even if a cancel were ignored by the server side, when the RPC returns, the thread will be at a cancellation point.
3. If cancelability state is **asynchronous**, then cancellation can happen at any time. In general, this state is not recommended across the scope of an RPC in line with the rule that most routines that do useful work are not asynchronous cancel safe and thus should not be called with asynchronous cancelability state.

Chapter 3. Security

For the purposes of the discussion in this chapter, the security services provided by DCE are assumed to consist of three elements: authentication, access control, and data protection. (The DCE Audit Service, which is also a part of DCE security, is described in the *OSF DCE Application Development Guide—Core Components*.)

The roles of these three elements can be broadly defined as follows (rigorous definitions can be found in the *AES/DC Security* volume, which is the definitive exposition of DCE security):

- *Authentication* establishes whether service requestors are who they say they are.
- *Access control* provides mechanisms that applications can use to establish whether a given requester is permitted to perform some operation.
- *Data protection* guarantees the secrecy and integrity of data exchanged between clients and servers.

As with other DCE services, use of the security services raise two kinds of policy questions. At one level, application programmers must decide which services and levels of service to employ. At a second level, once a service has been chosen, the application programmer must make many decisions about how to use it. This chapter covers both levels of policy, although it focuses mainly on the lower-level policy issues specific to each service. This emphasis is due both to the fact that the higher-level issues are relatively few—mainly whether to use a given service or not—and to the belief that it is far easier to understand the general issues once the specifics are clear.

Security is an especially complex area from the policy point of view. Security systems must anticipate threats both from human ingenuity and random accident, and it can be difficult—perhaps impossible—to be confident that no serious threat is being overlooked. DCE security provides an extensive security model that applications can incorporate in a few well-integrated chunks. Thus applications can get the benefit of the DCE security design—and the extensive, specialized analysis that went into it—with relatively little effort. Applications should avoid creating security solutions *ad hoc* and should stick closely to the solutions provided by DCE security. Unless the programmer is a security specialist, it is extremely unlikely that an application-specific solution will provide better security than the DCE security services, and it is practically guaranteed that such solutions will contain unforeseen weaknesses.

The Basic Security Model

At a high level, the DCE security model is as follows. Servers specify the authentication service they use (currently either none or DCE secret key). Clients request an authentication service (which may be none) when making a call. When a server specifies an authentication service, it is specifying the service it will use *if authentication is requested by the client*. This allows a server to permit both authenticated and unauthenticated access. When a client requests authentication and the server provides it, authentication is carried out silently by the runtime as part of the RPC protocol. The runtime will fail the call if the client cannot be authenticated. When no authentication is requested, none is performed. If the client requests authentication and the server does not provide it, the runtime will fail the call.

Table 3 shows how client and server authentication actions affect RPC calls. Clients specify an authentication service for a binding handle by calling **rpc_binding_set_auth_info( )**. Servers register an authentication service by calling **rpc_server_register_auth_info( )**. The possible values are **rpc_c_authn_none** for no authentication and **rpc_c_authn_dce_secret** (or **rpc_c_authn_default**) for DCE secret key authentication.

Table 3. Authentication

Client Specifies	Server Registers	Authentication
rpc_c_authn_none	**rpc_c_authn_none**	No authentication performed
rpc_c_authn_none	**rpc_c_authn_dce_secret**	No authentication performed
rpc_c_authn_dce_secret	**rpc_c_authn_none**	Call rejected by RPC runtime
rpc_c_authn_dce_secret	**rpc_c_authn_dce_secret**	Authentication performed

Authentication establishes only that each of the parties is a principal known to the authentication service, and that each party knows who the other is. Servers typically make an explicit authorization decision using one of the DCE authorization services, to decide whether a given authenticated principal should in fact be granted access to some operation or resource. In most cases, clients will not be satisfied with the mere assurance that they are communicating with an authenticated principal. Clients must then check the authenticated identity of the server to be sure that it is one with which they are willing to communicate. Note that this kind of server identity check is normally made at a low level of granularity: typically once per client-server session. Server authorization of clients is usually much more specific: typically once per remote operation.

Authorization is based on the identity of the caller, which may be expressed either as a principal name or as a set of privilege attributes. What the RPC authentication model provides to the server are, essentially, guarantees as to the authenticity of the identity, and possibly, the privilege attributes of the caller. Since an identity without such guarantees would be useless for access checking, authorization is supported only for authenticated RPCs. If the client chooses to call unauthenticated, the runtime permits the call and does not provide any authentication information.

It is entirely up to the application manager code to make an access decision based on any authentication and authorization data provided by the runtime for a client. Clients specify an authorization service for each binding: either none (**rpc_c_authz_none**), client principal name-based authentication (**rpc_c_authz_name**), or DCE credential-based authentication (**rpc_c_authz_dce**). When a server manager operation is invoked (implying either that no authentication was performed or that authentication was performed and succeeded), the application can retrieve any authorization information by calling **rpc_binding_inq_auth_caller( )**.

The application manager must then make an access decision based on the retrieved information. The DCE ACL facility provides applicaton support for ACL-based authorization using the client credentials. This is the recommended authorization scheme.

In addition to authentication and authorization, the DCE security services can also provide various levels of data secrecy and integrity guarantees. The basic model is that the client application requests the minimum acceptable protection level. The

runtime then provides the lowest supported protection level that is *at least* as high as the one requested by the client. If the runtime cannot provide at least the requested level, it fails the call. Supported levels as well as the services provided by each level depend on the authentication service in use, so clients must take care to request a level that is meaningful for the authentication service they have specified.

Application Roles

Each of the elements of DCE security makes very different demands on the application. In the case of data protection, the application need only specify a protection level. The RPC runtime takes care of data protection transparently and the guarantees provided are fairly easily understood.

In the case of authentication, clients and servers have to do more work to establish the required state for authentication to take place. The required steps are described in detail in "Authentication Model" . Once this initialization is taken care of, the RPC runtime provides authentication transparently.

The authorization component of DCE security requires the most work from the application. Essentially, DCE provides applications with a set of mechanisms for access control. These include the following:

- The authenticated identity and privilege attributes (in the form of credentials) of service requesters, provided by the RPC runtime to servers.
- ACLs which servers may associate with objects they control.
- A default mechanism for determining a service requestor's privileges from an ACL and the requestor's credentials.
- Tools for administering ACLs.

Servers that use the DCE ACL-based authorization services must do a fair amount of initialization to create an ACL manager. Each protected operation must then explicitly call the ACL manager to make an authorization decision for each protected operation. A set of ACL management APIs is provided to make these tasks easier, but the work required remains nontrivial. The steps are covered in detail in "Authorization" on page 65 .

Authentication Model

The DCE authentication model is currently based on the Kerberos shared secret key protocol. In theory, the application-level interface to authentication is sufficiently abstract that an alternative authentication protocol can be implemented. However, given that none so far has been implemented, it would be difficult to define protocol-independent authentication policies based on a realistic understanding of the behavior of alternate authentication services or the as yet unspecified programmer's interface to such services. The policy recommendations of this section do, therefore, make the assumption that Kerberos is the underlying authentication protocol. No guarantees can be given as to their appropriateness if an alternative authentication protocol is implemented.

The DCE Authentication Model

The authentication mechanism is based on two fundamental constructs: *principal identities* and *secrets* (keys). These are, in a sense, the fundamental data of authentication. The basic authentication policy issues therefore have to do with how applications manipulate this data: how they acquire their principal identities and how they maintain the security of their secret keys. This section discusses these questions. The following discussion assumes an understanding of the basic transactions of the Kerberos protocol as implemented by DCE. That is, it assumes that you understand such concepts as conversation keys, tickets, a trusted computing base, and the like, as described in the *AES/DC Security* volume. It does not assume that you know anything about the details of protocol encoding, encryption mechanisms, and so on.

At a very general level, authentication (Kerberos)-related activity takes place in three stages:

1. Before any application can make use of the authentication service, some *administrative actions* are required, mainly to establish the required principal identities and related secret keys.
2. Some *application-level actions* are then required of the client and server principals: fundamentally, the client must obtain validated credentials, and the server must point the RPC runtime to the storage for its keys. Note that, strictly speaking, the server need not itself obtain any credentials, as these are only used by the client of the Kerberos exchange. However, since servers typically must also act as clients (of the name service, for example), they will normally also need to acquire credentials.

   In the case of the client, the application-level actions required to obtain credentials are normally carried out by a login program before the client is run, and the client inherits valid credentials. Therefore, this stage of activity is not usually carried out explicitly by clients. In the case of the server, these activities are usually carried out by the server explicitly. The reasons for this difference are one of the topics covered in the discussion that follows.
3. Authentication related RPC protocol activity is then carried out transparently by the RPC runtime during each call.

In addition, server application code needs to make authorization decisions based on the assumption that authentication has been carried out, but these belong more properly to the realm of authorization, as described in “Authorization” on page 65 .

Note that the application code proper need only concern itself with item 2 in the above list. This item is therefore the appropriate realm for policy recommendations about application-level authentication. Item 1 is an administrative task required for the installation and maintainance of the application. Nevertheless, the required administrative actions depend on how the application treats authentication and are, therefore, indirectly a policy concern for the application programmer. What this policy guide recommends is essentially a standard application security model that results in a standard administrative task. Note that, once the administrative and application setup covered by items 1 and 2 have been performed, item 3 is handled transparently by the RPC runtime.

Application-Level Authentication

One of the obvious conclusions to be drawn from the general discussion of DCE authentication is that application-level client and server authentication responsibilities are highly asymmetrical:

- Clients typically inherit identities, while servers assume them implicitly.
- Clients are concerned with credentials while servers are concerned with keys.

The reasons for these asymmetries have to do both with the underlying asymmetry of the Kerberos model and with an underlying model of RPC client and server behavior that is also asymmetrical.

From the Kerberos point of view, the basic model is that a client acquires and holds tickets (credentials), valid for some period of time. These function as temporary proxies for the client's secret. The server, on the other hand, makes use of no such proxy: it needs constant access to its secrets in order to decrypt new client requests and discover the applicable conversation keys.

From the RPC point of view, the basic model is that servers are persistent entities in the sense that they normally perform services on behalf of more than one client principal session. This may mean that servers are persistent in time: that is, that they run for a long time, possibly for as long as the machine they are running on is up and running. But even servers that are invoked on demand (and therefore that run for a short period of time) can be invoked by multiple clients and, during their short lives, may well perform services for clients other than the invoker.

Clients, on the other hand, will typically be invoked by an interactive principal to run within the scope of a single principal login session. Such clients can therefore usefully acquire their credentials from the principal who invoked them. Note, however, that there is nothing to require clients to behave in this manner. A persistent client can easily be written that assumes its own identity, manages keys, and acquires and updates credentials. The basic authentication policies described here can be easily extended to cover this case.

For a client that runs with an inherited identity, the principal security problem—the maintainance of its secrets—is reduced to the problem of maintaining the security of its credentials while they are valid. The client is basically passive in this respect, depending on the local operating system to prevent unauthorized access to the credential cache of the DCE principal that initiates the client application. Direct management and discovery of keys (for example, reading them from a configuration file) is not required of such clients. Typically, such an application can do nothing about the security of the principal's keys used to acquire credentials, since all the authentication-related state is inherited. The client's real security responsibility is therefore negative: not to take any action outside of the specified authentication policy model that could compromise security for the identity with which it runs (for example, indiscriminately giving other processes access to its credentials).

[1] Clients may or may not be concerned with the identities of the servers they call. The Kerberos authentication exchange is mutual in the sense that both clients and servers must have genuine authentication identities to participate successfully. However, a client may not trust a server simply because it can successfully authenticate to the client. The client may want to make RPCs only to servers with

specific principal identities that it trusts. In this case, the client has the additional security task of safely maintaining a list of acceptable server identities with which it is willing to communicate.

For the server, the basic authentication problem imposed by the DCE secret key authentication protocol is the maintainance of keys. This depends on local operating system access control to the key storage (typically a so-called **keytab** file) for the DCE principal identity used by the server. However, since servers normally also need to acquire credentials (in order to behave as clients of other services), application programmers need to think carefully about how the server identity is acquired. In general, it is not satisfactory to have servers run with credentials inherited from human logins. For one thing, this requires the server to share keys with human users. This means that the server either needs to have access to the default key storage used by human principals (typically the default **keytab** file, probably owned by root) or it needs to keep separate copies of user keys in local storage. Both of these schemes decrease the security of keys, and the latter makes key management difficult.

A straightforward scheme that meets these requirements is to have the server identity supplied by the invoker (or a configuration file) and have the server assume this identity via a series of security service calls. The only administrative overhead is in establishing at least one principal and the required **keytab** file. This is typically handled through **dced** facilities.

Obtaining an Authentication Identity

DCE clients normally inherit valid credentials from the logged-in principal who invokes them. DCE servers normally need to establish an identity explicitly. The steps they take, and their relation to the Kerberos protocol, are described in this section.

In actual practice, clients want to obtain a privilege ticket granting ticket (PTGT), since they want to prove not only their identities to servers, but also to provide their certified privileges (in the form of credentials). However, from the point of view of authentication, the principle is the same: the client needs some kind of TGT. For simplicity's sake, the following discussion pays little attention to the distinction between TGTs and PTGTs (as well as the many extra protocol steps involved.)

The terms *credentials*, authentication identity, and *login context* are often used to mean vaguely the same thing. Here however, we will use *credential* to mean a ticket held by an application. An application's credentials at any point typically consist of a number of cached tickets, including a TGT, PTGT, and a variety of service tickets. (Also, an application may have acquired more than one principal identity, in which case it will have credentials for each.) We will use authentication identity to mean the set of authentication-related data—including

---

1. This is another of the basic asymmetries of the Kerberos-based security mechanism. Servers can control client access by demanding that the client be authenticated and then making authorization decisions based on the client's authenticated privilege attributes. Clients can only require that the servers they call be authenticated. This leaves the client with three server authentication options:

   1. The client doesn't care about the identity of the server.
   2. The client demands that the server be authenticated, but does not care which authenticated identity the server uses.
   3. The client only trusts principal identities known to it directly or indirectly, such as by being a member of a trusted group.

   Application steps for checking authenticated server identity are discussed in “Authentication Model” on page 53 .

credentials—referred to by a login context. Finally, we will use *login context* to mean the opaque handle to authentication-related data that applications use.

An instance of authentication identity data in its various states is represented to an application as an opaque login context (**sec_login_handle_t**). An application obtains an authentication identity by calling **sec_login_setup_identity( )**, which returns a login context containing the TGT data. An application validates the identity by passing the login context to **sec_login_validate_identity( )**. Parts of the TGT obtained by **sec_login_setup_identity( )** are encrypted using the requesting principal's key, obtained from the registry. The **sec_login_validate_identity(**) routine requires the principal's key (from the **keytab**) to perform the decryption. Once this has occurred, the client runtime also performs the further steps necessary to acquire a PTGT and other tickets.

The setup and validation operations are separate in order to minimize the amount of time that the application needs to maintain the principal's key in its address space. Applications obtain the principal's key by calling **sec_key_mgmt_get_key( )**. The call to **sec_login_validate_identity( )** destroys the key in place before returning. Applications should not violate the intention of this design by keeping the key in memory longer than necessary. That is, they should make the required calls strictly in the sequence illustrated in the following code fragment:

```
sec_login_setup_identity(prin_name, sec_login_no_flags,
                         &login_context, status);

sec_key_mgmt_get_key(rpc_c_authn_dce_secret, keytab,
                     prin_name, 0, (void**)&keydata,status);

sec_login_validate_identity(login_context, keydata,
                            &reset_pwd, &auth_src, status);
```

These calls are bundled into the **dce_server_sec_begin( )** routine.

Once an authentication identity has been obtained and validated, an application that intends to use the identity for authenticated RPC normally turns it into the default login context by calling **sec_login_set_context( )**. As the default login context, an authentication identity is implicitly available to authenticated RPC calls made within the same process. An application, such as a client, that inherits an authentication identity inherits it as the default login context.

The Authenticated RPC Call

Once an application has either inherited or established a validated authentication context, it establishes authentication for RPCs by annotating the binding handles on which those calls are made. Clients do this by calling **rpc_binding_set_auth_info( )**. No further action is required of the application: when an RPC is made on such a binding handle, all further authentication is carried out silently by the RPC runtime.

The call to **rpc_binding_set_auth_info( )** requires three pieces of authentication-related state:

1. The authentication service to use: either DCE secret key or none.
2. The login context to use. Most applications will specify the default login context (by setting the *auth_identity* parameter to NULL).
3. A principal name for the server being called.

Note that applications may need to establish a default login context even if they do not explicitly call **rpc_binding_set_auth_info( )** to set this context for a specific binding handle. In particular, access to name and other services involves authenticated RPC calls made by the runtime on the application's behalf. In these cases, the application does not have a chance to call **rpc_binding_set_auth_info( )** explicitly. These implicit calls therefore use the default identity for authentication purposes. It is mainly for this reason that servers need to establish a validated authentication identity for the principal under which they run and make this the default login context.

The principal name specified to **rpc_binding_set_auth_info( )** establishes the principal for which Kerberos service tickets will be requested for RPCs on the binding handle. An application making RPC calls may or may not care about who the server principal is. The client may be satisfied to call any server that provides the service it wants, or the client may need to trust the server and thus require a trusted server principal identity.

Typically, a client learns the principal identity of a server by calling **rpc_mgmt_inq_server_princ_name( )**. If the client is willing to call any server, the returned principal name may be passed to **rpc_binding_set_auth_info( )** without further checks. If the client must trust the server, then the client needs to check the returned principal identity against a list of (one of more) acceptable values. The client needs to obtain this list by some application-specific means.

Note that it is not the call to **rpc_mgmt_inq_server_princ_name( )** or any subsequent checks on the returned name that actually authenticates the server to the client. A malicious server could certainly arrange to return a false principal name. However, a false name would be useless for authentication since the false server would not have access to the secrets (keys) of this identity. However, the client does need to protect its list of acceptable server identities to prevent a malicious server from modifying the list to include its own identity.

Managing Keys

An application that wishes to perform the server side of the Kerberos protocol exchange is principally concerned with managing its keys. Keys are normally stored in **keytab** files which must be in the local host file system. The server needs local system permission to read and write them, and they must be protected from any access by other local identities.

Note: Keytab files are normally created by administrative action. Be aware that the local identity or the process running **rgy_edit** determines the initial local ownership of files created by **ktadd**.

This means that the server needs its own local identity too, to correspond to its DCE identity. Keytab files should be owned by this local identity. The programmer or installer must arrange for the server to run under this local identity, and only a locally privileged user should have execute permission for the server. On UNIX systems this can be arranged by having the server run **setuid( )** to the chosen local identity and giving execute permission only to specific local users.

Because the degree of integration between local and DCE login varies with DCE implementations, it is difficult to give more general advice about local identities. As the following paragraphs explain, however, it is generally not a good idea for the server to run with the DCE identity of a human user. If DCE and local identities are

the same, the same guideline must be applied to local identities. That is, the server's local identity should not be that of a human user.

When a server is initialized, it will get its key from its **keytab** file. The keys installed in **keytab** files should *not* be tied to some human readable password: that is, they should be randomly generated and updated frequently (as enforced by administrative policy). This means that servers do not have DCE passwords; passwords should be used for human login only.

In general, the domains of human and nonhuman users should be separate. For example, a human user needs a password from a restricted domain (typable on the keyboard), hence keys tied to passwords are generally less secure than keys not tied to passwords. Furthermore, when keys are tied to passwords, key management is much harder.

Servers therefore should acquire their own nonhuman, server-specific identities. Requiring a small amount of administrative overhead to set up a DCE identity for a server-specific principal is not an onerous task for a server that is not frequently installed. In an identity-based security system, the server's principal name is the essential persistent security datum for a server. Its importance is in some ways equivalent to that of the server's bindings.

One might complain that keeping keys in a **keytab** file places all of the server security burden on the local operating system, and this is correct. But an alternative scheme, such as requiring a user password to start a server, does nothing to improve on this. Indeed, it is the cardinal fact of DCE security that, on any local system, it is only as secure as the local operating system upon which it runs. It is therefore a sound policy to make this dependency explicit rather than erecting an illusory layer of DCE security on top of it.

Default Server Authentication Steps

The default model for server authentication consists of the following steps:

1. The server specifies a server-specific **keytab** file and server-specific principal name when it calls **rpc_server_register_auth_info( )**.
2. The server acquires valid credentials for its server-specific identity via a series of **sec** API calls.
3. The server does periodic key management by establishing a separate thread that calls **sec_key_mgmt_manage_key( )**. This keeps the server's key up to date according to local key management policies and thus prevents the server from becoming inoperable because of an expired key.
4. The server contains code to check and, if necessary, revalidate and recertify its credentials when undertaking operations that require valid credentials (such as name service export and unexport operations).

The following sample functions, reproduced from the sample DCE application reprinted in full in "Chapter 10. A Sample Application" on page 177, implement credential acquisition, credential revalidation, and key management.

In order to save space and to improve the readability of the text, the code shown below has been slightly edited: all status checks, and all calls to the DCE serviceability interface (to print or log status or informational messages), have been removed.

The managekey Routine

The **managekey( )** routine manages the server principal's key, making sure that it never expires.

```
/******
 *
 * managekey -- Make sure the server principal's key is changed before
 *              it expires.
 *
 *      The key management thread which runs this function is created
 *       in server_get_identity(), below.
 *
 *
 ******/

void managekey(char *prin_name){          /* Server principal name      */
        unsigned32 status;

        status = error_status_ok;

        sec_key_mgmt_manage_key(
                rpc_c_authn_dce_secret,   /* Authentication protocol    */
                KEYTAB,                   /* Local key file             */
                (idl_char *)prin_name,    /* Principal name             */
                &status);

}
```

The server_get_identity Routine

The **server_get_identity( )** routine sets up a new server identity.

```
/******
 *
 * server_get_identity -- Establish a new server identity with valid
 *                        credentials.  This includes setting up a key
 *                        management thread.
 *
 *
 *    Called from main().
 *
 ******/

void server_get_identity(
unsigned_char_p_t prin_name,            /* Server principal name.         */
sec_login_handle_t *login_context,      /* Returns server's login context. */
unsigned_char_p_t keytab,               /* Local key file.                */
unsigned32 *status)
{

        pthread_t keymgr;
        sec_passwd_rec_t *keydata;
        sec_login_auth_src_t auth_src;
        boolean32 reset_pwd;

        *status = error_status_ok;

        /* Spin off thread to manage key for specified principal...       */
        if (pthread_create(&keymgr,           /* Thread handle.    */
                        pthread_attr_default,  /* Specifies default thread */
                                               /*  attributes.            */
                        (pthread_startroutine_t)managekey, /* Start rou-   */
                                               /*  tine; see above.       */
                        (void*)prin_name))     /* Argument to pass to start */
```

```
                                                /*  routine: serverprinci-  */
                                                /*  pal name.               */
        {
                dce_svc_printf(CANNOT_MANAGE_KEYS_MSG);
                return;
        }

        /* Create a context and get the login context...                    */
        sec_login_setup_identity(prin_name,
                sec_login_no_flags,
                login_context,
                status);
        /* Get secret key from the keytab file...                           */
        sec_key_mgmt_get_key(rpc_c_authn_dce_secret,
                keytab,
                prin_name,
                0,
                (void**)&keydata,
                status);

        /* Validate the login context...                                    */
        sec_login_validate_identity(*login_context,
                keydata,
                &reset_pwd,
                &auth_src,
                status);

        /* Finally, set the context...                                      */
        sec_login_set_context(*login_context, status);

}
```

The server_renew_identity Routine

The **server_renew_identity( )** routine makes sure that the server's credentials are valid.

```
/******
 *
 * server_renew_identity -- Make sure that credentials are still valid, and
 *                          renew them if they are not.
 *
 *
 *     This routine is called (with the current credentials) whenever a task
 *     is about to be attempted that requires valid credentials.  For an ex-
 *     ample, see the cleanup code in "main()" above.  A valid credential will
 *     nevertheless be considered invalid if it will expire within time_left
 *     seconds.  This gives a margin of time between the validity check that
 *     occurs here and the actual use of the credential.
 *
 *    Called from main() (but can be called from elsewhere).
 *
 ******/
void server_renew_identity(
unsigned_char_p_t prin_name,      /* Server's principal name.              */
sec_login_handle_t login_context, /* Server's login context.               */
unsigned_char_p_t keytab,         /* Local key file.                       */
unsigned32 time_left,             /* Amount of "margin" -- see above.      */
unsigned32 *status)               /* To return status.                     */
{
        signed32 expiration;
        time_t current_time;
        sec_passwd_rec_t *keydata;
        sec_login_auth_src_t auth_src;
        boolean32 reset_pwd;
```

```
        *status = error_status_ok;

        /* Get the lifetime for the server's Ticket-Granting-Ticket (TGT).  */
        /* Note that sec_login_get_expiration() returns a nonzero          */
        /*  status for an uncertified login context.  This is not          */
        /*  an error.  Hence the special error checking...                 */
        sec_login_get_expiration(login_context,
                &expiration,
                status);

        /* Get current time...                                             */
        time(&current_time);

        /* Now, if the expiration time is sooner than the desired "time    */
        /*  left"...                                                       */
        if (expiration < (current_time + time_left))
        {
                /* Refresh the server's authenticated identity...          */
                sec_login_refresh_identity(login_context,
                    status);

                /* Get key from local file...                              */
                sec_key_mgmt_get_key(rpc_c_authn_dce_secret,
                        keytab,
                        prin_name,
                        0,
                        (void**)&keydata,
                        status);

                /* Validate the login context...                           */
                sec_login_validate_identity(login_context,
                        keydata,
                        &reset_pwd,
                        &auth_src,
                        status);
        }

}
```

The server initialization code need then only make the following calls to establish server authentication and obtain valid credentials:

```
        /* Register server authentication information...                  */
        rpc_server_register_auth_info(server_principal_name,
                        rpc_c_authn_dce_secret,
                        NULL,
                        KEYTAB,
                        &status);

        /* Assume new identity...                                          */
        server_get_identity(server_principal_name,
                        &login_context,
                        (unsigned_char_p_t)KEYTAB,

&status);
```

Once the server has been running for a while, so that credentials may have expired, the server calls **server_renew_identity( )** before undertaking any task that requires valid credentials. For example, a server typically needs to call this operation before attempting to clean up its name space before shutting down.

Default Client Authentication Steps

Once a client has inherited or created a validated identity, the only step required is to call **rpc_binding_set_auth_info( )**. The client must supply a server principal name as an argument to this call.

Clients can inquire for the principal identity of a server by calling **rpc_mgmt_inq_server_princ_name( )**. If the client does not care about the principal identity of the server, the returned value can be supplied to **rpc_binding_set_auth_info( )** without further ado. If the client will only accept certain server identities, then it needs to check the returned value against the acceptable ones.

The list of acceptable values must be obtained and maintained by the client by some means of its own choosing: for example, a principal name could be obtained from an environment variable. The only security issue here is that the client must be sure that the list of acceptable values is a legitimate one. For example, it must not be stored in such a way that a false server can modify it.

The task of maintaining a list of acceptable principal names can be simplified somewhat by having all acceptable principals belong to a single group that is maintained by some trusted authority, such as a system administrator. The client then needs to maintain only the name of the group, rather than the whole list of principal names. To be sure that the server is authentic, the client need only check the principal name returned by **rpc_mgmt_inq_server_princ_name( )** against the group by calling **sec_rgy_pgo_is_member( )**.

The following code fragment demonstrates this scheme.

The is_valid_principal Routine

The **is_valid_principal( )** routine checks the group membership of the specified principal.

```
/******
 *
 *
 * is_valid_principal -- Find out whether the specified principal is a
 *                       member of the group he's supposed to be.
 *
 *
 ******/

boolean32 is_valid_principal(
unsigned_char_t *princ_name,          /* Full name of principal to test.   */
unsigned_char_t *group,               /* Group we want principal to bein.  */
unsigned32 *status)
{

        unsigned_char_t *local_name;       /* For principal's local name.   */
        char *cell_name;                   /* Local cell name.             */
        sec_rgy_handle_t rhandle;          /* Local registry binding.      */
        boolean32 is_valid;             /* To hold result of registry call. */

        fprintf(stdout, "sample_client: Initial principal name == %s",princ_name);
        fprintf(stdout, "sample_client: Initial group name     == %s",group);

        /* Find out the local cell name...                                 */
        dce_cf_get_cell_name(&cell_name,
status);
```

```
        /* Now bind to the local cell registry...                          */
        sec_rgy_site_open(cell_name, &rhandle, status);

        /* Free the cellname string space...                               */
        free(cell_name);

        /* Get the specified principal's local (cell-relative) name...     */
        local_name = malloc(strlen((char *)princ_name));

        sec_id_parse_name(rhandle,    /* Handle to the registry server.    */
                        princ_name, /* Global (full) name of the principal. */
                        NULL, /* Principal's home cell name returned here.  */
                        NULL, /* Pointer to UUID of above returned here.   */
                        local_name, /* Principal local name returned here.  */
                        NULL, /* Pointer to UUID of above returned here.   */
                        status);
        fprintf(stdout, "sample_client: Full principal name  == %s",princ_name);
        fprintf(stdout, "sample_client: Local principal name == %s",local_name);

        /* And finally, find out from the registry whether that principal   */
        /*  is a valid member of the specified group...                    */
        is_valid = sec_rgy_pgo_is_member(rhandle,
                sec_rgy_domain_group,
                group,
                local_name,
                status);

        /* Free the principal name string area...                          */
        free(local_name);
        return(is_valid);

}

    <. . . . .>


        /* Resolve the partial binding...                                  */
        rpc_ep_resolve_binding(binding_h,
                sample_v1_0_c_ifspec,
                &status);

        /* Find out what the server's principal name is...                 */
        rpc_mgmt_inq_server_princ_name(binding_h,
                rpc_c_authn_dce_secret,
                &server_princ_name,
                &status);
        /* And now find out if it's a valid member of our sample_servers   */
        /*  group...                                                       */
        if (is_valid_principal(server_princ_name, (unsigned_char_t*)SGROUP, &status))
        {
                rpc_binding_set_auth_info(binding_h,
                        server_princ_name,
                        rpc_c_protect_level_pkt_integ,
                        rpc_c_authn_dce_secret,
                        NULL,
                        rpc_c_authz_dce,
                        &status);

        }
```

Authorization

Assuming either that authentication has taken place and succeeded, or that no authentication has taken place, some server manager operation will then be invoked by the RPC runtime to handle an RPC call. This operation should, as its first duty, make an authorization decision.

A server manager operation calls **rpc_binding_inq_auth_client( )** to extract any authentication information for the calling client and then makes a series of decisions. The usual model is that the server establishes a set of access criteria and rejects the call if all criteria are not met. This is implemented as a series of tests, the server rejecting the call at the first failed test. The possible tests are as follows:

1. Does the client binding provide any authentication information? For this purpose, the application should check status after the call to **rpc_binding_inq_auth_client( )**. If no authentication information is provided (the *status* returned is **rpc_s_binding_has_no_auth**), the authorization function must decide whether this is acceptable. The authorization function may make its decision based on the unauthenticated ACL type, as noted later in this section.

   If authentication information is provided, then the application should go on to ask:
2. Is the authentication service acceptable to the server? The application checks the *authn_svc* parameter. Currently this check is redundant, since the only authentication service available is DCE secret key (the *authn_svc* returned is **rpc_c_authn_dce_secret** [2]).

   The server may of course, simply be satisfied that the client is authenticated and check no further. Or the server can do one or both of the following two things:
3. Check that the protection level is acceptable. This too is a matter for negotiation between the client and server applications, but it is important to begin by considering the runtime's mediation of the protection level request. Recall that the client specifies a specific protection level for a binding, whereas the server, when it registers its authentication information, specifies only the authentication service it will use.

   The chosen (agreed upon by the client and server) authentication service may not support all protection levels for all protocols. Therefore, the runtime adopts the policy of translating the client's protection level request to the next highest protection level actually supported by the authentication service and protocol in use. This means that the server application will see a protection level greater than or equal to the one requested by the client.

   Most server applications will establish a policy for the minimum acceptable protection level. In this case, if the level returned by the server application when it calls **rpc_binding_inq_auth_client( )** is below the standard, the server manager fails the access request. It is perfectly possible, however, for a server to require a *lower* level of protection. For example, a server may want to avoid the considerable overhead of full data encryption and thus refuse to service requests for this level.

2. There is considerable asymmetry in the use of the *authn_svc* values on the client call to **rpc_binding_set_auth_info( )** and the server call to **rpc_binding_inq_auth_client( )**. If the client specifies **rpc_c_authn_none**, the server sees a status of **rpc_s_binding_has_no_auth**, and no meaningful value is returned for the *authn_svc* parameter. Furthermore, given that the default authentication service is DCE secret key, if the client specifies **rpc_c_authn_default**, the server returns **rpc_c_authn_dce_secret** from *authn_svc*. In other words, while the client can specify three different values for *authn_svc*, the server can return only one.

4. Check that the authorization service is acceptable. Once again, this is a matter for negotiation between the client and server applications. The server application provides an access testing mechanism for authorization services it supports. There are three possibilities:
   - Authorization based on the client's principal name (**rpc_c_authz_name**).
   - Authorization based on the client's credentials (**rpc_c_authz_dce**). This involves checking the client identity's permission set (extracted from an ACL associated with the object the client is attempting to access) against the required permissions for the requested operation. The client's identity is extracted from its credentials, contained in its binding.
   - The server may permit access without authorization checking (**rpc_c_authz_none**).

   Name-based authorization is straightforward, but of very limited utility. In the simplest form, the application compares the extracted name string with a set of permitted names. However, the application is entirely responsible for maintaining and manipulating the set of permitted names securely, which is a nontrivial task. For example, the application must provide for some administrative way to update the set of permitted users. Typically, this will require maintainance of a restricted access file in some application-specific format. This is the kind of administrative overhead that applications should be designed to avoid.

   If the server application is willing to permit access by group and organization, it can somewhat offset this difficulty by making a group or organization membership check for the specified principal name. However, the basic objection remains that an application doing name-based authorization must maintain and administer a private security namespace (consisting of principals, groups, and organizations associated with access privileges). Since the credential-based (ACL) method is designed to provide a general solution to this problem, it is much to be preferred. ACL based access checking is described in the following sections.

   If the authorization service requested is acceptable, the server application makes the appropriate access tests as described in step 6.
5. Check that the server principal name specified by the client is acceptable. This check is useful for a server that is running with more than one principal identity. The server may only want to allow the operation under a specific principal identity. If the server is running with only one principal identity, this check is redundant.
6. Extract the client privileges and perform the appropriate access testing. The form of the client privileges depends on the authorization service. The application needs to extract the privileges in the correct format and pass them to the appropriate access tests.

Client Credentials

A client's credentials may be implicitly passed on to an ACL manager via a call to **dce_acl_is_client_authorized( )**. Or the credentials may be extracted from the client binding by a call to **rpc_binding_inq_auth_client( )** and then passed on to an ACL manager via a call to **sec_acl_mgr_is_authorized( )**. In the latter case, there is some additional complication in the case that the client specified no authentication. If the server supports credential-based authorization, it should handle this case by testing for unauthenticated access via the ACL manager. However, no credentials are returned from **rpc_binding_inq_auth_client( )** in this

case. The convention is to set the **pac** argument to NULL in this case ((**rpc_authz_handle_t**)**0**). ACL managers that follow the recommended policies will test for unauthenticated access in the case of such a null handle.

Null credentials are not the same thing as anonymous credentials. Anonymous credentials are simply credentials for the well-known anonymous user UUID. They are tested in the normal way by the ACL manager against permissions for the anonymous user in the relevant ACL.

The following code fragment shows the necessary steps:

```
        rpc_authz_handle_t pac;

        /* Get the client's credentials...                              */
        rpc_binding_inq_auth_client(. . . &pac . . &status);

        /* If there is no authentication information, set up a set of null  */
/*  credentials...                                                  */
        if (status == rpc_s_binding_has_no_auth)
        {
                pac = (rpc_authz_handle_t)0;
        }

        /* And now test the client's possession of the required permissions */
        /*  by passing its credentials (along with other pertinent data) to */
        /*  the following call...                                          */
        sec_acl_mgr_is_authorized(. . . (sec_id_pac_t*)pac . ..);
```

Access Control Lists

Authorization decisions depend on the following information:

privilege attributes

A set of principal and group names qualified by the cell name in which the principals and groups exist.

This information comes from the entity (client) that is attempting to perform the operation in question.

ACL privilege attribute entries

This is the ACL. It consists of a list of entries, each of which consists of an *entry type*, a *key*, and a *permissions set*, which taken together describe what permissions a particular entity possesses for the object to which the ACL is attached.

The ACL is looked up by the server through which the client is trying to perform the operation.

ACL mask entries

These consist of two *entry_type*: *permissions_set* pairs.

requested permissions

A permission set which describes the permissions that a client must possess in order to perform the requested operation. The server itself calculates this information.

There are two levels of semantics/policy to be considered here. One is the semantics of privilege attributes, for which we specify a strict (POSIX compliant) policy in the form of an access checking algorithm. This is embodied in the default access checking algorithm provided by the ACL library. The second is the semantics

of permissions. Ultimately these depend on the ACL manager and the kinds of objects it protects. However, some recommendations for keeping permissions as intuitive and consistent across applications as possible are offered in the following subsection.

Permissions Semantics Recommendations

The basic model used for access checking is to iterate through a sequence of ACL privilege attribute entries for each member of the requested permissions set, looking for the first match with a privilege attribute (and possibly ANDing the result with the appropriate ACL mask entries (*mask_obj* and *unauthenticated*). Entry types are checked in essentially the following order:

- [*user_obj*]
- *user*
- *foreign_user*
- [*group_obj*], *group*, *foreign_group*
- *other_obj*
- *foreign_other*
- *any_other*

In actual practice, the bracketed [*user_obj*] and [*group_obj*] entry types are ignored by the access checking algorithm implemented by the DCE ACL library. The reasons for this will be explained shortly. The access check is made at the first match, effectively giving precedence to the most specific match. The group entries are unordered so the match is made against the union of all group entries. This precedence allows explicit inclusion and exclusion of permissions depending on whether a more restrictive set of permissions is matched before or after a less restrictive set.

Except for the *user_obj* and *group_obj* entry types, the ACL entry types have semantics clearly defined according to the specificity and the cell of the principals referred to. In the local cell, *user* is the most specific, referring to some specific local principal. The *group* entry type refers to a specific set of principals. The *other_obj* type refers to other local principals not accounted for by *user* and *group* entries.

The *user* and *group* entries are extended to foreign cells by *foreign_user* and *foreign_group.* These are user and group identifiers that include a cell name. Strictly speaking, this distinction between the local and foreign cells is not required, since *user* and *group* entries implicitly contain global names (that is, the global name of the local cell is implicitly known.) The *user* and *group* entries are therefore really an implementation convenience for principals and groups in the local cell.

The *other_obj* entry is extended by *foreign_other*, which is a list of cell names.

Finally, principals that do not meet any of the above criteria can be authorized as *any_other*. The *other_obj*, *any_other*, and *foreign_other* types are distinguished by cells: *other_obj* applies to the local cell, *foreign_obj* applies to specified foreign cells, *any_other* applies to any cell.

The *user_obj* and *group_obj* types have less straightforward semantics. They refer to a special principal and group that must be known to the ACL manager "out of band": that is, they cannot be determined from the ACL entry itself. The semantics of the *mask_obj*, which is applied to everything except the *user_obj* and *other_obj*

entries, are also complicated. The *mask_obj* is implemented to permit POSIX ACLs to more or less maintain UNIX semantics for **000** permissions.

In general, the use of *user_obj* and *group_obj* is deprecated: they unnecessarily create a special user, thus complicating the otherwise straightforward semantics of ACLs. Unless you are implementing a file system, you probably do not need these types. (The *other_obj* type is unobjectionable since it has well defined semantics.) Similarly, the use of *mask_obj* is deprecated because of its awkward semantics.

Thus it is recommended that you use only types from the following subset of entry types:

- *user*
- *group*
- *other_object*
- *foreign_user*
- *foreign_group*
- *foreign_other*
- *any_other*

These types allow for the most specific to the most general principals, both for local, specific foreign cells, and for unspecified foreign cells.

The DCE ACL library ignores *user_obj* and *group_obj*, because there is no generic way to determine the user and group owners of an arbitrary ACL protected object: the semantics of ownership are application-specific. However, since these types are not recommended for general use anyway, their absence should not be a serious limitation for most applications that use the DCE ACL library.

ACL Managers

DCE entities expect to be able to access other DCE entities' objects' ACLs through a standard set of DCE routines, knowing nothing more than the names of the objects. The names of the objects are in the form of CDS pathnames.

The DCE ACL library is an implementation of the remote ACL (**rdacl**) interface, designed in such a way as to allow any DCE application to use it instead of having to implement the interface itself. In DCE 1.0, applications that wished to use the DCE ACL functionality had to implement the full remote interface themselves; in DCE 1.1 this is no longer true. Once an application has registered certain information with the ACL library (see "The Requirements" on page 70), its ACL management information will be hooked into the remote ACL implementation routines that make up the DCE ACL library.

Of course, an application still must take care of the details of storing and retrieving its ACLs (though these tasks are now made much easier by the DCE backing store library routines), setting up definitions that determine how its ACLs are interpreted, and so on. Practical examples of how to do these things can be found in the DCE sample application (fully reprinted in "Chapter 10. A Sample Application" on page 177), which is explained in the following sections.

For more detailed information about the interfaces mentioned below, see the *OSF DCE Application Development Guide—Core Components*.

Who Does What?

In a properly-setup application ACL manager, who does what? That is, what does the application code have to do about ACLs, and what is left up to the ACL library?

The DCE Security Service ACL API consists of the following routines:

- **sec_acl_bind( )**
- **sec_acl_bind_to_addr(**)
- **sec_acl_calc_mask( )**
- **sec_acl_get_access( )**
- **sec_acl_get_error_info( )**
- **sec_acl_get_manager_types( )**
- **sec_acl_get_mgr_types_semantics( )**
- **sec_acl_get_printstring( )**
- **sec_acl_lookup( )**
- **sec_acl_replace( )**
- **sec_acl_test_access( )**
- **sec_acl_test_access_on_behalf( )**

As their names suggest (full descriptions can be found in the *OSF DCE Application Development Reference*), these routines are what DCE clients call to use and manipulate ACLs, namely: bind to an object's ACL; retrieve an ACL; replace (that is, write to) an ACL; test (via its ACL) access to an object, and so on.

A properly-set-up DCE application does not have to implement any of these operations; they are all taken care of by the remote ACL implementations in the DCE ACL library. The only exception to this statement involves the binding operation. The application must register a routine that can be called by the ACL library whenever necessary to make up a complete binding to a specific ACL (this involves returning an ACL UUID, as will be seen below). This is the application's hook into the ACL library implementations: the registered routine will always be called during a binding operation on any of the application's ACLs, and once it has given the library a binding to the desired ACL, the library routines can perform any requested operation with it.

The application is thus not responsible for implementing any ACL interface operations. What the application is responsible for is the following:

- Setting up the necessary ACL data types and descriptions.
- Supplying a routine that resolves object names into ACL UUIDs.
- Setting up persistent databases in which the ACLs can be stored and retrieved.
- Initializing the ACLs for all existing objects.

The purpose of the following sections is to describe how these requirements can be fulfilled.

The Requirements

In order for a DCE application to use the ACL library routines for ACL management, the following things must be true of its server code:

- There must be a procedure that can take the valid name of an object and return that object's ACL UUID to a caller. This typically is accomplished by (first) looking up an object UUID in a name-indexed database and then (secondly) extracting

the ACL UUID from the object state information, which was looked up in a database indexed by object UUIDs. The databases, of course, must be set up and maintained by the application. Clients to bind to the objects through a CDS junction at the server's entry.

- The application's object name resolver has to be registered into the DCE runtime remote ACL (**rdacl**) interface mechanism, so that the DCE routines (such as **sec_acl_bind( )**) and the **acl_edit** command can access it.

  This is the server object name resolution procedure described in item 1. The **acl_edit** accepts a (CDS entry) name which it expects to be able to resolve into an object that has an ACL it can access. For this to happen, the application server must register a routine (with the **rdacl** interface UUID) which, when called by **acl_edit** with an object name, returns to **acl_edit** the information it needs (that is, a UUID) to get the ACL itself. In other words, the routine must be able to turn an object name into an ACL UUID.
- A persistent database in which to store the ACLs must be created. (The database must be compatible with the interface that the security routines use; that is it must be created with the DCE backing store library routines.)
- The ACL database must be registered (together with a manager type UUID and a name-to-ACL UUID resolver procedure) with the ACL library.
- An object type (that is, manager type) UUID must be created to identify each of the application's ACL categories (that is, the kind of object the ACL applies to, and hence the kind of ACL itself: what permissions it can contain, and what they mean in regard to the object they protect). (The manager type will also serve as an identifier for the ACL database that the ACLs themselves are stored in—this however is internal to the ACL library.)
- UUIDs to identify the objects must be created.
- The ACLs themselves on the relevant objects must be created.
- The ACLs must be stored, indexed by UUID, in the backing store database.

Setting up an ACL manager is a matter of making these eight things happen. The sample application shows the easiest way of accomplishing this, namely by using the DCE ACL library. See in particular the routine **server_acl_mgr_setup( )** in **sample_server.c**.

Note, by the way, that discussing the details of setting up an ACL manager without first considering the representation and management of the objects themselves is a very artificial thing to do. The excuse for doing it here is that ACL managers are the subject of this section. However, keep in mind that ACLs are only an adjunct to the objects they guard access to. In a real application one would never put the cart before the horse by working out the details of ACL management before settling on the way object management itself was to be done.

What is an Object?

Network operations are like grammatical sentences: they must have a *subject* (the client performing some operation), a *predicate* (the operation itself), and an object (the "thing" on which the operation is performed). Although meaningful sentences can sometimes omit some of its grammatical elements, a network operation must always have all three of its elements.

In any application, distributed or not, an object is any externally accessible resource which is under the application's control. Objects can be anything: printers, files, other machines, data—it all depends on the application. What these things have in common is that they must be accessed through the application itself. Entities in a

distributed application request the use of these resources, via clients, from the application server; and the server normally decides whether or not to grant use of a resource to an entity by examining the object's ACLs.

The object can have an existence quite independent of the application that manages it. On the other hand, the state information associated with the object, which the application must have access to in order to manage the object in a reasonable way, is maintained by the application and is useful only to it. This information is stored in a backing store database, where each separate record normally contains the state information for a single object. An object's ACLs qualify as *state information* for the purposes of this discussion.

In the sample application, the object's state information is practically identical to the objects themselves, since the latter seem not to exist at all except as the information stored in the backing databases. However, this is only partly true. The **sample_object** object is indeed a dummy and exists only as a pretext for showing how ACLs on objects are set up and manipulated. The server management object (**server_mgmt**), however, is different: it really has a purpose, although it is an abstraction (that is, access to an interface). It is used whenever a client attempts to execute a remote management operation on the server. In the sample application this happens when the client is invoked with the “kill” option.

Why Three Databases?

You might think that only one database would be required to hold the object state information described above. Why, then, are three backing store databases employed in the sample application? The answer to this question has two parts.

First: It is true that only one database is needed to hold the object state information itself. The need for a second database arises from the necessity of organizing the object information in more than one way, so that it can be retrieved both by name and by object UUID. The object information is stored directly in a database indexed by object UUIDs, and that is how it must be retrieved. However, application users will specify resources by names, not UUIDs. In order to make this work, the application stores its objects' UUIDs in a separate database indexed by their names. Thus any object's information can be retrieved, if the object's name is known, by means of a two-step process involving (first) looking up an object UUID from the name-indexed database, and (second) looking up the object information from the object UUID-indexed database.

Secondly: There is a third database to hold only the objects' ACLs. Theoretically speaking, there is no reason why the ACLs couldn't be held with the rest of the objects' information, in the object UUID-indexed database. However, the application's ACLs must be accessible to the DCE ACL library routines, and these routines expect a database, indexed by ACL UUIDs, containing only ACLs.

This allows us, for example, to call a DCE routine such as **dce_acl_is_client_authorized( )** (see the **sample_mgmt_auth( )** callback routine in **sample_server.c**), passing the ACL manager type UUID and the ACL UUID, and get back an answer to some query about permissions—the library routine is able to go into the database and access and read the ACL; we don't have to bother with that. It also allows the **rdacl** implementations in the ACL library to do the same thing, since they have a full ACL binding (which includes a handle to the database in which the ACL is stored).

Object Name Resolution Routine

Our application's name-to-ACL UUID resolution routine uses the following algorithm:

1. Take the object name that has been passed to it and use it to look up the UUID that identifies the object itself (in the name-indexed database).
2. Use the object UUID to retrieve the object information, which contains (among many other things) the UUID that identifies the object's ACL (in the object UUID-indexed database).
3. Use the retrieved ACL UUID to retrieve the ACL itself (from the ACL UUID-indexed database). If the manager types match, return the ACL UUID extracted in step 2 to the caller.

The caller is usually some routine in the ACL library. All it needs from the resolution routine is the ACL UUID; with this it can retrieve the ACL itself and proceed to do whatever needs to be done with (or to) it.

What is an ACL Manager?

A lot is said here and elsewhere about ACL managers, but you will not find in the sample application any specific routine or block of code with that name. So where exactly is our sample ACL manager? What does it consist of?

Conceptually, ACL manager is a way of referring comprehensively to the code and data present in an application to support ACLs. Practically speaking, the ACL manager in the sample application consists of all the places in the code where **dce_acl_is_client_authorized( )** is called to check a requestor's authorization. This is done in **sample_mgmt_auth( )** (in **sample_server.c**) and **sample_call( )** (in **sample_manager.c**).

Note that there are actually two ACL managers in the sample application. In **sample_call( )**, the client's access to the sample_object is being checked, and the ACL manager type UUID passed to the call is **sample_acl_mgr_uuid**. In **sample_mgmt_auth( )**, on the other hand, the client's access to the **server_mgmt** object is being checked, so the ACL manager type UUID passed there is **mgmt_acl_mgr_uuid**.

Why Two ACL Managers?

The application has two ACL managers because it uses two different kinds of object. This circumstance is a little obscured by the fact there are only two objects used in the application (in a real application, we might have expected many instances of **sample_object**, although there would still of course be only one **server_mgmt** object). Still, **sample_object** and **server_mgmt** are very different kinds of object, and having access to one means something quite different from having access to the other. **sample_object** is a dummy object with no independent meaning, but **server_mgmt** represents access to the server's remote management routines, which involves such things as being able to kill the server.

A practical sense of what this means can be had from looking at the two managers' ACL printstrings, near the top of the **sample_server.c** file. These strings, which contain text representations of the full range of permissions supported by the respective managers, show that there are many permissions that are unique to a single manager. For example, there is a **m_inq_if** permission (permission to execute the **rpc_mgmt_inq_if_ids( )** routine against the server). This permission makes sense only in the context of the **server_mgmt** object. A manager type identifies what set of permissions applies to a given set of objects.

How the ACL Library Routines Extract and Evaluate ACLs

One way of using ACLs to evaluate an entity's authorization to do something is by making a call to the DCE library routine **dce_acl_is_client_authorized( )**. For example, there are two places in the sample application where this is done to check client access to the application's own objects:

- In **sample_call( )** (in **sample_manager.c**)

  This is an interface operation, called by the client.
- In **sample_mgmt_auth( )** (in **sample_server.c**)

  This is the remote management callback function.

Similar routines are called remotely through the **sec_acl_ *( )** routines.

Evaluation takes the form of a call to the procedure, passing (among other things)

- The client (that is requestor's) binding
- The ACL manager type UUID
- The ACL UUID
- The desired permission set

The routine, given these parameters, is able to find and open the correct ACL database in which the ACL is held, extract the ACL, find the requestor's permission set (it determines who the requestor is from the credentials buried in the client binding), and compare it with the set of required permissions. If the latter can be found among the former, the routine will return a Yes answer; if not, it will return a No.

How does the library routine (especially when it is called, not from inside the application, as noted at the beginning of this section, but, say, by **acl_edit**) know how to access the correct ACL database from which to extract and examine the ACL identified by the ACL UUID? The answer is that the application's database will have become known to the caller in the course of establishing a binding to the server.

This is done by calling the application's registered resolver routine; the library finds the right resolver routine by calling all the resolvers that have been registered with it until it gets a successful return. It finds the ACL manager type in the same way, since it calls each attempted resolver passing the manager type UUID that was registered with it. See the **sample_resolve_by_name( )** function in the **sample_server.c** file.

Backing Store Database Items and Headers

Note that although backing stores are necessary in implementing an ACL manager, their use is not limited to ACL management. Backing stores are designed to be used for all kinds of persistent storage of distibuted data. For more information, see the *OSF DCE Application Development Guide—Core Components* .

As mentioned earlier, backing store databases are necessary for storing any information about the application's objects that must be preserved between application server sessions. The sample application uses three such databases, as described in "Object Name Resolution Routine" on page 73 .

From the point of view of the application that uses it, a database is characterized in the following two ways:

- How it is indexed
- What kind of data item (record) can be stored in it

The former is specified by a flag passed to **dce_db_open( )** when the database is first created; the latter is determined by the declarations you make in an **.idl** file.

An example of defining a backing store database item can be seen in the **sample_db.idl** and **sample_db.acf** files (note that the **dce/database.idl** file must be imported into the **.idl** file). A server stub and a header file is generated from these files when the application is compiled. The purpose of the **.idl** definitions is to establish the routine that will handle the transmission of the data items across the wire. Note that we don't implement the conversion routine; we just declare it in the **.idl** file: IDL itself does the rest, generating the necessary code in the client stub.

As has already been mentioned, the sample application uses three databases. The most complex of these is the object-indexed store (its handle is **db_object**). The other two, name-indexed (**db_name**) and ACL UUID-indexed (**db_acl**), are much simpler. Each of the three is briefly described in the following sections.

Object-Indexed Store

The sample application maintains objects whose data consists of a simple text string; however, the data type is also defined to contain a *standard header*. The standard header is a structure defined in **dce/database.idl**. Mostly it contains fields for a set of UUIDs that identify

- The object itself
- The owner of the object
- The owner's group
- The object's ACL
- The default object ACL
- The default container ACL

The standard header is a convenient means of keeping track of all the object's associated UUIDs, without having to define fields for them in one's own data structure. It is initialized by a call to the **dce_db_std_header_init( )** routine.

This is the only database whose data type is explicitly defined in the **.idl** file, because it's the only database whose data type contains an application-defined field (that is, **s_data**). The data type is also complex: that is, it contains both a header part and a data part. The other two databases have record types that contain only (simple) data, no headers.

Name-Indexed Store

The name-indexed store contains only object UUIDs, indexed by the object names that they are stored (and looked up) by. Note that there is no place where we actually declare the data type of this database; all we do is declare the conversion routine (**uu_convert( )**, in the IDL file). The database is created without a header (the default), so all it will hold is UUIDs.

If, for some reason, we did want to declare a header, then we would have to go through the steps of declaring a separate complex data type for the store in the **.idl** file, wherein would be declared the header type and the UUID type.

ACL UUID-Indexed Store

The ACL database contains only ACLs; its records have no headers. The records are indexed by ACL UUIDs. Here we do not even explicitly declare the conversion routine (**rdacl_convert**); it is generated by IDL (from a definition in **dce/dceacl.idl**). All we have to do is pass the routine's name to the **dce_db_open( )** call that opens this database.

Note that this is the database that the ACL library has to have access to; this access is set up by a call to **dce_acl_register_object_type( )**, which registers a manager type plus database plus resolver routine combination. The registration then allows the ACL library to derive any or all of these three things from an object name (the application's resolver routine has to help out in this, of course).

ACL Manager Coding Example

The following subsections contain extracts from the DCE sample application which is reprinted in full in "Chapter 10. A Sample Application" on page 177. The subsections below contain only the ACL manager code portions of the server application.

In order to save space and to improve the readability of the text, the code shown below has been slightly edited: all status checks, and all calls to the DCE serviceability interface (to print or log status or informational messages), have been removed.

Data Definitions

The following code consists of all ACL manager-related data and other definitions for the sample server application.

```
#define mgmt_perm_inq_if sec_acl_perm_unused_00000080
#define mgmt_perm_inq_pname sec_acl_perm_unused_00000100
#define mgmt_perm_inq_stats sec_acl_perm_unused_00000200
#define mgmt_perm_ping sec_acl_perm_unused_00000400
#define mgmt_perm_kill sec_acl_perm_unused_00000800

/* The constants below come from aclbase.h (aclbase.idl)...              */
#define OBJ_OWNER_PERMS sec_acl_perm_read | sec_acl_perm_write \
                                | sec_acl_perm_delete |
sec_acl_perm_test \
                                | sec_acl_perm_control |
sec_acl_perm_execute

#define ALL_MGMT_PERMS mgmt_perm_inq_if | mgmt_perm_inq_pname \
                                | mgmt_perm_inq_stats | mgmt_perm_ping \
                                | mgmt_perm_kill | sec_acl_perm_test \
                                | sec_acl_perm_control

/* These two UUIDs could be treated as "well known": that is
applications   */
/*   that use the same ACL manager for mgmt operations can use these...    */

uuid_t mgmt_acl_mgr_uuid = {/* 0060f928-bbf3-1d35-8d7d-0000c0d4de56       */
        0x0060f928, 0xbbf3, 0x1d35, 0x8d, 0x7d, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};

uuid_t mgmt_object_uuid = {/* 00573b0e-bcc2-1d35-a73e-0000c0d4de56        */
        0x00573b0e, 0xbcc2, 0x1d35, 0xa7, 0xe3, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};
```

```
/* These UUIDs are specific to this server...                            */
/* Some ACL UUIDs that will be globally used:                            */
uuid_t mgmt_acl_uuid;
uuid_t sample_acl_uuid;

/* The UUID of the sample ACL manager:                                   */
uuid_t sample_acl_mgr_uuid = { /* 001a15a9-3382-1d23-a16a-0000c0d4de56     */
        0x001a15a9, 0x3382, 0x1d23, 0xa1, 0x6a, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};
/* A UUID for a sample object:                                           */
uuid_t sample_object_uuid = {/* 00415371-f29a-1d3d-b8c8-0000c0d4de56      */
        0x00415371, 0xf29a, 0x1d3d, 0xb8, 0xc8, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};

/* The mgmt printstrings could be treated as standard for                */
/*   a standard mgmt ACL manager...                                      */
sec_acl_printstring_t mgmt_info = {"mgmt", "Management Interface"};

sec_acl_printstring_t mgmt_printstr[] = {
        { "i",        "m_inq_if",     mgmt_perm_inq_if      },
        { "n",        "m_inq_pname",  mgmt_perm_inq_pname   },
        { "s",        "m_inq_stats",  mgmt_perm_inq_stats   },
        { "p",        "m_ping",       mgmt_perm_ping        },
        { "k",        "m_kill",       mgmt_perm_kill        },
        { "c",        "control",      sec_acl_perm_control  },
        { "t",        "test",         sec_acl_perm_test     }
};

sec_acl_printstring_t sample_info = {"sample", "Sample RPC Program"};

sec_acl_printstring_t sample_printstr[] = {
        { "r",        "read",         sec_acl_perm_read     },
        { "w",        "write",        sec_acl_perm_write    },
        { "d",        "delete",       sec_acl_perm_delete   },
        { "c",        "control",      sec_acl_perm_control  },
        { "t",        "test",         sec_acl_perm_test     },
        { "x",        "execute",      sec_acl_perm_execute  }
};

/* These are the two entry point vectors that are explicitly initialized:   */
extern rdaclif_v1_0_epv_t dce_acl_v1_0_epv;
```

The server_get_local_principal_id Routine

The **server_get_local_principal_id( )** routine retrieves a principal's UUID from the local cell registry.

```
/******
 *
 * server_get_local_principal_id -- Get (from the local cell registry) the
 *                                  UUID corresponding to a principal name.
 *
 *
 *    Called from server_create_acl() and server_acl_mgr_setup().
 *
 ******/

void server_get_local_principal_id(
                unsigned_char_t *p_name,     /* Simple principal name.      */
                uuid_t *p_id,                /* UUID returned here.         */
                unsigned32 *status)          /* Status returned here.       */
{

        char *cell_name;                     /* For local cell name.        */
        sec_rgy_handle_t rhandle;            /* For registry server handle.  */
```

```
        /* First, get the local cell name...                              */
        dce_cf_get_cell_name(&cell_name, status);

        /* Now bind to the cell's registry...                             */
        sec_rgy_site_open(cell_name, &rhandle, status);

        /* Free the string space we got the cell name in...               */
        free(cell_name);

        /* Now get from the registry the UUID associated with the principal */
        /*  name we got in the first place...                            */
        sec_rgy_pgo_name_to_id(rhandle,
                        sec_rgy_domain_person,
                        p_name,
                        p_id,
                        status);

}
```

The server_create_acl Routine

The **server_create_acl( )** routine creates an ACL for a specified principal.

```
/******
 *
 * server_create_acl -- Create an ACL with some specified set of permissions
 *                      assigned to some principal user.
 *
 *
 *    Called from server_acl_mgr_setup().
 *
 ******/
void server_create_acl(
                uuid_t mgr_type_uuid,     /* Manager type of ACL to create. */
                sec_acl_permset_t perms,  /* Permission set for ACL.      */
                unsigned_char_t *user,    /* Principal name for new entry.  */
                sec_acl_t *acl,           /* To return the ACL entry in.   */
                uuid_t *acl_uuid,         /* To return the ACL's UUID in.   */
                unsigned32 *status)       /* To return status in.         */
{

        uuid_t u;  /* For the principal's UUID (from the registry).       */

        *status = error_status_ok;

        /* Create a UUID for the ACL...                                   */
        /* Note that the new UUID doesn't get associated with the entry in  */
        /*  this routine.  It must happen in server_acl_mgr_setup()...     */
        uuid_create(acl_uuid, status);

        /* Create an initial ACL object with default permissions for the   */
        /*  designated user principal identity...                         */
        dce_acl_obj_init(&mgr_type_uuid, acl, status);

        /* Get the specified principal's UUID...                          */
        server_get_local_principal_id(user, &u, status);

        /* Now add the user ACL entry to the ACL...                       */
        dce_acl_obj_add_user_entry(acl, perms, &u, status);

}
```

The server_store_acl Routine

The **server_store_acl( )** routine stores an ACL and its related information in the appropriate backing store databases.

```
/******
 *
 * server_store_acl -- Store ACL-related data.
 *
 *
 *      The data is stored in databases that support a
 *      name->object_uuid->acl_uuid style of ACL lookup.
 *
 *
 *    Called from server_acl_mgr_setup().
 *
 ******/

void server_store_acl(
                dce_db_handle_t db_acl,     /* ACL (UUID)-indexed store.   */
                dce_db_handle_t db_object,  /* Object (UUID)-indexed store. */
                dce_db_handle_t db_name,    /* Name-indexed store.         */
                sec_acl_t *acl,             /* The ACL itself.             */
                uuid_t *acl_uuid,           /* ACL UUID.                   */
                uuid_t *object_uuid,        /* Object UUID.                */
                unsigned_char_t *object_name, /* The name of the object.   */
                void *object_data,      /* The actual object data contents. */
                                        /* NOTE: NOT USED NOW.             */
                boolean32 is_container,  /* Are we storing a container ACL? */
                unsigned32 *status)         /* To return status.           */
{

        /* These two variables are used to hold UUIDs for the ACLs we will  */
        /*  need to create if we have a container ACL on our hands...      */
        uuid_t def_object, def_container;
        sample_data_t sample_data;

        *status = error_status_ok;

        /* Null the contents of the object_data variable...                */
        bzero(object_data, sizeof object_data);

        /* If we have a container ACL, then we have to create and store the */
        /*  special stuff associated with it-- namely, the container ACL   */
        /*  itself, and a default object ACL to go with it...              */
        if (is_container)
        {
                /* Create a UUID for the default object ACL...             */
                uuid_create(&def_object, status);

                /* Create a UUID for the default container ACL...          */
                uuid_create(&def_container, status);

                /* Store the default object ACL into UUID-indexed store...  */
                dce_db_store_by_uuid(db_acl, &def_object, acl,status);

                /* Store the default container ACL into UUID-indexed       */
                /*  store...                                               */
                dce_db_store_by_uuid(db_acl, &def_container, acl, status);

        }

        /* Store the plain object ACL into ACL UUID-indexed store...       */
        dce_db_store_by_uuid(db_acl, acl_uuid, acl, status);
        /* Store the ACL UUID(s) into a standard object header...          */
        dce_db_std_header_init(
                        db_object,      /* Object database.         */
                        &(sample_data.s_hdr), /* Object data hdr.  */
                        object_uuid,    /* Object UUID.             */
                        acl_uuid,       /* ACL UUID.                */
                        &def_object,    /* Default object ACL.     */
```

```
                        &def_container, /* Default container ACL.  */
                        0,              /* Reference count.         */
                        status);



      /* Now store the object data keyed by object UUID...              */
        if (strcmp(object_name, SAMPLE_OBJECT_NAME) == 0)
                strcpy(sample_data.s_data.message,
                       "THIS IS AN OFFICIAL SAMPLE OBJECT TEXT!");
        else if (strcmp(object_name, MGMT_OBJ_NAME) == 0)
                strcpy(sample_data.s_data.message,
                       "THIS IS AN OFFICIAL MGMT OBJECT SAMPLE TEXT!");
        else
                strcpy(sample_data.s_data.message,
                       "I DON'T KNOW WHAT THIS IS!");

       dce_db_store_by_uuid(db_object, object_uuid, (void*)&sample_data, status);

      /* Finally, store the object UUID keyed by the object("residual")  */
      /*  name...                                                         */
        dce_db_store_by_name(db_name, (char *)object_name, object_uuid, status);

}
```

The server_acl_mgr_setup Routine

The **server_acl_mgr_setup( )** routine performs all the steps necessary to set up ACL databases for the two object types used by the sample application.

```
/******
 *
 * server_acl_mgr_setup -- Open and, if necessary, create the ACL-related
 *                         databases, that is:
 *
 *        1.  Set up a default ACL manager for the management interface.
 *
 *        2.  Create an initial ACL.  For servers that dynamically create
 *           objects, this ACL is intended to be used as the ACL on the
 *           "container" in which objects are created.  If the server
 *           manages static objects, this ACL can be used for some other
 *           purpose.
 *
 *
 *    Called from main().
 *
 ******/

void server_acl_mgr_setup(
                unsigned_char_t *db_acl_path,    /* Pathname for databases. */
                dce_acl_resolve_func_t resolver, /* sample_resolve_by_name. */
                uuid_t acl_mgr_uuid,             /* ACL manager UUID.      */
                uuid_t object_uuid,              /* Object UUID.           */
                unsigned_char_t *object_name,    /* Object name.           */
                sec_acl_permset_t owner_perms,   /* Owner permission set.   */
                unsigned_char_t *owner,          /* Owner name.            */
                boolean32 is_container,      /* Is this a container object? */
                                             /* == TRUE from main().    */
        /* [out] parameters: */
                dce_db_handle_t *db_acl,    /* ACL-indexed store handle.   */
                dce_db_handle_t *db_object, /* Object-indexed store handle. */
                dce_db_handle_t *db_name,   /* Name-indexed store handle.   */
                uuid_t *object_acl_uuid,    /* Object ACL UUID.            */
                uuid_t *mgmt_acl_uuid,      /* Mgmt ACL UUID.              */
                unsigned32 *status)
{
```

```
        sec_acl_t new_acl;
        uuid_t machine_princ_id;
        unsigned_char_t machine_principal[MAXHOSTNAMELEN + 20];
        unsigned_char_t *uuid_string;
        boolean32 need_init;
        unsigned32 dbflags;
        static sample_data_t datahdr;
        unsigned_char_t *acl_path_string;
        sec_acl_permset_t   permset = (sec_acl_permset_t) 0;

        *status = error_status_ok;
        bzero(&datahdr, sizeof datahdr);

        uuid_create_nil(object_acl_uuid, status);

        need_init = 0;

        /* Build the full pathname string for the db_acl database...       */
        acl_path_string = malloc(MAX_ACL_PATH_SIZE);
        strcpy(acl_path_string, db_acl_path);
        strcat(acl_path_string, (unsigned_char_t *)"/");
        strncat(acl_path_string, "db_acl", strlen("db_acl"));

        /* If the thing doesn't exist yet, then we need to do some init-   */
        /*  ialization...                                                  */
        if (access((char *)acl_path_string, R_OK) != 0)
                if (errno == ENOENT)
                        need_init = 1;


/**********************************************************************/

        /* Create the indexed-by-UUID databases. There are two of these:    */
        /*       One for the ACL UUID-indexed store, and                   */
        /*       One for the Object UUID-indexed store...                  */

        dbflags = db_c_index_by_uuid;
        if (need_init)
                dbflags |= db_c_create;

        /* Open (or create) the "db_acl" ACL UUID-indexed backing store...  */
        dce_db_open(
                (char *)acl_path_string,  /* Filename of backing store.    */
                NULL,      /* Backing store "backend type" default == hash. */
                dbflags,   /* We already specified index by UUID for this.  */
                (dce_db_convert_func_t)dce_rdacl_convert,  /* Serialization */
                                /* function (generated by IDL).            */
                db_acl,    /* The returned backing store handle.           */
                status);

        /* Set the global variable that records whether we actually have   */
        /*  opened the databases; this enables us to avoid calling the     */
        /*  dce_db_close() routine for unopened databases, which will cause */
        /*  a core dump...                                               */
        databases_open = TRUE;

        /* For the object database, we need standard backing store headers  */
        dbflags |= db_c_std_header;
        if (need_init)
                dbflags |= db_c_create;
        /* Now open (or create) the "db_object" store...                   */
        /* Build the full pathname string for the database...              */
        free(acl_path_string);
        acl_path_string = malloc(MAX_ACL_PATH_SIZE);
        strcpy(acl_path_string, db_acl_path);
        strcat(acl_path_string, (unsigned_char_t *)"/");
        strncat(acl_path_string, "db_object", strlen("db_object"));
```

```
        dce_db_open(
                (char *)acl_path_string, /* Filename of backing store.     */
                NULL,      /* Backing store "backend type" default == hash. */
                dbflags,   /* Specifies index by UUID, and include standard */
                                /* headers.                                 */
                (dce_db_convert_func_t)sample_data_convert, /*Serializa-   */
                                /* tion function for object data.           */
                db_object,    /* The returned backing store handle.         */
                status);

        /* Create the indexed-by-name database...                           */
        dbflags = db_c_index_by_name;
        if (need_init)
                dbflags |= db_c_create;

        /* Build the full pathname string for the database...               */
        free(acl_path_string);
        acl_path_string = malloc(MAX_ACL_PATH_SIZE);
        strcpy(acl_path_string, db_acl_path);
        strcat(acl_path_string, (unsigned_char_t *)"/");
        strncat(acl_path_string, "db_name", strlen("db_name"));

        dce_db_open(
                (char *)acl_path_string,  /* Filename of backing store.    */
                NULL,      /* Backing store "backend type" default == hash. */
                dbflags,   /* Specifies index by name.                      */
                (dce_db_convert_func_t)uu_convert, /* Serialization func-   */
                                /* tion for name data.                      */
                db_name,   /* The returned backing store handle.            */
                status);

        free(acl_path_string);

/*********************************************************************/

        /* Now register our ACL manager's object types with the ACL         */
        /*  library...                                                      */

        /* Register for the mgmt ACL...                                     */
        dce_acl_register_object_type(
                *db_acl,  /* Backing store where ACLs are to be stored.    */
                &mgmt_acl_mgr_uuid,     /* Type of ACL manager: this one is */
                        /* for mgmt ACL operations; the UUID is defined    */
                        /* globally at the top of this file.               */
                        /* Why do we need this parameter? Well, the way    */
                        /*  that the ACL library keeps track of the differ- */
                        /*  ent "sets" of ACL databases is by manager UUID. */
                        /*  The manager UUID is what the library will use   */
                        /*  to figure out which ACL database to open and   */
                        /*  retrieve a requested ACL's contents from.      */
                        /*  Essentially what we are doing here is setting   */
                        /*  up things so that calls to the library routine  */
                        /*  dce_acl_is_client_authorized() can be made to   */
                        /*  check our ACLs, giving only the ACL UUID and a  */
                        /*  manager UUID to get the desired result.        */

                sizeof mgmt_printstr/sizeof mgmt_printstr[0], /* Number of  */
                                /* items in mgmt_printstr array.           */
                mgmt_printstr,  /* An array of sec_acl_printstring_t struc- */
                                /*  tures containing the printable repre-   */
                                /*  sentation of each specified permission. */
                &mgmt_info,     /* A single sec_acl_printstring_t contain-  */
                                /*  ing the name and short description for  */
                                /*  the given ACL manager.                 */
                sec_acl_perm_control,  /* Permission set needed to change   */
                                /* an ACL. Constants like these are defined */
```

```
                        /* in <dce/aclbase.h>.                      */
                sec_acl_perm_test, /* Permission set needed to test an ACL. */

                resolver,  /* Server function to get ACL UUID for a given   */
                                /*  object; for us it's the                  */
                                /*  sample_resolve_by_name() call, below.   */
                                /* This routine is for the use of acl_edit: */
                                /*  it allows acl_edit to receive an object */
                                /*  name and come up with the ACL UUID; at  */
                                /*  least that's what I think it's for.     */
                NULL,           /* Argument to pass to resolver function.   */
                0,              /* Flags -- none here.                      */
                status);
        /* Now register for the regular ACL...                              */
        dce_acl_register_object_type(
                *db_acl, /* Backing store where ACLs are to be stored.     */
                &sample_acl_mgr_uuid,  /* Hard-coded at the top of this     */
                                       /*  file.                            */
                sizeof sample_printstr/sizeof sample_printstr[0], /* Number */
                                /* of items in our printstring array.      */
                sample_printstr, /* An array of sec_acl_printstring_t      */
                                /* structures containing the printable rep- */
                                /* resentation of each specified permis-   */
                                /* sion set.                                */
                &sample_info,   /* A single sec_acl_printstring_t contain-  */
                                /*  ing the name and short description for  */
                                /*  the manager we're registering.         */
                sec_acl_perm_control, /* Permission set needed to change an */
                                /*  ACL.                                    */
                sec_acl_perm_test,    /* The permission you need to test an */
                                        /* ACL maintained by this manager.  */

                resolver,       /* Application server function that  gives   */
                                /*  the ACL UUID for a given object, when   */
                                /*  presented with that object's name; for  */
                                /*  us it's the sample_resolve_by_name()    */
                                /*  routine, below.                         */
                NULL,           /* Argument to pass to resolver routine;    */
                                /*  identified as the "resolver_arg" in the */
                                /*  code to that function below.            */
                0,              /* Flags -- none here.                      */
                status);

        /* If we're initializing, then we have to create all this stuff...  */
        if (need_init)
        {

                dce_svc_printf(NO_ACL_DBS_MSG);
                /* Create the mgmt interface ACL...                         */
                server_create_acl(
                        mgmt_acl_mgr_uuid, /* Create mgmt manager type ACL. */
                        ALL_MGMT_PERMS, /* Permission set for new ACL.     */
                        owner,          /* Principal name for new entry.   */
                        &new_acl,       /* This will contain the new ACL.   */
                        mgmt_acl_uuid,  /* This will contain the ACL UUID.  */
                        status);

/**************************************************************/
                /* For the management ACL we must add a default entry for   */
                /*  the machine principal so dced can manage the server.   */

              /* Construct the name entry string...                     */
              strcpy(machine_principal, "hosts/");
              gethostname((char *)(machine_principal + 6), MAXHOSTNAMELEN + 1);
              strcat(machine_principal, "/self");

                /* Get the machine principal's UUID...                      */
```

```
                server_get_local_principal_id(
                                              machine_principal,
                                              &machine_princ_id,
                                              status);

                /* Add a user entry for the machine principal to the new   */
                /*  ACL...                                                  */
                permset = ALL_MGMT_PERMS;
                dce_acl_obj_add_user_entry(
                                         &new_acl,
                                         permset,
                                         &machine_princ_id,
                                         status);

                /* By default everybody must be able to get the principal   */
                /*  name. They should be able to ping too. So add an appro- */
                /*  priate unauthenticated permissions entry to the ACL...  */
                permset = mgmt_perm_inq_pname | mgmt_perm_ping;
                dce_acl_obj_add_unauth_entry(
                        &new_acl,
                        permset,
                        status);

                /* Add permissions for the any_other entry in the ACL...   */
                permset = mgmt_perm_inq_pname | mgmt_perm_ping;
                dce_acl_obj_add_any_other_entry(
                        &new_acl,
                        permset,
                        status);
                /* Store the mgmt ACL...                                    */
                server_store_acl(
                        *db_acl,     /* The ACL UUID-indexed store.        */
                        *db_object,  /* The object UUID-indexed store.     */
                        *db_name,  /* The name ("residual")-indexed store.  */
                        &new_acl,          /* The ACL itself.              */
                        mgmt_acl_uuid,     /* The mgmt ACL UUID.           */
                        &mgmt_object_uuid,  /* The mgmt object UUID.       */
                        (unsigned_char_t *)MGMT_OBJ_NAME,  /* The mgmt ob-  */
                                               /*  ject name.              */
                        /* (void*) */ &datahdr, /* The data header = object */
                                               /*   contents.              */
                        0,                 /* Not a container ACL.        */
                        status);


/******************************************************************/
        /* Object ACL creation code...                                      */

                /* Now create the object ACL...                             */
                server_create_acl(
                        sample_acl_mgr_uuid,  /* Create an ACL with this   */
                                        /*  manager type.                  */
                        owner_perms,   /* Give it these permissions.       */
                        owner,         /* Make this the principal name.    */
                        &new_acl,      /* This will contain new ACL.      */
                        object_acl_uuid, /* This will contain new ACL UUID. */
                        status);

                /* Null the data header...                                  */
                bzero(&datahdr, sizeof datahdr);

                /* Store the object ACL...                                  */
                server_store_acl(
                        *db_acl,     /* The ACL UUID-indexed store.        */
                        *db_object,  /* The object UUID-indexed store.     */
                        *db_name,  /* The name ("residual")-indexed store.  */
                        &new_acl,          /* The ACL itself.              */
```

```
                object_acl_uuid,    /* The object ACL UUID.        */
                &object_uuid,       /* The object UUID.            */
                object_name,        /* The object name.            */
                /* (void*) */ &datahdr, /* The data header = object */
                                        /*  contents.              */
                /* is_container */  0,       /* Is this a container */
                                        /*  ACL?                   */
                status);

        /* Finally, free the space we were using...                */
        dce_acl_obj_free_entries(&new_acl, status);
    /* ...end of object ACL creation code.                         */

/*********************************************************************/
    }
    else /* ACL databases already exist; get the two ACL UUIDs...      */
    {

        /* This is a call to sample_resolve_by_name() (see below);  */
        /*  it gives us the UUID of the ACL of the object whose    */
        /*  name we pass it...                                     */
        (*resolver)(
                NULL,         /* No client bind handle; local call. */
                object_name,  /* Object whose ACL UUID we want.    */
                0,            /* Type of ACL we want UUID of.      */
                &sample_acl_mgr_uuid, /* Object's manager type.     */
                0,            /* Ignored as far as we'reconcerned. */
                NULL,         /* "resolver_arg"; unused.           */
                object_acl_uuid, /* Will contain object ACL UUID.   */
                status);

        (*resolver)(
                NULL,         /* No client bind handle; localcall. */
                (sec_acl_component_name_t)MGMT_OBJ_NAME, /* We want */
                                        /* mgmt object's ACL UUID.  */
                0,             /* Type of ACL we want UUID of.     */
                &mgmt_acl_mgr_uuid, /* Object's manager type=mgmt.  */
                0,            /* Ignored as far as we're concerned. */
                NULL,         /* "resolver_arg"; ignored.          */
                mgmt_acl_uuid, /* Will contain mgmt ACL UUID.      */
                status);

      }

    /* Set up remote management authorization to use the ACL manager.   */
    /*  Note that the first parameter to this call is the address of a  */
    /*  management authorization callback routine, which is defined    */
    /*  later in this file...                                          */
    rpc_mgmt_set_authorization_fn(sample_mgmt_auth, status);

    /* Finally, register the rdacl interface with the runtime...       */
    rpc_server_register_if(
                rdaclif_v1_0_s_ifspec, /* Interface to register.   */
                NULL,                  /* Manager type UUID.       */
                (rpc_mgr_epv_t) &dce_acl_v1_0_epv, /* Entry point   */
                                                /* vector.         */
                status);

}
```

The server_acl_mgr_close Routine

The **server_acl_mgr_close( )** routine closes the ACL databases.

```
/******
 *
```

```
 * server_acl_mgr_close --  Called at cleanup time to close
 *                          the three ACL databases.
 *
 *
 *    Called from main().
 *
 ******/

void server_acl_mgr_close(
dce_db_handle_t *db_acl,          /* ACL UUID-indexed database.            */
dce_db_handle_t *db_object,       /* Object UUID-indexed database.         */
dce_db_handle_t *db_name,         /* Name-indexed database.                */
unsigned32 *status)
{

        *status = error_status_ok;

        /* Close the ACL UUID-indexed database...                           */
        dce_db_close(db_acl, status);

        /* Close the Object UUID-indexed database...                        */
        dce_db_close(db_object, status);

        /* Close the name-indexed database...                               */
        dce_db_close(db_name, status);

}
```

The server_rdacl_export Routine

The **server_rdacl_export( )** routine registers the remote ACL interface in the local endpoint map.

```
/******
 *
 * server_rdacl_export -- Make the rdacl interface available
 *                        for ACL editors.
 *
 *
 *     Note that we don't export to the namespace.  Instead, the ACL editor
 *      will typically bind to the server via some other entry that holds
 *      the application-specific interface bindings.  This must hold at least
 *      one object UUID, and the same UUID must be put into the endpoint map
 *      too.  If not, ACL editors will have no way to distinguish the end-
 *      points of this server from those of other servers on the same host
 *      that also export the rdacl interface.
 *
 *    Called from main().
 *
 ******/

void server_rdacl_export(
rpc_binding_vector_t *binding_vector, /* Binding handles from RPC runtime.  */
uuid_vector_t *object_uuid_vector,    /* Server instance UUID(s).          */
unsigned32 *status)
{

        uuid_vector_t my_uuids;

        *status = error_status_ok;

        /* Register the server's endpoints with the rdacl interface at the  */
        /*  local endpoint map...                                           */
        rpc_ep_register(rdaclif_v1_0_s_ifspec,
                binding_vector,    /* Our binding handles from RPC runtime. */
                object_uuid_vector,  /* Server instance UUID (only one).   */
```

```
                (unsigned_char_p_t) "rdacl interface", /* Annotation.      */
                status);

}
```

The server_rdacl_cleanup Routine

The **server_rdacl_cleanup( )** routine removes the remote ACL interface information from the local endpoint map.

```
/******
 *
 * server_rdacl_cleanup -- Called at cleanup time to
 *                         unregister the rdacl interface.
 *
 *
 *    Called from main().
 *
 ******/

void server_rdacl_cleanup(
rpc_binding_vector_t *binding_vector,  /* Binding handles from RPC runtime. */
uuid_vector_t *object_uuid_vector,     /* Server instance UUID(s).         */
unsigned32 *status)
{

        *status = error_status_ok;

        rpc_ep_unregister(rdaclif_v1_0_s_ifspec,
                binding_vector,
                object_uuid_vector,
                status);

}
```

The sample_mgmt_auth Routine

The **sample_mgmt_auth( )** routine assesses the authorization of any client attempting to execute a remote management operation on the sample application server.

```
/******
 *
 * sample_mgmt_auth -- Management authorization callback function.
 *
 *    This is the routine that is implicitly called to test authorization
 *     whenever someone tries to use the mgmt interface to tinker with us
 *     or our ACLs.
 *
 *    The callback is set up by a call to rpc_mgmt_set_authorization() in
 *    server_acl_mgr_setup().
 *
 ******/
boolean32 sample_mgmt_auth(
rpc_binding_handle_t client_binding, /* Client's binding, whoever he is.    */
unsigned32 requested_mgmt_operation, /* What client is attempting to do.   */
unsigned32 *status)
{
        boolean32 authorized = 0;
        sec_acl_permset_t perm_required;
        unsigned_char_t *uuid_string;

        *status = error_status_ok;

        /* Discover what permission is required in order to do what the    */
```

```
        /*  client is trying to do...                                     */
        switch (requested_mgmt_operation)
        {
                case rpc_c_mgmt_inq_if_ids:
                        perm_required = mgmt_perm_inq_if;
                        break;
                case rpc_c_mgmt_inq_princ_name:
                        perm_required = mgmt_perm_inq_pname;
                        break;
                case rpc_c_mgmt_inq_stats:
                        perm_required = mgmt_perm_inq_stats;
                        break;
                case rpc_c_mgmt_is_server_listen:
                        perm_required = mgmt_perm_ping;
                        break;
                case rpc_c_mgmt_stop_server_listen:
                        perm_required = mgmt_perm_kill;
                        break;
                default:
                /* This should never happen, but just in case...           */
                        return(0);
        }

        /* Okay, now check whether the client is authorized or not...      */
        dce_acl_is_client_authorized(
                client_binding,          /* Client's binding handle.       */
                &mgmt_acl_mgr_uuid,      /* ACL manager type UUID.         */
                &mgmt_acl_uuid,          /* The ACL UUID.                  */
                NULL,                   /*         Pointer to owner's UUID. */
                NULL,                   /* Pointer to owner's group's UUID. */
                perm_required,          /* The desired privileges.         */
                &authorized,            /* Will be TRUE or FALSE on return. */
                status);

        /* Return the result to the caller...                              */
        return(authorized);
}
```

The sample_resolve_by_name Routine

The **sample_resolve_by_name( )** routine derives the ACL UUID of an object from its name.

```
/******
 *
 * sample_resolve_by_name -- take the name of an object, and return the
 *                           UUID of the object's ACL.
 *
 * The address of this function is passed (via the call to
 * server_acl_mgr_setup()) to the dce_acl_register_object_type() call. So
 * it gets implicitly called anytime someone tries to retrieve the ACL of
 * an object managed by the ACL manager we've set up.
 *
 *
 * Basically, the most a server needs is one resolve-by-name routine and
 * one resolve-by-UUID routine; the former gets you the desired object's
 * UUID; and the latter then will get you the object data itself (the way
 * this works can be seen in the body of this routine below).  In most
 * cases, these routines will share the same name and UUID databases; if
 * they don't, the resolver_arg can be used to point to the correct other
 * database.  Typically, the only difference between the managers is that
 * they use different print strings.
 *
 *
 * For the official statement of the signature of a dce_acl_resolve_func_t,
 * see the dce_acl_resolve_by_uuid() reference page; that routine has the same
```

```
 * type.
 *
 *
 *******/

dce_acl_resolve_func_t sample_resolve_by_name(
handle_t h,                     /* Client binding handle passed into the    */
                                /*  server stub. sec_acl_bind() is used to  */
                                /*  create this handle.                     */
sec_acl_component_name_t name,  /* The object whose ACL's UUID we want.    */
sec_acl_type_t sec_acl_type,    /* The type of ACL whose UUID we want.     */
uuid_t *manager_type,           /* The object's manager type.              */
                                /* NOTE that this parameter isn't used be-  */
                                /*  low.                                    */
boolean32 writing,              /* "This parameter is ignored in OSF'sim-  */
                                /*  plementation" (from the reference page  */
                                /*  for dce_acl_resolve_by_uuid()).         */
void *resolver_arg,             /* This is the app-defined argument passed  */
                                /*  to dce_acl_register_object_type(); it   */
                                /*  should be a handle for a backing store  */
                                /*  indexed by UUID. Note that it isn't     */
                                /*  used here though.                       */
uuid_t *acl_uuid,               /* To return ACL's UUID in.                */
error_status_t *st              /* To return status in.                    */
)
{
        uuid_t u, *up; /* To hold the retrieved object UUID, and to         */
                       /*  take a pointer to it.                            */
        unsigned_char_t *uuid_string;
        sec_acl_t retrieved_acl;

        /* The definition of the following is in the sample.idl file.      */
        /*                                                                  */
        /* See the "Examples" section in the dce_db_open() ref page,        */
        /*  where the skeleton IDL interface for a server's backing         */
        /*  store is given.  The data type definition (which is what        */
        /*  sample_data_t is) is there prescribed as consisting of a        */
        /*  dce_db_header_t, plus whatever server-specific data is          */
        /*  quired, all in a single structure.                              */
        /*                                                                  */
        /*  Essentially it's a dce_db_header_t structure (with an           */
        /*  application-defined message string tacked on); this is          */
        /*  the object header data structure that is returned, such as,     */
        /*  by dce_db_header_fetch(); in other words, this is the           */
        /*  thingie that actually contains the data "in" an object          */
        /*  held in an object store.                                        */
        sample_data_t dataheader;

        *st = error_status_ok;

        /* Check for nonexistence of object name...                         */
        if (!name || !*name)
        {
                dce_svc_printf(CANNOT_RESOLVE_NAME_MSG);
                return;
        }

        /* Get the object's UUID, which will be the key that we will use to */
        /*  fetch this particular object's data in the call following this  */
        /*  one...                                                          */
        dce_db_fetch_by_name(db_name, (char *)name, /* (void *) */ &u, st);

        up = &u;     /* ...take the pointer to the key.                     */
        /* Using the UUID "key" that we just retrieved, get the dataheader  */
        /*  for the desired object (note that the data that one retrieves   */
        /*  with this routine can be anything; it depends on what we are    */
        /*  using the backing store for)...                                 */
```

```
        dce_db_fetch_by_uuid(db_object, up, /* (void *) */ &dataheader, st);

        /* Now, depending on the kind of ACL we're hunting for (that is ob- */
        /*  ject, container, etc.), extract its UUID from the object's     */
        /*  header structure...                                             */
        switch (sec_acl_type)
        {
                case 1:
                        *acl_uuid = dataheader.s_hdr.tagged_union.h.def_object_acl;
                        break;
                case 2:
                        *acl_uuid = dataheader.s_hdr.tagged_union.h.def_container_acl;
                        break;
                default:
                        *acl_uuid = dataheader.s_hdr.tagged_union.h.acl_uuid;
        }

        /* Here it might be interesting to try retrieving the ACL itself,   */
        /*  and e.g seeing what its manager type is...                     */
        dce_db_fetch_by_uuid(db_acl,
                        acl_uuid,
                        &retrieved_acl,
                        st);

        /* We are handling two ACL managers through this function, so we     */
        /*  have to make sure that we've extracted from the single ACL       */
        /*  database the correct ACL: that is, one whose manager type UUID is */
        /*  identical to the manager_type parameter we were passed: this is   */
        /*  the manager whose ACL the runtime is trying to bind to. So...    */
        if ((manager_type != NULL) &&
            (!uuid_equal(manager_type, &(retrieved_acl.sec_acl_manager_type), st)))
        {
                /* Return a bad status...                                     */
                *st = acl_s_bad_manager_type;
                /* And no ACL UUID...                                         */
                acl_uuid = NULL;
                return(0);
        }

}
```

Chapter 4. Binding

Binding is the process by which an RPC client establishes a relationship with a server that supports an interface, object, or some other resource the client is interested in. Since clients operate on server-held resources by making RPCs, you can think of binding, specifically, as creating the state required for an RPC to be made. In practice, the work of binding clients to servers normally involves name and endpoint mapping services. Strictly speaking, however, neither of these services is required for binding, since well-known bindings and endpoints can be used (in the form of string bindings). This chapter discusses the underlying binding model, apart from the use of name and endpoint services. It forms an essential introduction for the discussion of name and endpoint services that follows in “Chapter 5. Using the DCE Name Service” on page 103 .

The Binding Model

Binding refers to the establishment of a relationship between a client and a server that permits the client to make a remote procedure call to the server. The term binding usually refers specifically to a protocol relationship between a client and either the server host or a specific endpoint on the server host, and *binding information* means the set of protocol and addressing information required to establish such a binding. But, for a remote procedure call, such a binding occurs in a context that involves other important elements, paralleling the notion of a binding in a local procedure call. In order for an RPC to occur, a relationship must be established that ties a specific procedure call on the client side with the manager code that it invokes on the server side. This requires both the binding information itself and a number of additional elements (see Figure 2 on page 92 . The complete list is as follows:

- A protocol sequence that identifies the RPC and underlying transport protocols
- An RPC protocol version identifier
- A transfer syntax identifier
- A server host network address
- An endpoint of a server instance on the host
- An object UUID that can optionally be used for selection among servers and/or manager routines
- An interface UUID that identifies the interface to which the called routine belongs
- An interface version number that defines compatibility between interface versions
- An operation number that identifies a specific operation within the interface

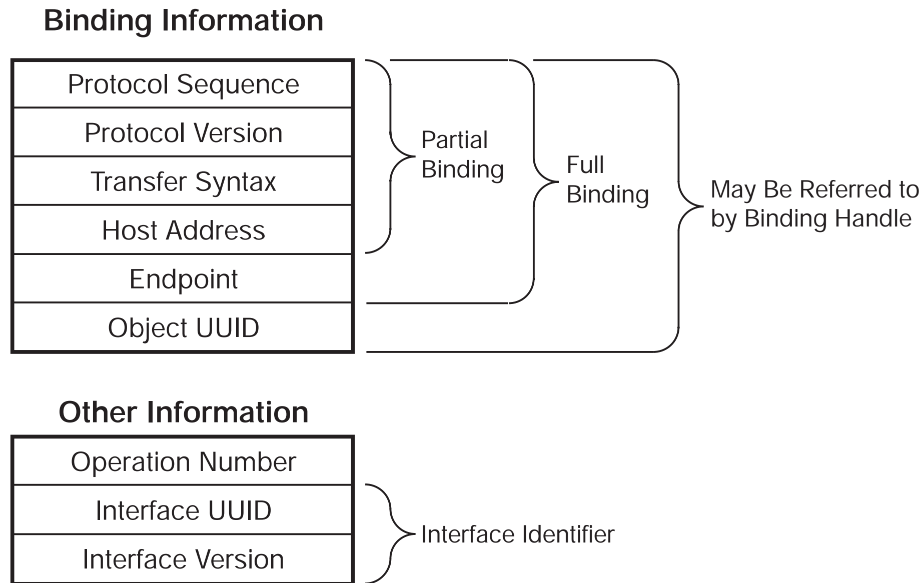


Figure 2. Information Required to Complete an RPC

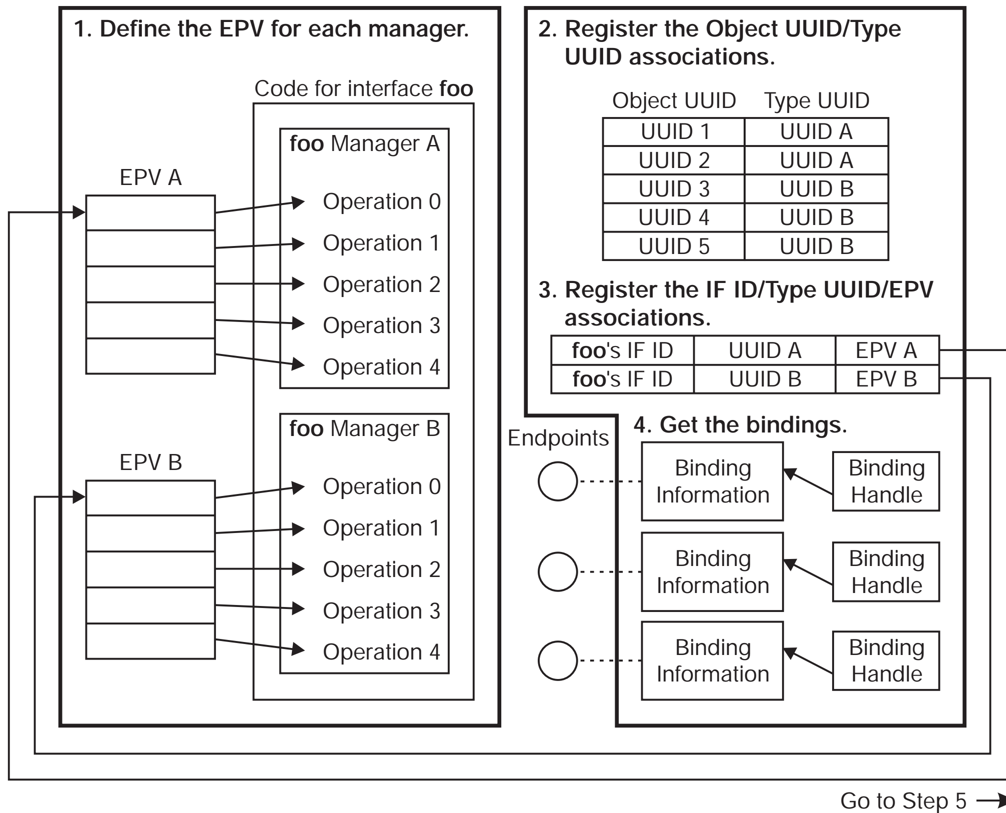
The binding information itself covers the first five elements of the list—the protocol and address information required for RPC communications to occur between a client and server. (“Chapter 5. Using the DCE Name Service” on page 103 also shows the object UUID as part of the binding information. This applies to clients, as explained in “Client Binding Model” on page 95 .) In RPC terminology, such a binding can be partial or full. A partial binding is one that contains the first four elements of the list, but lacks an endpoint. A *full binding* contains an endpoint as well. The distinction is that a partial binding is sufficient to establish communications between a client and a server host, whereas a full binding allows communications to a specific endpoint on the server host.

In order to complete an RPC call, all of the elements listed in Figure 2 must be present. The binding process consists of a series of steps taken by the client and server to create, make available, and assemble all the necessary information, followed by the actual RPC, which creates the final binding and routing using the elements established by the previous steps.

Server Binding Model

Figure 3 on page 94 shows the set of relationships that a server must establish to receive remote procedure calls. As Figure 3 on page 94 indicates, these are maintained in several places:

- By the server runtime
- In the stub and application code
- By the endpoint mapper
- By a name service


Stub and Application Code
Maintained by Server Runtime
1. Define the EPV for each manager.
Code for interface foo
foo Manager A
EPV A
Operation 0
Operation 1
Operation 2
Operation 3
Operation 4
foo Manager B
EPV B
Operation 0
Operation 1
Operation 2
Operation 3
Operation 4
2. Register the Object UUID/Type UUID associations.
Object UUID
Type UUID
UUID 1
UUID A
UUID 2
UUID A
UUID 3
UUID B
UUID 4
UUID B
UUID 5
UUID B
3. Register the IF ID/Type UUID/EPV associations.
foo's IF ID
UUID A
EPV A
foo's IF ID
UUID B
EPV B
4. Get the bindings.
Endpoints
Binding Information
Binding Handle
Binding Information
Binding Handle
Binding Information
Binding Handle
Go to Step 5

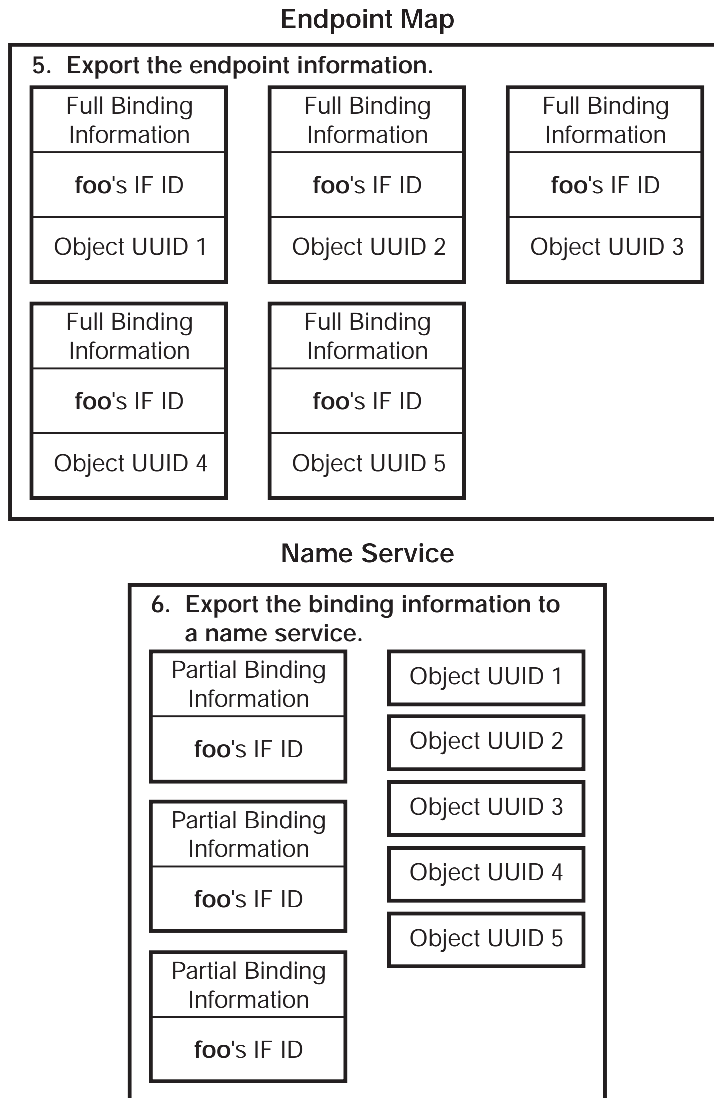


Figure 3. Server Binding Relationships

The steps that server applications take to establish these mappings are not discussed here since they are fully documented in the *OSF DCE Application Development Guide—Core Components*. Once established, this set of relationships allows the server runtime to construct a complete binding, with routing to a specific server operation, for a call that contains the following information:

- Full or partial binding information
- An interface identifier
- An object UUID, which may be nil
- An operation number

Note that the server runtime itself maintains only a very limited set of relationships: interface identifier/type UUID/manager EPV and object UUIDs/type UUIDs. It is especially worth noting that the runtime maintains no relationships between the protocol-address bindings it has created and any of the other information. The server merely advertises the relationships it wants to export in a name service and registers them in the endpoint map. Bindings are exported and registered along with interface identifiers and, possibly object UUIDs.

When the exported and registered information is used by clients to find the server, client calls arriving at the server endpoints should contain interface identifier/object UUID pairs that the server can, in fact, service, although the RPC mechanism itself can provide no guarantee of this. This means that name service and endpoint map operations, while they are not, strictly speaking, a required part of an RPC call, usually play an important role in constructing bindings.

The indirect mapping from object UUID to type UUID to EPV (and hence to the manager called) also gives the server some flexibility in organizing its resources based on object UUIDs. This is explained in “Call Routing” .

Client Binding Model

To make a call, the client needs a compatible binding: that is, one that offers the interface and version desired, uses a mutually supported protocol sequence, and if requested, is associated with a specific object UUID.

Clients typically find compatible bindings by making calls to RPC API routines that search the name service. Typically, the client specifies the interface and object UUIDs desired, and the runtime takes responsibility for finding bindings with protocol sequences that it can use.

For each binding that the client imports, the runtime provides a server binding handle that refers to the binding information maintained by the client runtime. This includes the protocol sequence and address information for the server host and possibly includes an object UUID.

Once the client has found a compatible binding, it makes a call using the binding handle for that binding. When the call is made, the client runtime has available to it the binding information and any object UUID referred to by the binding handle. Also available in the stub code are the interface identifier of the interface on which the call was made, and the operation number of the routine being called. Recall that the last three items of this information—the object UUID/interface identifier/operation number—are precisely what the server needs to route the call to a specific manager operation.

Call Routing

Once the server and client have taken all the necessary steps to set up server and client side relationships, the call mechanism can use them to construct a complete binding and call routing when the call is made. When the client makes a call with a binding that lacks an endpoint (typically the case for bindings imported from the name service), the endpoint is acquired from the endpoint mapper on the target host. The endpoint mapper finds a suitable endpoint by searching the local endpoint map for a binding that provides the requested interface UUID, and if requested, object UUID.

The endpoint map interface and protocol information must match in order for an endpoint to be found, but an object UUID match may not be required. A server can provide a default UUID match by registering the nil UUID. Calls with a nil or unmatched object UUID will get the default endpoint.

Once an endpoint is selected, a call can be routed to one of the endpoints being used by a compatible server instance. The server can unambiguously select the correct interface and operation by using the interface identifier and operation

number contained in the call. Recall, however, that the RPC mechanism makes it possible for a server to implement multiple managers for an interface. Hence it may be necessary to select the correct manager. Manager selection is based on the object UUID contained in the call. The selection mechanism depends on two of the relationships established by the server: the object UUID/type UUID mapping and the interface ID/type UUID/manager UUID mapping.

For routing, the server provides a default path by registering a default manager for the nil type UUID. Calls containing the nil object UUID, or any UUID for which the server has not set another type UUID, will be directed to the default manager.

Once the manager is selected, the call is dispatched via the selected manager EPV using the operation number contained in the call.

Routing Policy

There are many ways in which clients and servers can arrange the details of binding among themselves, including: how bindings are exported and imported, whether object UUIDs are used, and how object-type mappings are established. High-level resource policy issues relating to the name service and endpoint mapper are discussed in “Chapter 5. Using the DCE Name Service” on page 103 . In the present chapter, some of the lower-level routing policy questions that arise from the binding model itself will be discussed. These, in fact, have a substantial impact on how the namespace is used by applications.

The most important issues concern the role of UUIDs in the binding model. Interface identifiers, which consist of a UUID and version number, have a well-defined and unambiguous role. But object UUIDs are somewhat overloaded by the binding model. An object UUID may be used to select bindings from the name service, to select endpoints from the endpoint mapper, and to map a call to the correct manager type within the server. Furthermore, a server may use object UUIDs in some application-specific way to identify and manipulate the objects it manages.

There is great potential for conflict between the use of object UUIDs to select bindings and endpoints and their use to identify objects and routes to manager types. This conflict is particularly evident in the case of servers that provide so-called ubiquitous interfaces, such as the **rdacl** interface. Because many servers on a host are likely to export such an interface, it is essential to have an object UUID to identify the correct endpoint in the endpoint map. Without an object UUID, the endpoint mapper can only return the endpoint of some server that exports the requested interface, very likely the wrong one.

An alternative strategy does exist: a client can call **rpc_ep_resolve_binding( )** using a nonubiquitous interface that it knows the server of interest does export. The call to the ubiquitous interface can then be made with the resolved binding. Clients often use this technique to call the remote **pc_mgmt_ *** routines. Nevertheless, the objection remains that it is still impossible to select among endpoints of servers or server instances that export the same nonubiquitous interface.

The most straightforward solution is for a server to export a UUID to the namespace where it functions as an unambiguous tag for the servers’ endpoints. Clients can find this UUID either by importing it from a named entry or it may be made well-known, effectively becoming a stable, well-known tag for the server’s volatile endpoints. When endpoint UUIDs are well-known, they become useful for

finding servers even when the client is interested in a nonubiquitous interface. Exactly how servers export and clients find these UUIDs depends on the resource model adopted, as discussed in “Chapter 5. Using the DCE Name Service” on page 103 .

This obvious use of UUIDs as endpoint identifiers, however, potentially conflicts with their use as object identifiers. According to the RPC binding model, when clients import bindings based on object UUIDs, these UUIDs are incorporated into call bindings where they may used for endpoint selection, for manager selection, and possibly for some application-specific purpose. If an application exports its object UUIDs to the namespace, then they are used both to identify objects and to identify endpoints. This means that, at a minimum, a server would need to maintain a potentially large number of mappings to the same endpoints.

Moreover, especially when servers manage many objects or create them dynamically, clients will typically know objects by names rather than by UUIDs. Servers can provide such mappings via the namespace itself by exporting each object UUID to a different namespace entry, but this even further complicates the server’s job of maintaining its exports and mappings.

The obvious solution to these problems is to have servers maintain their object UUIDs and name-to-object UUID mappings internally. The basic RPC binding mechanism does not provide much support for this approach: there is no generic way for servers to make objects or names available to clients except through the name service. Also, a UUID used to identify a server endpoint is probably useless for call routing to a manager type within a server. However, the higher-level object management interfaces discussed in “Chapter 5. Using the DCE Name Service” on page 103 provide this functionality.

This leads to two important recommendations:

- Servers should export to the namespace at least one UUID as a tag for its endpoints, and should register the UUID with the endpoint map.
- Servers which support multiple objects should also support the object management interface(s) discussed in “Chapter 5. Using the DCE Name Service” on page 103, instead of exporting multiple object UUIDs to the namespace.

Binding Handles

Binding handles, although they appear as parameters of RPCs, are in fact purely local to the server or client applications that use them. A binding handle is simply a reference to binding information that is cached by the local runtime. The runtime uses this binding information to construct its side of a client-server association. Even when a binding handle appears as an explicit parameter of an RPC, it is not marshalled or unmarshalled as call data in the same way as other call parameters.

On the client side, a binding handle parameter simply permits an application to indicate explicitly to the runtime which cached binding should be used for the call.

On the server side, a binding handle parameter provides a manager operation with a reference to cached binding information for the calling client so that the manager can, for example, extract authorization information about the client.

In calls to ubiquitous interfaces, such as the **rpc_mgmt** interface, partial bindings without an object UUID are rarely adequate, since the endpoint mapper cannot

know which server supporting the ubiquitous interface is of interest to the client. The usual model is that the ubiquitous interface is not exported to the name space. Instead, the client imports bindings based either on another interface supported by the server or an object UUID. If servers follow the recommendation to export at least one UUID with their bindings, no additional preparation will be necessary to allow their clients to successfully call the ubiquitous interfaces the offer. If they do not export the UUID, they will have to adopt the **rpc_ep_resolve_binding( )** method described in “Routing Policy” on page 96 .

Binding Methods

In view of what was said earlier about binding handles, the binding method chosen also will be a purely local matter for the client application and stubs. For example, it is perfectly feasible for a server manager to make explicit use of binding information via a binding handle parameter in a remote call, even though the client does not use an explicit handle for the call.

DCE RPC provides the automatic, implicit, and explicit methods for clients to manage bindings for remote procedure calls:

- Automatic method

  This is the simplest method of managing the binding for remote procedure calls of an entire interface. With the automatic method, the server exports its binding information to a namespace, and the client stub automatically manages a binding for the application code.

  The automatic method completely hides binding management from client application code. The stub imports the binding information and maintains a binding handle. The stub passes the binding handle to the runtime with the remote procedure call, and the runtime uses the binding handle to retrieve the associated binding information. If the client makes a series of remote procedure calls to the same interface, the stub passes the same binding handle with each call.

  With the automatic method, a disrupted call can sometimes be automatically rebound. The automatic rebinding requires either that the remote procedure never begins to execute or that the operation is idempotent. If the call meets either of these requirements, the RPC runtime automatically tries to rebind the client to another server (if one is available).

- Implicit method

  This is a relatively simple method of managing a binding for an entire interface. With the implicit method, prior to making any remote procedure calls, the client application code obtains server binding information from a namespace or a string binding. The client assigns a server binding handle to a global variable in the client application (for each interface using this method). When calling a remote procedure using the implicit method, the client stub passes the specific interface’s global binding handle to the runtime.

  Note: Multithreaded clients must be careful not to allow one thread to change the value of the shared global binding handle while another thread is using it.

- Explicit method

  This is a more complex yet more flexible method of managing a binding. As with the implicit method, the explicit method requires that the client application code call runtime routines to initialize a binding handle. In the explicit method,

however, this binding handle is supplied by the application code as a parameter to the remote procedure call. By allowing a client to manage bindings for individual calls, the explicit method enables clients to meet specialized binding requirements.

Figure 4 shows the distribution of responsibility for binding management in applications for each of the three methods. The top portion of each box represents the client application code written by the developer. The bottom portion of each box represents the client stub code generated from an IDL interface definition.

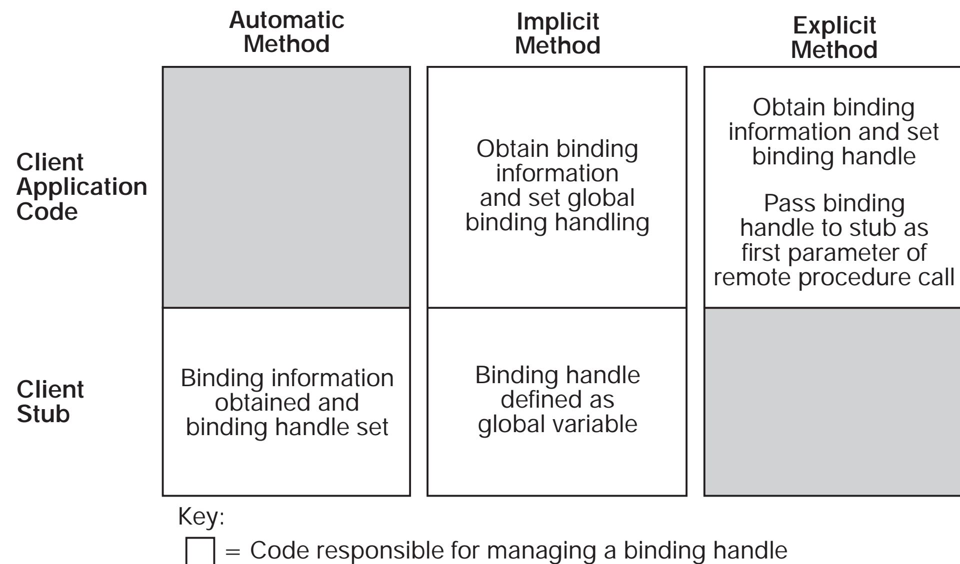


Figure 4. Methods of Binding Management

You can see from this figure that with the automatic method, binding management belongs completely to the client-stub code generated by the DCE IDL compiler. The implicit method provides the application developer with some control over binding management without having to pass a binding handle as a call argument. With the explicit method, the application developer is completely responsible for binding management. The automatic method requires the server to store binding information in server entries in a namespace; the implicit and explicit methods work with any source of binding information.

A client can use a combination of methods, even for an individual interface or if it uses more than one interface. for example, one interface might use the automatic method, another interface could use the implicit method, and a third could use the explicit method. In addition, some procedures for the interfaces that use automatic or implicit methods could use the explicit method instead. The method(s) of binding management for an interface is specified using the interface definition, the attribute configuration file (ACF), or both. In the interface definition, the explicit method can be specified for the whole interface, or for an operation by declaring a binding handle (using the IDL type **handle_t**) as the first parameter of the operation declaration.

The automatic and implicit methods are interface-wide and therefore mutually exclusive; that is, for a given interface, a client can use only one of these interface-wide methods. A client that uses either the automatic or implicit method for an interface can also use the explicit method for some or all of the remote

procedure calls to that interface. If the remote procedure call has a binding handle parameter, the explicit method takes precedence over either the automatic or implicit methods of managing bindings.

Explicit and implicit binding both give the client application means to select and modify the binding information used by calls. Explicit binding allows the client to specify binding information per call. This method may be established either by declaring a binding handle parameter as the first parameter for a call in the IDL, or by applying the **[explicit_binding]** attribute in the associated ACF, either to the interface as a whole, or to specific operations.

Implicit binding allows the client to establish a default binding for an interface. When the **[implicit_binding]** attribute is applied to a data item in the ACF, then each call that does not specify an explicit binding parameter (either in the IDL or via the **[explicit_binding]** attribute in the ACF) uses the default binding information referenced by the implicit binding data item.

With automatic binding, the client stub finds a useable binding for each RPC. Automatic binding is the default for any operation when the following three things are true:

- Implicit or explicit binding has not been specified in the ACF for the interface
- The call does not specify an explicit binding handle parameter
- The ACF does not specify explicit binding for the call

The semantics of automatic binding may differ between the first and subsequent calls on an interface. When the runtime does not have a cached compatible binding, the stub will perform a namespace search to find and import one. The imported binding will be cached for use in subsequent calls. If the client-server connection for the cached binding fails, the client stub will attempt to find a new binding. Therefore, it is possible that later calls will not be made on the same binding, and possibly will even be made to a different server.

A server binding handle that the runtime provides directly to an application is a primitive binding handle. To declare a primitive binding handle, application code uses the predefined RPC binding handle data type **rpc_binding_handle_t**, and an interface definition uses the IDL data type **handle_t**. Primitive binding handles offer a simple means of referring to binding information, which works in most cases. The automatic method of binding management always uses primitive binding handles.

Applications that use the implicit or explicit methods of binding management can choose to store primitive binding handles in an application-specific data structure known as a customized binding handle. Customized binding handles enable application developers to manage binding information to meet the special needs of a specific application. For example, a customized binding handle can be the handle of a file whose records contain the information required to construct a string binding.

Using customized binding handles requires the application developer to perform several special tasks. The RPC interface definition must include a declaration of the customized binding handle as a data structure with a handle data type; this is done by using the **handle** attribute. The client application code must contain specialized procedures that the client stub calls to obtain a primitive binding handle from the customized handle and to release any resources, such as memory, used for the customized handle.

When a customized binding handle is used with the explicit method, responsibility for setting the binding handle shifts to the client stub. The client code provides procedures for obtaining the primitive binding handle from the customized handle and for freeing the primitive binding handle after the call completes. However, it is the stub that calls these procedures to set and free the primitive binding handle.

Calls made with a context handle and no explicit binding handle also have automatic binding semantics. That is, such calls will use the cached binding associated with the context handle. Of course, this binding may have been constructed by the client application and passed, either as an explicit or implicit binding, to the call that returned the context handle. Also, the stub will not attempt to renew such a cached binding if the client-server connection fails. Even if the server is still running and the connection could be reestablished, the server will have rundown the context it is holding for the client, so that the context handle will no longer be valid. When implicit binding is in effect, a call made with a context handle and without an explicit binding parameter will use the cached binding associated with the context handle rather than the implicit binding.

The following table summarizes the binding semantics applied to a client operation:

Table 4. Binding Semantics

ACF	**ACF**	**ACF**	**ACF/IDL**	**IDL**	**Binding**
auto_handle attribute?	**implicit_handle attribute?**	**explicit_handle attribute on interface?**	**explicit_handle attribute on operation?**	**context handle?**	**Semantics**
No	No	No	No	No	Auto
No	No	No	No	Yes	Auto (context handle)
No	No	No	Yes	No	Explicit
No	No	No	Yes	Yes	Explicit
Yes	No	No	No	No	Auto
Yes	No	No	Yes	No	Explicit
Yes	No	No	No	Yes	Auto (context handle)
Yes	No	No	Yes	Yes	Explicit
No	Yes	No	No	No	Implicit
No	Yes	No	Yes	No	Explicit
No	Yes	No	No	Yes	Auto (context handle)
No	Yes	No	Yes	Yes	Explicit
No	No	Yes	No	No	Explicit
No	No	Yes	Yes	No	Explicit
No	No	Yes	No	Yes	Explicit
No	No	Yes	Yes	Yes	Explicit

When a binding handle is selected automatically by the client stub, there is no way for the application to specify authentication data. In principle, it would be possible to have the client authenticate itself to the server in such a case, although a client that does not care about which server it calls obviously cannot authenticate the server.

In practice, calls made with automatic bindings are simply unauthenticated. Therefore, if your application cares about authentication, it should avoid using automatic binding.

Chapter 5. Using the DCE Name Service

Correct use of the DCE RPC Name Service Interface (NSI) is essential to the operation of a distributed application, since NSI is the medium through which the application's distributed parts must find each other. NSI works with named database entries which are hierarchically organized into subdirectories and referenced by the familiar pathname convention.

Introduction to Using NSI

It is important to remember that names and objects are separate things in DCE. Consider, for example, these two DCE names:

```
/.../tinseltown.org/dce/printers/macmillan

/.../tinseltown.org/dce/employees/goethe
```

These strings are *not* filenames or file directory names; if you attempt to execute the **ls** command on them, you will only get an error message. They are pathnames that identify entries in the DCE Directory Service, which is DCE's database for storing distributed information. This database is often informally referred to as the namespace.

The most important type of distributed information stored in the namespace is information that enables RPC clients to rendezvous with RPC servers; it is called binding information. The directory service can be used to hold other kinds of data too, but the main subject of the following discussions will be its use as a binding repository.

The set of binding name entries is like a huge data structure of pointers from object names to object locations, and the directory service is used mostly as a public DCE locational database, enabling servers to advertise themselves and the objects and resources that they manage, and clients in turn to find and access them. You should never confuse objects with their names; the two are separate things. In particular, the directory service data associated with a name is held in one place (namely, the directory server's database), while the data associated with the object named is held in other place (namely, the object server's database).

How then, you might ask, are filenames represented in DCE? Here are two examples of remote filenames:

```
/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps

/.../tinseltown.org/fs/doc/tolstoy/novels/war_and_peace/chap2.ps
```

As you may have guessed, these too are namespace entries, but the entries in this case refer to remote files, and the entry name as a whole is the remote filename. What makes these names different from the other two names given earlier is their third element, **fs/**, which identifies a junction from the DCE Directory Service's namespace into the DCE Distributed File Service's own, separately maintained, namespace.

What happens is that **/.../tinseltown.org/fs** is the DFS file server's DCE namespace entry, and any attempt by a file service client to access a file object whose name begins with **/.../tinseltown.org/fs** will implicitly bind to this server, which will then be

responsible for finding, in its own namespace, the file object referred to by **doc/jones/app.gd/chap2.ps** or **doc/tolstoy/novels/war_and_peace/chap2.ps** and performing the requested operations on it.

The UUID

Thus, it is a mistake to suppose that a name is identical to an object. The name merely points in the direction of the object it names. Objects do, however, have identifiers. These are the 128-bit universal unique identifier (UUID) data structures, which are the identities that the DCE components recognize. They are not usually seen by users, although they play a part in the object-finding process.

UUIDs are used within DCE to identify all sorts of things. From the standpoint of the application programmer, they have two main uses: to identify objects and to identify interfaces.

Object UUIDs

Although object is necessarily a rather vague term, a reasonable definition would be the following: an object is any DCE entity that can be accessed by a client, and which can be represented by a namespace entry and identified therein by a UUID. This category can include servers, devices, and other resources. UUIDs that are used in this way are called object UUIDs in order to distinguish them from the other main use of UUIDs, namely to identify interfaces (*interface UUIDs*). The difference between these two uses consists only in the way the UUIDs are interpreted by the name service and RPC runtime. Note that it follows from this discussion that an interface is usually *not* an object. Clients do not normally access an interface as such; the interface is rather a description of the rules of access.

As far as the DCE RPC and name service mechanisms are concerned, it is enough if a client is brought into contact with some server, as long as that server offers the service the client is looking for; in other words, as long as the server offers the interface the client wants to use. To accomplish this rendezvous, interface UUIDs are sufficient. They are also mandatory. There cannot be a client/server relationship without an interface, and the entire RPC runtime mechanism is dependent on the concept of interfaces.

Object UUIDs are different. The RPC runtime usually does not care if they are present or not. But if they are present, they activate various runtime mechanisms that allow clients and servers to be much more specific (always within the bounds of a given interface) about what servers are bound to, and/or what resources the servers will use to fulfill the clients' requests. How this works is explained later in this chapter.

Interface UUIDs

Every IDL-compiled interface specification has its own UUID associated with it, and the IDL-generated stub routines include this interface UUID with every operation request or return sent over the network by clients and servers. In this way receiving stubs ensure that they and the sending stubs are sharing exactly the same interface. If the interface UUIDs are different, or are not present, then the remote call will not be completed. But interface UUIDs, although they are required, play only a secondary role in a client's finding the interface (that is, finding a server that offers the interface); the main tool for this is NSI, which makes use of the DCE Directory Service, as explained later in this part of the chapter.

Summary: Names and UUIDs

Both names and UUIDs identify objects. But names are separable from the objects they identify, and are only as trustworthy as the binding information their entries contain. UUIDs, on the other hand, are inalienable identifiers. Once the desired binding information for an interface or an interface/object combination has been found and used, the name that was used to retrieve it can be forgotten; it is of no further use. This is not true of either interface or object UUIDs.

Note that names become completely unnecessary only if clients have some other means of obtaining valid binding information for the desired service, such as string bindings.

The following figure illustrates how the information a client finds through a name is turned into network contact with the object named.

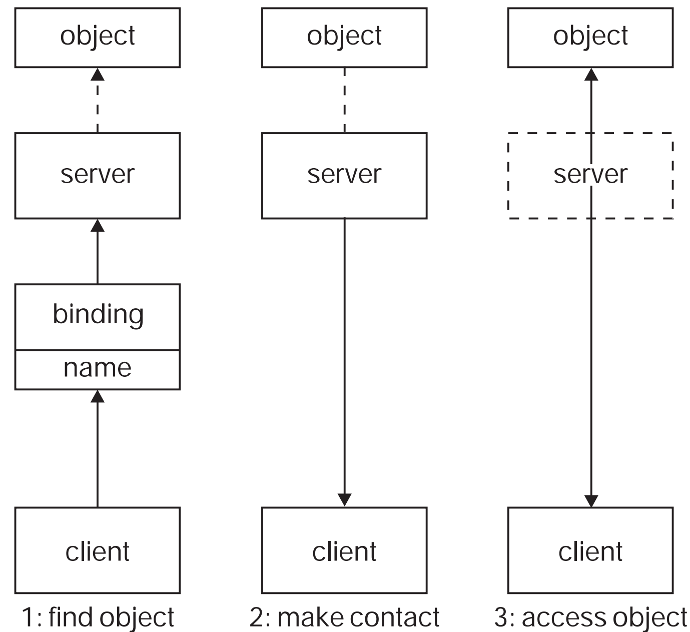


Figure 5. How a Name Turns into an Object

Binding to an Object

The difference between, for example, reading a local file on a single machine and performing the same read on a remote file in DCE is like the difference between reading information from a phone book yourself and dialing an operator for the same information. The remote operation requires the addition of another active entity that can be requested to perform it, since you cannot. Associated with every piece of remote data available on a network is a remote server to manage that data and make it available. The user may not see the server; even the client may be unaware of it, but it is there.

The DCE documentation often speaks of “binding to an object.” In reality, clients can bind only to servers, which then may be requested to perform operations on objects that are under their management. However, it is possible for a server to put bindings into namespace entries that are named for the objects that it manages. Furthermore, these exported bindings can be tagged with object UUIDs in such a

way that incoming remote calls from clients can be applied by the server to the object whose name entry the binding was read from (the details of this technique are described later in this chapter). When an application uses this kind of binding model, it is reasonable to say that the client is logically bound to the object, although it is physically always bound to the server that manages the object.

Junctions

Namespace junctions are another example of the hidden server effect. The following remote filename was discussed earlier:

```
/.../tinseltown.org/fs/doc/jones/app.gd/chap2.ps
```

There it was explained that **doc/jones/app.gd/chap2.ps** is an entry in DCE DFS's own namespace, while **/.../tinseltown.org/fs** is a DCE namespace entry. Suppose a user enters the following:

```
ls -l /.../tinseltown.org/fs/doc/jones/app.gd
```

The clerk agent program (called as a result of the user's entering **ls**) will bind to the remote file server via its **/.../tinseltown.org/fs** DCE namespace entry, and pass to it the residual DFS entry name **doc/jones/app.gd** along with other parameters. The **ls** command behaves this way because the underlying (VFS+ layer) system calls are coded that way. The DFS server then performs the request (note that the details of interaction within DFS are somewhat more complex than implied by this description). The user only types the command line; the rest is done by DCE, and a directory listing appears on the user's screen.

Because the VFS+ system routines, which are used by all possible clients of DFS services (for example, commands like **ls** and **rm**, library routines like fopen( ) and fclose( )), know about the remote file server at **/.../tinseltown.org/fs** and bind to it correctly, the transition from the DCE to the DFS namespace is completely transparent to users. And this is how junctions work. As long as all possible clients behave correctly with a name that includes a junction, the junction will not be perceptible to the clients' users.

A Junction Example

The next figure illustrates the principle of junctions. A junction server, which is reached normally through binding information in the DCE namespace, maintains its own namespace of named objects. The junction server's clients allow users to refer to these objects by actually concatenating the server's entry name and an object's internal name. The client then in effect breaks this string apart by contacting the server named in the first part of the string, and passing to it the second part, which is a valid name within the server's namespace. The client's user seems to access the object directly.

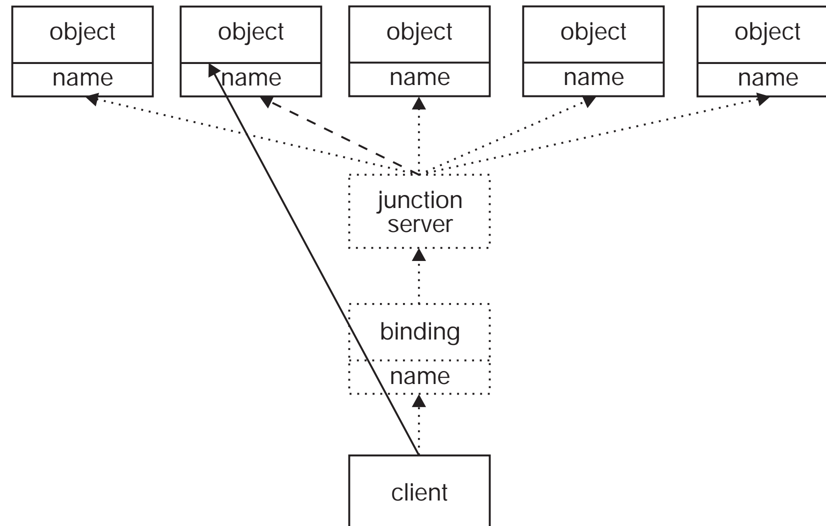


Figure 6. A Namespace Junction

The dashed lines in the above figure show the progress of the client's efforts to get access to the desired object, which involves acquiring a binding to the junction server, making contact with it, and passing to it the object's name. The solid line shows the apparent direct access to the object that the client's user seems to enjoy. The dotted lines show other possible paths of access to the other objects that the server manages.

Junction protocol is generally a private matter between an application's clients and servers. However, the **acl_edit** command uses a generalized protocol.

Junctions and the ACL Editor

The binding routines that **acl_edit** uses are discriminating enough to detect a junction anywhere in an entry name that is passed to it. This allows a distributed application to have its own namespace for objects with ACLs on them, rather than burdening the DCE namespace by separately exporting binding information for every one of these objects. The separate objects have to be made publicly accessible somehow because entities should be able to access ACLs directly, regardless of whether they happen to already be in contact with the server that manages the ACL'ed object, and indeed regardless of whether or not they happen to be a client of the particular server to which the objects belong.

Suppose, for example, a user enters the following in order to interactively edit the ACL for the printer object **cotta**, where the namespace entry for a print server is **/.../tinseltown.org/dce/dce_print**, and there is no **/.../tinseltown.org/dce/dce_print/cotta** entry in the DCE namespace:

```
acl_edit  /.../tinseltown.org/dce/dce_print/cotta
```

The binding routine, **sec_acl_bind( )**, which is called internally by **acl_edit**, receives an error when it tries to bind to the object **cotta**. However, the DCE Directory Service also tells it how much of the name it passed is valid. The **sec_acl_bind( )** routine then retries the binding operation, this time through the valid entry name **/.../tinseltown.org/dce/dce_print** and passes the residual part of the name (`cotta`) as a parameter. Now it is up to the application ACL manager to interpret the residual name correctly and find the requested ACL.

Name Service Terminology

DCE RPC NSI is an RPC-based interface that uses the DCE Cell Directory Service (CDS) as its database. The NSI routines do not constitute a general interface into CDS as such; they are a set of specialized routines whose purpose is simply to provide ways for RPC servers to advertise themselves to RPC clients, and for clients to find and bind to them.

In fact there is no public general application programming interface (API) to CDS. There is a general CDS interface that is used internally by the DCE components, but applications normally access CDS through NSI. Applications can get full access to CDS, if necessary, by using the XDS interface.

CDS Entries

NSI uses a subset of the many possible kinds of CDS entry in order to accomplish its tasks. CDS entries are characterized by the CDS attributes they have; each entry can have one or more such attributes. Each separate attribute defines that entry's ability to contain one or more items of a particular kind of simple or complex information.

The name service creates and uses CDS entries that use only the following four attributes:

binding
: The entry has a field that can contain one or more sets of binding information. When the field is read, a binding handle that contains the necessary information from one of these sets is returned, in no particular order.

object
: The entry has a field that can contain one or more object UUIDs. When the field is read, one of the UUIDs is returned, in no particular order.

group
: The entry has a field that can contain a pool of one or more references to other (independently existing) NSI entries; each time the field is read, one of these entries is returned. Different entries are returned on successive reads, but the order of return is undefined.

  Note that the other NSI entries referred to in the group can themselves be server or group entries. As a result, the act of reading from a group attribute can, depending on the actual API routine called, lead to a series of nested operations. Any nesting is transparent to the client application, however, which seems to perform a simple read and to receive the contents of a single entry in return.

`profile`
: The entry has a field that can contain one or more prioritized elements, each of which consists of a reference to another (independently existing) NSI entry. When the field is read, the elements are read in a specified order. The entry referred to in the element may itself be a server or a group or a profile. As a result, any element may in fact, depending on the actual API routine called, resolve on access to a nested path of referred-to entries. As with group entries, this is transparent to the client application.

Although a single entry could contain both group and profile attributes (and for that matter, binding and object attributes as well), it is not a good idea to mix attributes in this way because the results of importing (reading) from such an entry are too indeterminate.

The typical name service entries are as follows:

server entry
Contains a binding and an object attribute, making it suitable for containing the necessary binding information for a single server.

group entry
Contains a group attribute.

profile entry
Contains a profile attribute.

There are no official names for hybrid entries that contain other combinations of attributes, which is perhaps another reason for not creating such entries.

The general name for entries that contain any of these attributes is *NSI entries*, since they are a by-product and tool of the NSI DCE RPC library routines.

CDS Entry Attributes

Within the DCE Directory Service, entry attributes such as the four previously described attributes are identified by object identifiers (OIDs). This is an exception to the general rule that things in DCE are identified by UUID.

OIDs are not seen by applications that restrict themselves to using only the name service routines (**rpc_ns_ *( )**), but these identifiers are important for applications that use the X/Open Directory Services (XDS) interface to create new attributes for use with namespace entries.

As was seen in the immediately preceding sections, the name service makes use of only four different entry attributes in various application-specified or administrator-specified combinations. CDS, however, contains definitions for many more than these, and attributes from this supply of already existing ones can be added by applications to NSI entries through the XDS interface. Attributes that already exist are already properly identified, so applications that use these attributes do not have to concern themselves with the OIDs, except to the extent of making sure that they handle them properly.

A further possibility is that an application requires new attributes for use with namespace entries. Such attributes can be created using the XDS interface. When it creates new attributes, the application is responsible for tagging them with new, properly allocated OIDs.

Unlike UUIDs, OIDs are not generated by command or function call. They originate from the International Organization for Standardization (ISO), which allocates them in hierarchically organized blocks to recipients. Each recipient (typically an organization of some kind) is then responsible for ensuring that the OIDs it received are used uniquely.

For example, the following OID identifies the NSI profile entry attribute. This number was assigned by the Open Software Foundation out of a block of numbers,

beginning with the digits **1.3.22**, which was allocated to it by ISO, and OSF is responsible for making sure that **1.3.22.1.1.4** is not used to identify any other attribute.

```
1.3.22.1.1.4
```

When applications have occasion to handle OIDs, they do so directly, since the numbers do not change and should not be reused. However, for users' convenience, CDS also maintains a file (whose name is **/opt/dcelocal/etc/cds_attributes**) that lists string equivalents for all the OIDs in use in a cell, in entries like the following:

```
1.3.22.1.1.4        RPC_Profile     byte
```

This allows users to see **RPC_Profile** in output, rather than the mysterious **1.3.22.1.1.4**. Further details about the **cds_attributes** file and OIDs can be found in the *OSF DCE Administration Guide—Core Components* .

Broadly speaking, the procedure you should follow to create new attributes on CDS entries consists therefore of three steps:

1. Request and receive, from your locally designated authority, OIDs for the attributes you intend to create.
2. Update the **cds_attributes** file with the new attributes' OIDs and labels; that is, if you want your application to be able to use string name representations for OIDs in output.
3. Using XDS, write the routines to create, add, and access the attributes.

Non-NSI attributes on NSI entries can be very useful, even though you cannot access the extra attributes through the name service routines but must use XDS instead.

Binding

In order to highlight the essentials of name lookup and storage and the management of binding information, many details of DCE RPC operation are either greatly simplified in the following descriptions or omitted altogether.

A binding is a package of information that describes how a client can contact and communicate with a particular server. Although the underlying protocol that implements the communication can be connectionless or connection-oriented, the relationship itself is still expressed as a binding.

Importing and Exporting Bindings

The name service exists to store server binding information into the cell namespace, and to retrieve that information for clients. Using NSI, servers export their binding information to be stored under meaningful names, and clients import these bindings by looking up those names. Thus, the locations of the servers can change, but clients can continue to use the same names to get bindings to the servers. The following figure shows how client and server use the name service.

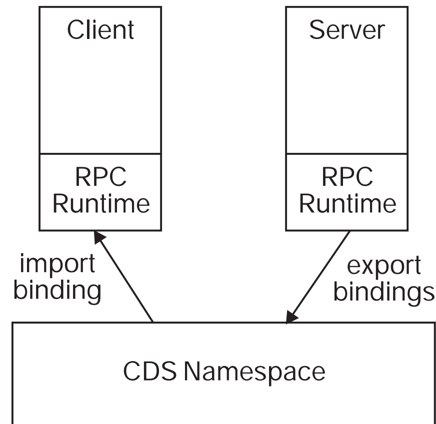


Figure 7. Client and Server Use of the Name Service

When a prospective client attempts to import binding information from a namespace entry that it looks up by name, the binding is checked by NSI for compatibility with the client. This is done by comparing interface UUIDs. The client presents an interface UUID when it begins the binding import operation; the UUID of the interface being offered is exported to the name entry, but not in the binding handle itself, by the server. If these interface UUIDs match, then the binding handle contained in the entry is considered compatible by the RPC runtime and is returned to the client. If more than one handle is contained in the entry (this is often the case), they are returned one by one on successive imports. NSI also checks for protocol compatibility.

The import routines will return only client-compatible bindings, but a client can sift through the returned bindings and make its own choice as to which ones to use, based on its own criteria. The technique by which this is done consists of converting the bindings into string bindings, and then inspecting (or comparing) the strings.

Note that binding handles do *not* include an interface UUID. Binding handles do contain a host address, an endpoint, and an optional object UUID, among other things. The interface UUID is associated with the interface's stub code, which inserts it into outgoing RPCs and checks it in incoming ones, thus guaranteeing client/server operational compatibility. This allows binding handles to be used very flexibly: once a client has successfully bound to a server, it can utilize any of the interfaces that server offers, simply by making the desired remote call.

Summary

The mapping from name to server that occurs when bindings are imported from the namespace is indirect because binding is a two-step process: first the binding handle is obtained by lookup from a named entry, and then the handle is used to reach a server. The crucial point is that the imported handle will not usually contain a complete binding to a specific server (namely, the one that happened to export it). Completion of the partial binding occurs later, when the client makes its first remote procedure call; the RPC runtime uses UUIDs, not names, to determine how it should complete a binding.

Partial Binding and the Endpoint Mapper

Binding handles imported by clients from the namespace normally contain only partial binding information. The exported binding information is sufficient to locate the DCE host daemon on the server's host (the machine the server resides on), but it does not yet include a specific endpoint (UDP or TCP port number) for the desired service on that host.

The reason for omitting dynamic endpoint information in exported binding handles is to avoid unnecessary multiplication of accesses to the namespace. Since dynamically generated endpoints are necessarily reassigned every time a server starts up, entering them into the namespace (and thus forcing CDS to propagate the new information throughout the various directory replicas) would greatly increase namespace housekeeping chores.

Thus, the last step in the binding process is obtaining an endpoint. The step is performed transparently as far as the client is concerned. It is accomplished by the endpoint mapper service of the DCE host daemon, **dced**, when the client makes its first call to the partially bound-to server. The endpoint mapper service manages its own private database of server endpoints for the host on which it is located. The endpoints are registered by the servers as part of their startup routine.

The binding information that accompanies a prospective client's first remote procedure call takes that call to the well-known endpoint of **dced** on the exporting server's host machine. The endpoint mapper now takes over. It looks up a valid endpoint for the requested service, copies it into the binding handle, and transfers the call to that endpoint. Subsequent calls from the client, which now has a binding with one of the server's endpoints, will bypass the endpoint mapper.

The endpoint mapper picks an appropriate endpoint for an incoming partial binding by matching interface UUIDs by default. Any endpoint that has been registered under an interface UUID that matches the incoming interface UUID, which identifies the interface requested by the prospective client, is eligible for selection. This mapping process is called *forwarding* when it occurs with connectionless protocols, and *mapping* when it occurs with connection-oriented protocols.

The following figure shows the endpoint mapper service completing a binding.

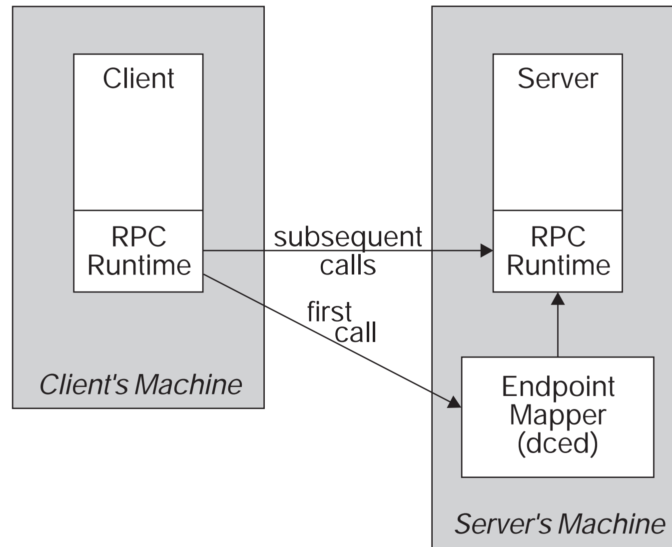


Figure 8. The Endpoint Mapper Service Completes a Binding

There is an exception to this scheme. Some servers are designed to occupy well-known addresses. The DCE host daemon itself, **dced**, is reached in this way, making its accessibility independent of whether or not the namespace is accessible. The endpoint(s) of a well-known address do not change; they are usually specified in the application's interface specification (contained in its **.idl** file). Bindings to servers that use well-known endpoints are already complete at the time of import; the endpoint mapper never sees these bindings.

Interface Ambiguity and Partial Bindings

The interface UUID, which was generated by the IDL compiler, uniquely identifies the set of operations that the client will access through that interface. In short, it identifies the interface. An interface UUID may also happen to identify a server which offers that interface. But if more than one server on the same host offers the same interface (which could easily be the case), the interface UUID alone will not be sufficient to identify a *specific* server. The result is that if a remote call comes in with such an ambiguous interface and a partial binding, the endpoint mapper will have to randomly choose any one of its eligible registered endpoints, complete the binding with it, and send the call on to that server.

Imagine several print servers residing on the same machine (see Figure 9 on page 114). Each server manages a group of printers that share a common physical location. All the printers in room A are managed by the A print server, all the printers in room B by the B print server, and so on. Now suppose each of these servers has a separate entry in the namespace. The following figure shows the sequence of events that occurs.

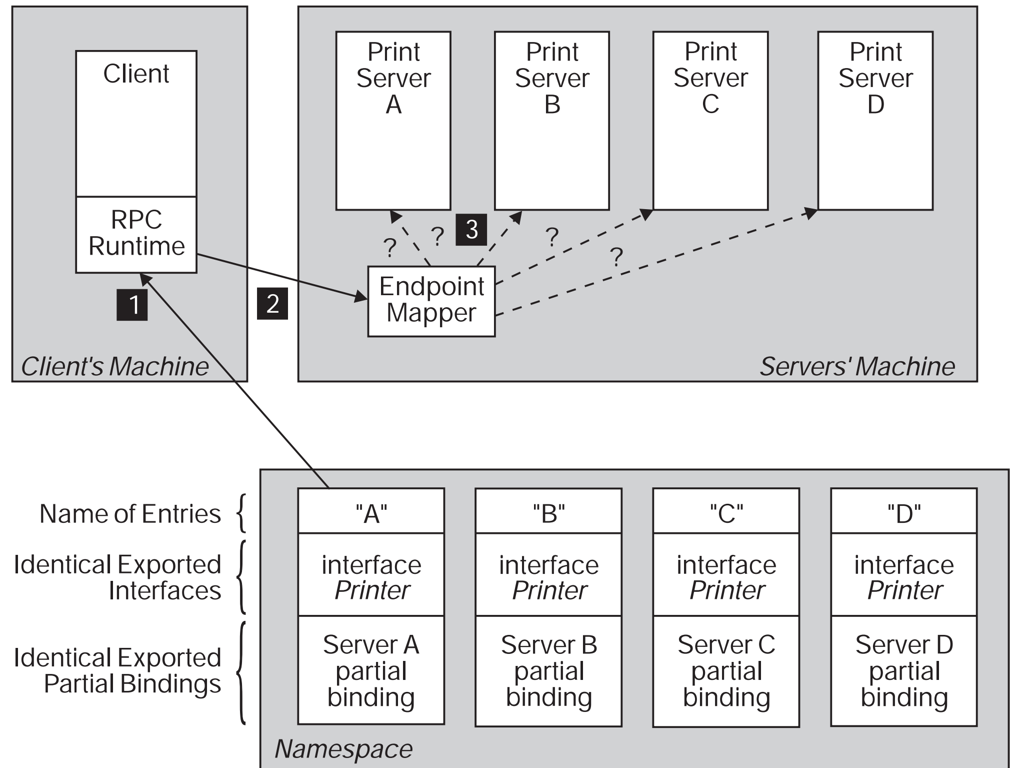


Figure 9. Print Server Entries in Namespace

The following steps describe the sequence of events shown in the above figure:

1. The client imports a partial binding to the *printer* interface from the entry A in the namespace.
2. The client makes its first call with the binding it imported from A.
3. The endpoint mapper at print server A's host, when it receives the call from the client, has no way of knowing which of the four print servers it should map the call to, since all four servers have registered their endpoints under the same interface. It therefore picks one at random to complete the binding.

The entry names are different, but the partial binding information contained in the entries is identical, since the servers' host machine is the same. The interface UUID included in the call is no help, since that same interface is offered by all the servers. A client seeking a print server may not care to which server (and thus to which printer) its request goes, but then again, it may care. If it does, there is a way it can specify a server so that the endpoint mapper can select an appropriate endpoint to complete the partial binding.

Using Object UUIDs to Avoid Binding Ambiguity

Binding handles can contain, besides host address and endpoint information, an object UUID as well. The endpoint mapper will try to match an object UUID contained in a binding handle with one of the object UUIDs associated with its map of registered endpoints. This allows even a partial binding to specify a target more precisely than just by host machine. Since object UUIDs are generated by the **uuid_create( )** function call (see the *OSF DCE Application Development Reference*), servers can create as many of them as they need.

For the print server example discussed in the previous section, the namespace entries for the servers could be set up as shown in the following figure.

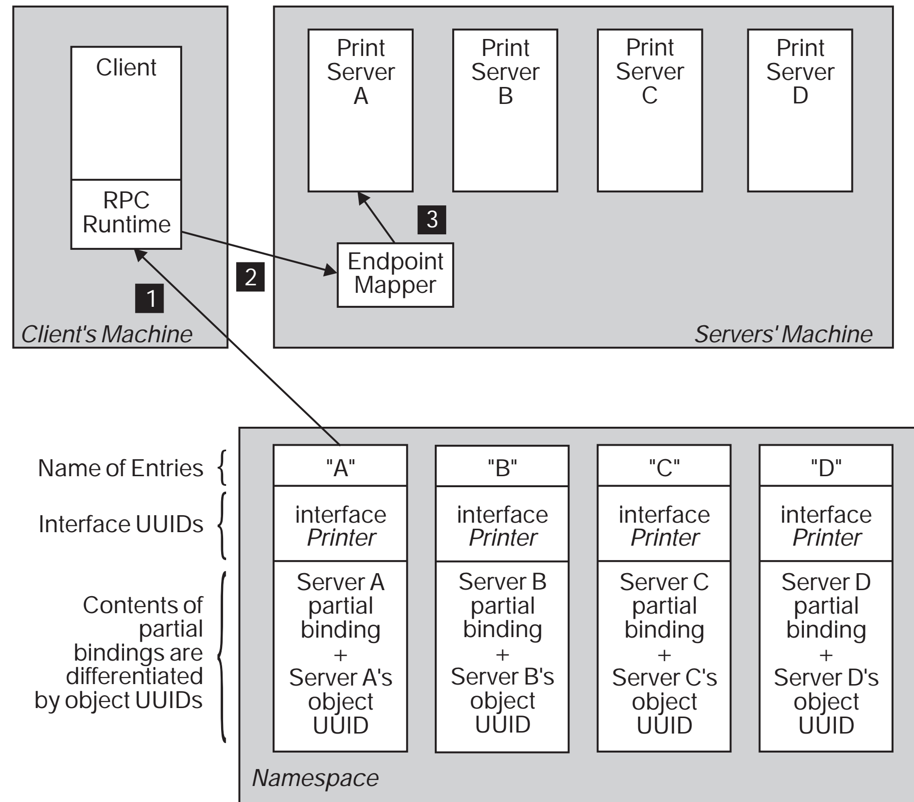


Figure 10. Print Server Name Entries with Object UUIDs

The following steps describe the sequence of events shown in the preceding figure:

1. The client imports a partial binding to the *printer* interface from the entry A in the namespace.
2. The client makes its first call with the binding it imported from A.
3. This time the endpoint mapper at print server A's host is able to match the call with A's registered endpoints, because the endpoints have been registered with both the *printer* interface and print server A's object UUID, and the incoming call's partial binding also contains print server A's object UUID.

Each server has exported a set of partial bindings that differs from all other servers' by its object UUID (which thus becomes, in effect, a server ID). If, for example, server A has properly registered its endpoints with the same object UUID as the one it exported its bindings with, the endpoint mapper will make sure that a partial binding exported from server A's name entry will result in a full binding to server A.

Now suppose that each print server sets up a separate namespace entry for each printer it manages. The printers themselves would, in effect, be identified by their own object UUIDs. The following figure illustrates this.

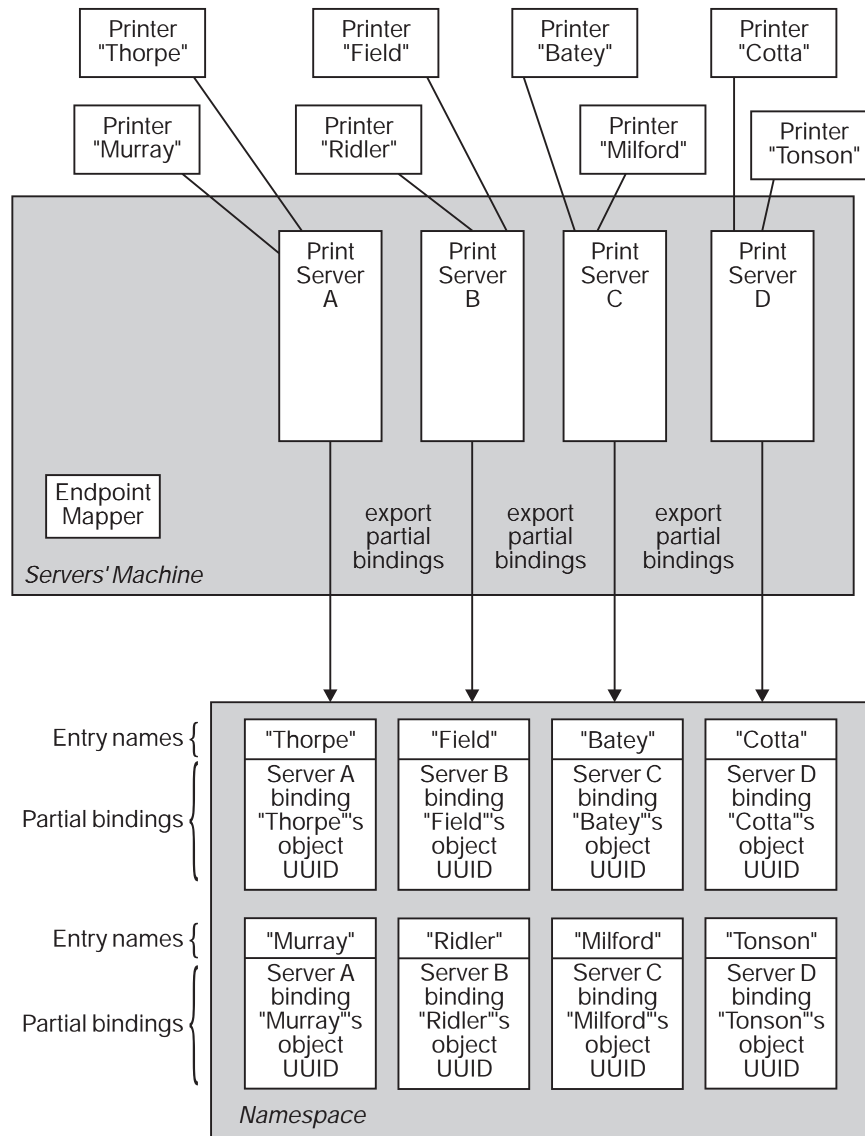


Figure 11. Separate Printer Name Entries

Now a client will be able to access a specific printer by importing a binding handle from that printer's name entry. The endpoint mapper at the target host would compare the object UUID in the partial binding with the object UUIDs registered by the print servers, and select an appropriate server. The server in turn would also use the object UUID to select the correct printer for the request, if it managed more than one printer. A namespace set up in this way with a separate entry that contains a unique object UUID for each accessible service resource is called an *object-oriented* namespace.

An Object-Oriented Namespace

Object-specific entries are namespace entries that each contain binding information only for one specific object or resource, as demonstrated in the last printer service shown in the last previous figure. *Object* can mean any of several things, depending on what kind of service the application's servers are offering. Here are some examples.

Table 5. Some Examples of Objects

Service	Object(s)
Printing	A specific printer
Process Server	A specific server
Queue Service	The print queue, the kill queue, the backup queue

Thus, for a client that wants to have a file printed, it is natural to allow it to specify a printer as a destination. Therefore, the client would bind to the print server through a name entry that specifies a printer. To send something to a different printer, the client would import a binding from the name entry for that other printer. The server may (or may not) be identical, but the object UUID in the binding handle returned would uniquely specify the one printer represented by that entry.

On the other hand, consider an application that returns statistics about the processes currently active on a group of machines. In this case it would be reasonable to regard the server as the object. In the namespace entries for such an application, each entry would uniquely represent one server. A client would import a binding from the name entry for the server it wanted to work with.

In other words, object is a handy way of saying "the thing that clients will want to access" in order to accomplish the task set for the application. If the namespace is organized correctly, clients will be able to import bindings from these objects' entries.

Setting Up an Object-Oriented Namespace

Once you have distinguished the objects your application uses, you must decide on an appropriate set of names for the entries themselves. The entries can be created either by the application (server), if it has the necessary privileges, or by a system administrator using the **rpccp** command interface.

After the entries have been created, each server must do the following:

1. Create an object UUID for each object managed by the server under an interface, insert it into the binding handle(s) for that object, and export the handle(s) for each object to a separate entry in the namespace.

   Note that the object UUID should be generated and exported in general *only once* per created namespace entry, and not each time the server starts up (see the example that follows of how to do this). When a newly restarted server exports its partial bindings, nothing actually happens in the namespace because the partial binding information remains the same (unless the server has moved to a different machine). However, if the object UUIDs are regenerated, then the change in exported information will force needless update activity in CDS, which is where the entries exist.
2. Register with the endpoint mapper the full bindings (including endpoints) obtained for the interface; **rpc_ep_register( )** performs this operation.

One way of avoiding unnecessary regeneration of object UUIDs would be to have a restarted server check the namespace for the presence of its previously exported object UUIDs, as demonstrated in the following code fragment. Refer to the *OSF DCE Application Development Reference* for further information on the function calls.

```
have_object = false;

/* Create an inquiry context for inspecting the object  */
/*  UUIDs exported to "my_entry_name"...           */
rpc_ns_entry_object_inq_begin(my_entry_name_syntax,
                              my_entry_name,
                              &context, &st);

/* If we successfully created context, look at         */
/* object UUIDs...                                   */
if (st == rpc_s_ok)
{
    /* Try to get one object UUID from the entry...     */
    rpc_ns_entry_object_inq_next(context, &obj, &st);

    /* If an object UUID is there already, we don't     */
    /* need to generate another one...                */
    have_object = (st == rpc_s_ok)

    /* Delete the inquiry context...                    */
    rpc_ns_entry_object_inq_done(&context, &st);


/* If there were no object UUIDs in the entry,         */
/* generate one now...                               */
if (! have_object)
{
    uuid_create(&obj, &st);

    /* Put it in an object UUID vector...               */
    objvec.count = 1;
    objvec.id[0] = &uuid;
}

/* Export bindings.  If an object UUID was generated,    */
/* export it too...                                   */
rpc_ns_binding_export( my_entry_name_syntax,
                       my_entry_name,
                       my_interface_spec,
                       my_bindings,
                       have_object ? NULL : &objvec, &st);
```

Whenever you want to offer more than one instance of the same interface on the same host, you *must* distinguish by object UUID the binding information in the name entries exported by the servers, if it is important to distinguish among the servers when binding to them. Otherwise, the endpoint mapper’s selection of an endpoint with which to complete the binding from among all the servers on that host that offer the appropriate interface will be random.

The next figure illustrates what such an object-oriented namespace should look like.

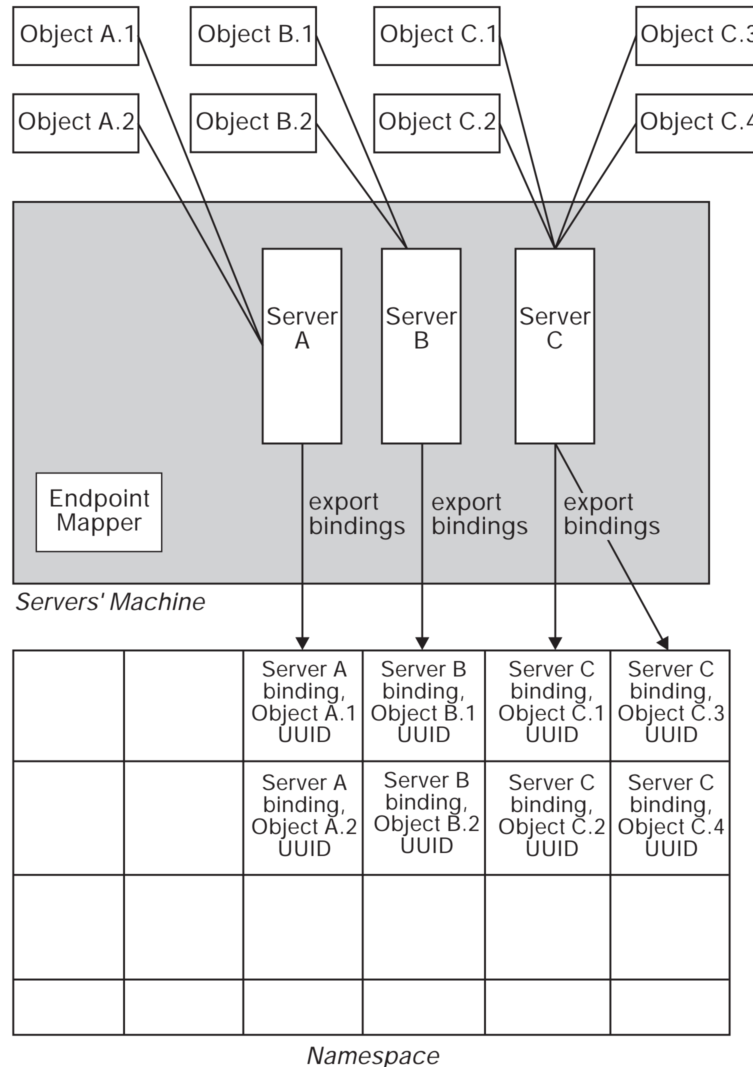


Figure 12. Object-Oriented Namespace Organization

Each entry has a name denoting the object represented, although the names are not shown in this figure.

Under this model, clients bind to servers via named objects in the namespace, each of which contains enough specific information in its partial binding to allow the endpoint mapper at the destination host to choose an appropriate endpoint for the incoming RPC.

By setting a namespace up this way, however, you do not necessarily restrict yourself to this one model for accessing binding information. Through the use of two other types of entry, groups and profiles, which can be superimposed on the simple object model, you can set up models where clients bind to abstractions such as services, or directly to the servers themselves. These techniques are described in the next section.

Nevertheless, at this point you have enough information to set up a namespace that consists of an entirely flat expanse of separate resource entries. Bindings can be

imported by clients by looking up specific names. If the client has no specific name to look up, or if the lookup on the name(s) it has fails, it has no alternative way of binding to a server.

Groups and Profiles

Name lookups can be made more flexible with two other types of entry; namely, groups and profiles.

Group Entries

A group entry consists essentially of multiple independent other entries whose names are also associated under the group name. These other entries can be simple (single-name) entries, or they may themselves be group entries. Doing an import from the group entry will return the contents (the binding handles) of its included entries (which are called *members*), but the selection is made by the DCE RPC runtime, and from the client's point of view is undefined and implementation dependent.

In practice, the way this works with the usual binding import operations is as follows. Clients normally import bindings by first calling **rpc_ns_binding_import_begin( )** to set up an import context. Once this is done, successive calls to **rpc_ns_binding_import_next( )** will return binding handles from namespace entries until the handles have all been returned or the client decides to stop; the client decides which handle(s) to use based on its own criteria. When it is finished importing, it calls **rpc_ns_binding_import_done( )** to free the context.

The kind of entry the information is returned from is usually unknown to the client, which needs to know only a name to look up and the interface UUID by which it wants to bind. If the name is that of a simple server entry, then the bindings contained in that entry only will be returned. If the name is of a group entry, then bindings will be returned from members (single entries) of the group, selected (by the RPC runtime) in an undefined order. If one or more members of the group are themselves groups, then the same thing happens recursively whenever these lower-level groups are accessed.

Note that the group entry and its members are separate things. The group entry can be deleted, but its former members will continue to exist as independent entries, unless they too are explicitly deleted. Thus, you can implement a namespace organization where the same bindings can be imported through individual simple entries or through group entries, depending on how the client is coded.

Profiles

A profile entry specifies a search path or hierarchy of search paths to be followed through the namespace in order to obtain a binding to a server that offers a specified interface.

When a client imports from an entry that happens to be a profile, successive imports (accomplished by calling **rpc_ns_binding_import_next( )**) return the contents of entries that are read as a result of following the specified path through the namespace. All this is transparent to the client, which sees only the bindings returned. Profiles can be used to set up default paths and groups of paths for

users. The **RPC_DEFAULT_ENTRY_NAME** environment variable, which is the default entry name used by the name service in import operations, usually contains the name of a profile.

As with groups, the entries contained in profiles, which are called *elements*, exist independently of the profile entry itself.

A very important property of profiles is that they allow clients to know little or nothing about the organization of the namespace itself. Using the default case as an example, consider the following: if the profile at **RPC_DEFAULT_ENTRY_NAME** has been set up with elements containing entries for all possible active servers for a particular application, clients can simply import from this name and trust the profile mechanism to walk through the various compatible possibilities and return binding handles via successive calls to **rpc_ns_binding_import_next( )**. (Note that a profile entry is not limited to containing entries for just one interface; thus, **RPC_DEFAULT_ENTRY_NAME** could be set up to contain *all* the defaults for a cell.)

Summary of Namespace Entry Types

Clients access binding information in the namespace by looking up (by name) one of three different kinds of entry:

- A server entry
- A group entry, which contains other entries whose contents are returned to the caller when it reads the group entry
- A profile entry, which specifies a path of entries to be searched whose contents are returned to the caller when it reads the profile entry

Lookups behave differently depending on the kind of entry read. If an entry is a simple server entry, then the search begins and ends right there, whether successful or not. If the entry is a group, then the lookup is more complicated. A binding will be returned from among those that are found to be compatible by the name service, but within that category the selection is undefined. If the entry is a profile, then a specified path of entries is searched. The entries in this path may themselves be other profiles, or groups, or simple entries. The search continues until either a compatible binding is found, or the entire path has been unsuccessfully traversed.

Three Models for Accessing Binding Information

By adding groups and profiles to the object-specific namespace organization originally described, you can implement any or all of the following three basic models for accessing binding information:

- Clients bind to services
- Clients bind to servers
- Clients bind to resources or objects

Each of the three models is described in the following sections.

Access By Services

Servers have separate namespace entries; each server distinguishes the bindings it exports with its own identifier; that is, an object UUID that it generates for itself *the*

first time it starts up. These separate server entries are also members of group namespace entries, which represent services. The criterion for membership in a service group is that all the servers in it export the interface that identifies that service. (They may happen to export other interfaces as well.)

Clients, in effect, bind to services by importing their binding handles from the group entries. Note, however, that the server-specific entries still exist independently and are accessible to lookup.

This model is appropriate for applications where clients do not care which server they happen to bind to or where that server is located as long as it offers the desired service. The eligible servers are pooled into a group entry from which bindings to one of them are selected in an undefined order and returned whenever a client performs an import operation from the group entry.

Access By Servers

In this model, distinct servers have separate and distinct name entries, and clients import bindings directly from the server entries. Hence, an application using this kind of binding model will own just as many simple entries in the namespace as there are active servers.

Since the client in this model is looking for a specific server, imports will be done directly from the server entries. The only exception to this rule would be where two or more instances of a server were active on the same host, and it was indifferent to the client as to which one it is bound to. The entries for the multiple same-host servers then could be put into a group entry, and binding imports done from the group.

Access By Objects

Servers operate on or manage multiple objects. Clients use these objects (via the servers) as resources. For each such resource, the server creates a separate namespace entry and exports its binding information there, distinguishing each object entry with its (the object's) own object UUID.

An example of this model is the printer service that was previously described. Clients will import directly from the name entry of the resource they want to use. For this kind of application, there will generally be more namespace entries than active servers, since each server presumably manages more than one object. If the name entries have been set up correctly and the servers have properly registered the object UUIDs they created, there will be no difficulty in routing any partial binding to the correct server (namely, the server that manages the object or resource specified).

Summary of Binding Models

Although the name service allows other approaches, we recommend that whenever possible you use the object-oriented scheme to organize your namespace entries. There are at least two good reasons for doing so. First, it is easy to administer; at the simple entry level, things really are simple. Second, this is the most flexible foundation for building other more complicated access models using group entries and profiles.

The separate name entries in your namespace should contain bindings that will unambiguously resolve to specific server instances. Since interface UUIDs are often offered by more than one server, more information than just an interface UUID is needed in order to give an RPC with a partial binding the required specificity. Object UUIDs provide this extra information. When using object UUIDs to distinguish bindings in this way, servers must take care to preserve their uniqueness across name entries.

Finally, profile entries allow clients to walk through a specified search path of namespace entries and yet be completely ignorant of the actual names themselves. While name independence may not be desirable for an object-based or resource-based distributed application, it can be a powerful mechanism when used with other models.

As you are setting up the namespace organization for your application, remember that there is not a direct exact mapping from names to bound servers. Different names, once imported from, may resolve to identical bindings if the partial bindings were exported on the same interface, from the same host, and not otherwise distinguished from each other by object UUIDs. It is the application developer's responsibility to tailor an application's export and import procedures so that this mapping behaves as intended.

Models Based on Non-CDS Databases

The three models previously described are not mutually exclusive; if the namespace is set up correctly, all three can coexist at the same time. All three of the models are implemented through the functionality of the DCE RPC name service.

Although the emphasis in this discussion has been placed on the storage and retrieval of binding information, the namespace entries can be used to store additional states for objects. In order to do this, an application would have to create additional attributes on the CDS entries it intended to use because the name service recognizes only the four NSI attributes: binding, object, *group*, and **profile**.

Such additional entry attributes would be created and accessed through XDS. However, whenever you find yourself contemplating extending the name service in this manner, you should carefully consider whether the name service (and, consequently, CDS) is the best mechanism for doing what you want to do.

In the preceding example, where an object-oriented namespace containing separate entries for individual printers was described, only the identifier for the printer (the object UUID) and the binding for the server that managed it were stored in the CDS entry. Other information, such as what jobs are currently queued for the printer, who owns the jobs, and so on, was maintained by the server. This data could be stored in CDS only by creating new attributes to put it in, but it would be changing too quickly for CDS to efficiently keep up with it anyway. The performance of both the application and CDS would suffer from such an arrangement.

It is possible to imagine distributed applications whose resources (the objects they are managing) are of such a nature that they could be more efficiently managed through a private application-implemented database. Suppose the number of managed objects is very large, or that the state of the objects is volatile. It would certainly be a bad idea to try to use CDS to store this kind of information, which would be changing much more rapidly than CDS's ability to propagate the updates.

Example of a Privately Managed Database

As an example of such a privately managed database, consider a print service where jobs are submitted not to individual printers, but rather to a generic printer service. The client, **lpr**, binds (probably through a group entry) to some certain print server, and sends the job to be printed to that server, which then, after some thought, sends the job to one of the printers that it manages.

Consider, for example, what happens if a user invokes the client **cancel** sometime later to stop a job. If, for example, the original command was

```
lpr War_and_Peace.ps
```

and the subsequent request to cancel is

```
cancel War_and_Peace.ps
```

then how does the server that **cancel** binds to find the right job to delete? There is no guarantee that **cancel** will bind to the same server that happened to receive the original print request, so having each print server keep track of its own jobs would not be the answer.

One way to keep track of jobs queued would be to have a dedicated *job location server* as part of the application. Each time a print server queued a job to a printer it would record the fact (with all the pertinent details) with the location server. Whenever a job completed, the server would again notify the location server to remove its record of that job from its database. A client **cancel** then binds first to the location service, where it receives the name of the print server associated with the job it wants to cancel. It then looks up that name, binds to the right print server, and sends the cancel request. In effect, the location server has become a name service for **cancel**.

This method of organizing activity results in a split-model database. The print servers' binding information is managed through CDS, as usual, and the location server manages other more volatile information associated with those same servers.

Another way a server could maintain its own database of named objects would be by implementing a junction.

Combining Models

In designing a binding access model for an application, consider also whether it may be appropriate to combine some of the models previously discussed. In the print service application, it may be desirable for servers to also offer a management interface to specific servers rather than to specific objects; for example, **lpr**, **lpq**, and **lprm** are generic application clients, so it is appropriate for them to bind to printer objects, but if **lpr_mgmt** is supposed to manage characteristics of a whole service, then it should bind to servers.

An Object-Oriented Model with Grouped Binding Information

The following variation on the object-oriented binding model shows how the group attribute can be used in object entries. In this model, each of the object entries contains, as before, an object UUID that will uniquely identify (either to the endpoint mapper on the exporting server's machine, and/or to the server itself) the object referred to by that entry. However, the object entries do not contain any binding information. Instead, a group attribute in each object entry refers clients' import operations back to the server's own separate entry, which contains the binding information for that server.

The namespace ingredients of this model are the following:

- A single namespace entry for the server, which contains a binding attribute and, possibly, an object attribute. Thus, this entry contains all the binding information that is exported to the namespace by the server.
- One namespace entry for each object that the server offers. Each entry contains an object attribute that contains that object's UUID, and a group attribute that refers back to the exporting server's namespace entry.

Note that the object entries consist of a combination of attributes not encountered before (object and group). Although unorthodox combinations of attributes are not generally recommended, they can sometimes be useful, as in this example.

The advantages of this scheme are twofold:

- It greatly reduces the amount of server-provoked export activity into the namespace.
- It allows the server application to associate a people-readable name (that is, the name of each object's namespace entry) with a UUID.

When the server is first activated it creates all the namespace entries, exports the objects' UUIDs into the object entries, and initializes the group attributes to refer to the server entry. It exports its binding information into the server entry only. From then on, whenever it is restarted, all the server needs to do is reexport its binding information into the single server entry. Everything else remains the same; that is, the objects' UUIDs have not changed, nor has the name of the server entry to which the object entries' group attributes refer. Thus, instead of exporting bindings to every one of its object entries on subsequent startups, the server exports to only one entry.

Of course, if the system were restarted or the namespace reinitialized, then the original start-up process would have to be repeated.

The slight disadvantage of this scheme occurs on the client side, where the import process becomes somewhat more complicated than it would be if all necessary information (both binding and object UUID) could be read in from the same entry.

Server and Client Steps

The following subsections describe in detail, from both the server's and the client's side, how this model works.

Server Export

This section lists the steps that the server must perform to set up and initialize its namespace. Each step consists of the NSI function that must be called to perform the operation.

1. **uuid_create( )**

   To create an object UUID for each object that the server intends to export.

2. **rpc_server_register_if( )**

   To register interface(s) and EPVs with the RPC runtime. (This is also where manager types, if any, are registered.)

3. **rpc_server_use_all_protseqs( )**

   To request bindings from the RPC runtime for each object.

4. **rpc_server_inq_bindings( )**

   To get the binding handles for each object.

5. **rpc_ns_binding_export( )**

   To export the binding information of the objects' common server and the object UUIDs for each of the namespace objects to the server's own separate name entry. This step is performed *only once* for each collection of objects managed by the same server.

   The final three steps set up the grouped collection of service objects. Note that the next two steps are executed once for each object managed by the server:

6. **rpc_ns_binding_export( )**

   To export each object's object UUID to its own name entry. A NULL is passed as the *binding_vec* parameter to specify that only an object UUID, and no bindings are being exported.

   Note that each object UUID must be exported to *both* the object name entry and the server entry; hence the need for this export operation in addition to the operation described in Step 5 above.

7. **rpc_ns_group_mbr_add( )**

   To add the server's name entry (created in the first step) as the sole member of an NSI group attribute in each of the separate objects' name entries created in the second step.

8. **rpc_ep_register( )**

   To register each object's UUID with the server's host machine's endpoint mapper. Note that **rpc_ep_register( )** takes an object UUID vector as an argument, and generates from this all the necessary relationsips between UUIDs and bindings; thus the call is made only once.

   The point of this step is to make sure that when presented with an object UUID in an incoming RPC, the endpoint mapper can look that UUID up in its database and find an endpoint that has been registered with it. Registering the server's bindings (that is, endpoints) with all object UUIDs will accomplish this.

Step 6 above is made necessary by the way the ACL editor's binding mechanism works. (Applications gain access to the ACLs that an application maintains on its objects through the client agent **acl_edit**, which uses a standard DCE-wide interface for ACL operations.) The **acl_edit** mechanism contains code that allows it to bind to the server that implements the ACL manager responsible for the object whose ACL is desired. However, these generalized binding routines necessarily conform to certain fixed ways of doing things. If the **acl_edit** binding mechanism obtains an exported object's object UUID from the object entry, it will use that object UUID in its subsequent import through the group attribute.

Thus, the object UUID will be contained in the handle structure that the client presents to the **rpc_ns_binding_import_next( )** call, expecting it to be filled in with binding information. However, the RPC runtime always tries to match such an input object UUID with a UUID contained in the entry that the caller is trying to import from. If no matching object UUID is found, no binding information will be returned. Thus, *all* the single object UUIDs separately exported to the object entries must be exported to the server entry as well, if the exported objects are to have ACLs accessible through the **acl_edit** mechanism.

The following figure illustrates the resulting namespace arrangement.

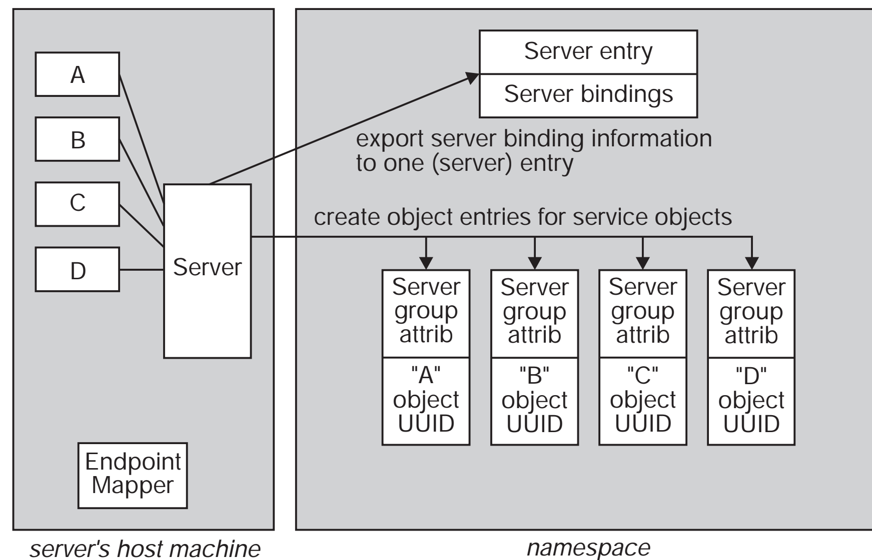


Figure 13. The Export Operation in a Model with Grouped Bindings

This generic server manages four objects, called simply A, B, C, and D. One entry is created for each of these objects, and a separate entry is created for the server itself, where the binding information is held.

The result of all this is that there is now one more namespace entry for a given service instance than there would have been with the object-oriented model discussed earlier. The group attribute in each entry is a level of indirection that allows the server to dispense with exporting many copies of the same thing.

If a directory with the proper permissions has been set up for it in the namespace by the system administrator, a server should be able to create the object entries simply by making the calls described here.

Client Import

To bind to an object managed by the server as previously described, a client performs the following series of library calls:

- **rpc_ns_entry_object_inq_begin( )**

  To set up an object inquiry context; the client application here specifies the name of the desired namespace object entry.
- **rpc_ns_entry_object_inq_next( )**

  To return the object UUID that the server exported to the object's entry.

This UUID (which will be passed to the **rpc_ns_binding_import_begin( )** routine, below) will enable the server host's endpoint mapper to accurately map the incoming remote procedure call to the server that exported this entry.

The UUID may also be used by the server itself to determine which object the client wants to access. Note that although this set of library routines is designed to accommodate schemes in which multiple object UUIDs have been exported to the same entry, the model described here requires that *only one* object UUID (the unique identifier of the object to bind to) be exported.

- **rpc_ns_entry_object_inq_done( )**

  To delete the object inquiry context.

- **rpc_ns_binding_import_begin( )**

  To set up a binding import context.

  Note that the object UUID that was returned by the call to **rpc_ns_entry_object_inq_next( )** must be passed to **rpc_ns_binding_import_begin( )**; as a result of this the import operation (**rpc_ns_binding_import_next( )**) will return only a binding with that object UUID.

  An alternative to using the binding import routines would be to use the group member inquiry (**rpc_ns_group_mbr_inq_ * ( )**) routines to learn the name of the entry referred to in the group attribute, and then to do a direct import from that entry.

  The reason for using the **rpc_ns_group_mbr_inq_ * ( )** routines, rather than the normal import functions (**rpc_ns_binding_ * ( )**), would be to make sure that the group (and not some other) attribute in the entry is read. The **rpc_ns_binding_import_next( )** routine is defined to successively exhaust the contents of an entry's

  – binding attribute

  – *group* attribute

  – **profile** attribute

  Since the model described here employs object entries with only group attributes and no binding or profile attributes, using the normal import routine should work fine.

- **rpc_ns_binding_import_next( )**

  To read the entry's group attribute.

  The name service's access to (and return of the binding handle from) the entry's group attribute is transparent and unerring because there is only one set of binding information associated with a given entry in this scheme, and that information is found only in the group attribute. Note that if there had been more than one member in the group, which in fact is generally the case when group attributes are used, then the order of return would be random. Or if there had been binding information associated with *both* attributes, then here also the order in which binding handles would be returned would be random; that is, the caller might get a handle from the simple name attribute first, and then the handles exported to the group members, or it might get one or more of the group's member's handles, then one or more of the simple entry's handles, and so on.

- **rpc_ns_binding_import_done( )**

  To delete the binding import context.

The next figure illustrates this activity.

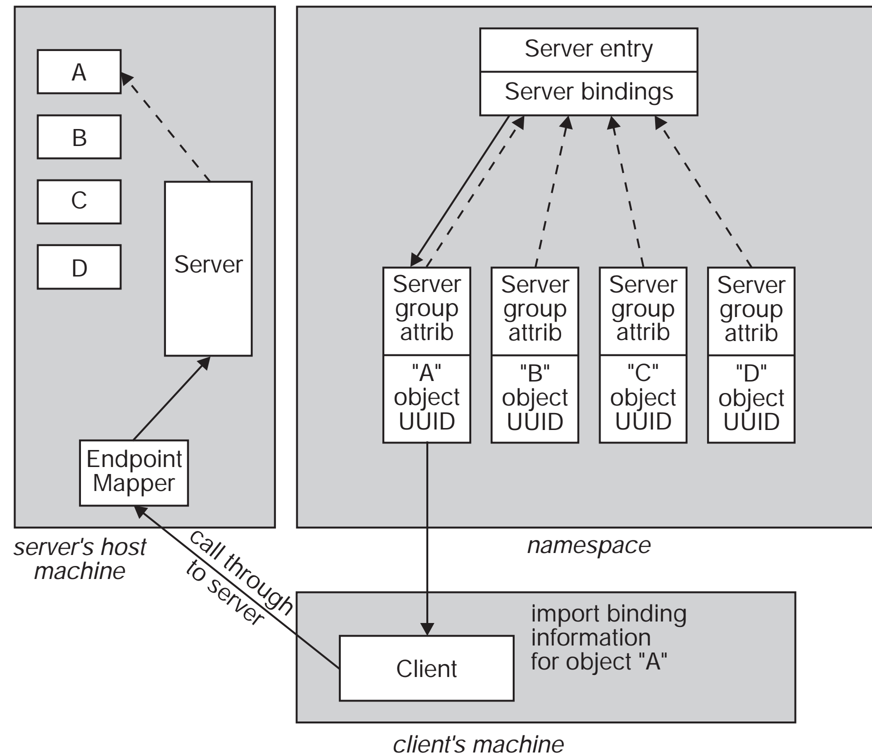


Figure 14. Importing from a Model That Uses Grouped Bindings

The client shown in the figure imports a binding for object A. This becomes (through the group attribute) a referral back to the server's entry where the bindings are held, and a binding is indirectly imported from the server entry. The object UUID for A is read, in a separate operation, directly from the object's entry. With this information in its binding handle, the client makes its first remote call through the server's interface. The call finds its way to the endpoint mapper via the partial binding information, and the endpoint mapper completes the binding by looking up the object UUID, which was registered there by the server.

Global Organization of the Namespace

Since DCE is designed to support very large namespaces, it uses a hierarchical service for binding. The global scale is separated into cells whose boundaries are administratively defined. For example, a company using DCE might have a cell containing its employees and local services. The cell namespace administrator could decide to put all the service entries in a single directory if the cell were small.

Both the import and export name service operations support default values derived from environment variables; for example, **RPC_DEFAULT_ENTRY_NAME**. The environment variables can be set by start-up files to the name of a well-known directory within the cell. The only remaining decision then will be how to name the actual entries within the directory. One easy method is to use mnemonic names, or names of interfaces such as **binop**, **spm_library**, and so on. If these entries are only being accessed by clients through profiles, their names will not be directly visible to the client anyway.

But now imagine a larger organization. The administrator will want to define some naming hierarchy based on geography, organization, or other criteria. Somewhere

within this hierarchy some writable directories (or parent directories) would be created, which could contain server entries, profiles, and so on. If clients are using only profiles to access bindings, then this organization will still be transparent to them. If clients want to bind to specific servers or objects, then more attention must be paid to the names given the servers' or objects' entries. The names should in some way reflect the organization, geography, or other relevant aspects of the server or object.

In summary, the important points to keep in mind are the following:

- The model should be appropriate for the organization and permit efficient administration of the namespace.
- There should be simple guidelines for naming objects and services so that users have a good chance of guessing the right answer.

Chapter 6. RPC Parameters

The RPC mechanism attempts to provide a data model as close as possible to the familiar local call model. For example, you can pass data by reference—by passing a pointer to a data item—despite the fact that client and server do not share an address space. Nevertheless, there are significant differences in both the syntax and semantics of RPC parameter data compared with C language local call data. For example, RPC provides directional attributes, conformant arrays, discriminated unions, and pipes, constructs which have no equivalents in C. Each requires an IDL specific syntax and has new semantics. Also, familiar constructs, such as pointers, closely mimic their local C language counterparts, but nevertheless must behave differently in some circumstances.

The DCE RPC programmer is thus confronted with a number of unfamiliar style and policy issues. The policy issues have mainly to do with which data types to use in given circumstances: for example, would you be better off using an array or a pipe to transfer a large block of data? This chapter contains recommendations that should help you make such choices. The style issues arise from the rich and unfamiliar syntax for RPC parameters which can make the mechanics of using many of the RPC data types seem rather daunting. This chapter contains numerous examples of basic data passing styles.

Execution Semantics

Before we begin to discuss the RPC data types themselves, a slight digression is necessary. Whatever data you pass, all RPCs must deal with the unreliable nature of remote network connections. A call may not complete due to a network failure, possibly leaving the call operations in an indeterminate state. For this reason, the IDL provides execution semantics attributes that applications can use to request certain (limited) guarantees about call completness.

Ideally, in order for an application to behave in a determinate fashion, each operation needs to be invoked exactly once each time it is invoked. This requirement can be relaxed somewhat for idempotent operations: those which have the same effect when they are invoked one or more times. In this case, an application can settle for at-least-once semantics.

Unfortunately, with a remote procedure call, there is no way to guarantee either exactly once or at-least-once call semantics. Instead, RPC provides **at-most-once** and **idempotent** semantics. When a call completes and returns to the client, then at-most-once semantics is equivalent to exactly-once semantics, and idempotent semantics is equivalent to at-least-once. When a call fails to return to the client—either because of a server or communications failure—the semantics make the following guarantees:

at-most-once
: The call was invoked on the server either 0 or 1 times. If the call was invoked, it may or may not have completed execution.

idempotent
: The call was invoked on the server 0 or more times. If the call was invoked, it may or may not have completed execution for any invocation.

In reality, idempotent semantics provides no guarantee for calls that fail to return to the client. In fact, DCE provides no guarantee about how idempotent semantics are

actually implemented. It is perfectly correct to implement idempotency by using at-most-once semantics, and depending on protocol and implementation, this may be the case. Idempotent semantics is therefore really a hint from the application that a call is a candidate to be retried if the implementation uses a retry strategy.

These characteristics lead to two kinds of policy guidelines for call semantics. The first has to do with the behavior required of idempotent operations. An operation is a good candidate for idempotent semantics if it either changes no state on the server (such as, a read operation), or if the server state will be the same even if the same call is invoked more than once (such as, a call that writes the same record) with the same **[in]** data. Note that in either of these cases, the result returned by a call may not be the same on each retry, since some other thread or process may have modified server state. A server that allows simultaneous reads and writes provides a good example. However, the runtime does guarantee commutativity of operations on the same association: an idempotent call will not be retried if a later call on the same association has been invoked.

The second policy issue has to do with how applications respond to call failures. The issues are the same for idempotent and at-most-once calls. In neither case can the client know whether the server manager operation was invoked, and, if it was invoked, whether it was completed. This leads to three possible failure states:

1. The manager operation was not invoked.
2. The manager operation was invoked but did not complete.
3. The manager operation was invoked and completed, but failed to return to the client.

The burden of determining which state applies, and implementing recovery actions rests almost entirely with the application. The RPC mechanism provides limited support for cleanup in the case of applications that use context handles to maintain state between calls. Application provided context rundown routines will be called on behalf of the application if a communications failure is detected. Beyond this rather elementary mechanism, DCE RPC does not provide any internal support for transaction processing, roll-back, or other recovery mechanisms. For applications where error recovery and maintainance of consistent state is essential, these must be implemented by the application programmer. The topic is beyond the realm of this policy guide.

IDL also provides two execution semantic attributes of somewhat more limited use: **broadcast** and **maybe**. Broadcast semantics may be used with connectionless transports when there are multiple servers on the local network that can handle a call. The client broadcasts the call request to all servers, and completes the call with one of them. Maybe semantics provides a calling style that may be used when a call has no **[out]** or **[in, out]** parameters. The call is attempted once, and no response is returned. Both **broadcast** and **maybe** semantics implictly require that the operation be idempotent.

Parameter Semantics

RPC calls and the RPC API specify directional attributes for their parameters, even though such attributes are not formally supported by C. As a general rule, an **[in]** parameter is one that must be passed with a meaningful value and an **[out]** parameter is one whose value will be changed by the call. An **[in,out]** parameter is therefore one which must have a meaningful value on input and which may be changed on output.

The following table summarizes parameter semantics:

Table 6. Parameter Semantics

Semantics	Meaningful value input	Changed on output
[in]	yes	no
[out]	no	yes
[in,out]	yes	yes

An **[out]** or(**[in,out]**) parameter is one whose value is changed by the call, so it must be passed by reference, that is, as a pointer to the datum of interest. RPCs and the RPC APIs therefore always specify output parameters as pointers. The address passed must always point to valid storage. For example, the ubiquitous *status* parameter may be declared in the IDL as follows:

```
[out] error_status_t *status
```

The application code then needs to declare a variable such as the following, and pass it as **&st** to each RPC:

```
error_status_t st;
```

When a call allocates storage for an output parameter, it is declared as a pointer to a pointer. For example:

```
rpc_binding_vector_t  **binding_vector
```

The application follows the same rule as in the **status** case, declaring a variable such as the following, and then passing this as **&binding_vec**:

```
rpc_binding_vector_t *binding_vec
```

This obeys all the rules for output parameters: the address passed to the call points to valid storage, but the contents of that storage need not contain a meaningful value (in this case, need not be a valid pointer). A simple rule of thumb for output parameters is to declare a variable with one less asterisk than contained in the IDL (or RPC API) declaration and pass its address when calling the operation.

Parameter Memory Management

RPC attempts to extend local procedure call parameter memory management semantics to a situation in which the calling and called procedure no longer share the same memory space. In effect, parameter memory has to be allocated twice, once on the client side, once on the server side. Stubs do as much of the extra allocation work as possible so that the complexities of parameter allocation are transparent to applications. In some cases, however, applications may have to manage parameter memory in a way that differs from the usual local procedure call semantics.

For the purposes of memory allocation, three classes of parameters must to be considered:

- Nonpointer types
- Reference pointers
- Full pointers

For all types, the client application supplies parameters to the client stub, which marshals them for transmission to the server. The client application is entirely responsible for managing the memory occupied by the passed parameters. On the server side, the server stub allocates and frees all memory required for the received parameters themselves.

In the case of the pointer types, however, the application and stubs must manage memory not only for the parameters themselves, but also for the pointed-to nodes. In this case, the memory management requirements depend both on the pointer type and on the parameter's directional attributes.

The rules are described in the following sections.

Client Side Allocation

in parameters
: For all pointer types, the client application must allocate memory for the pointed-to nodes.

out parameters
: For reference pointers, the client application must allocate memory for the pointed-to nodes, unless the pointer is part of a data structure created by server manager code. For parameters containing full pointers, the stub allocates memory for the pointed-to nodes.

in, out parameters
: For reference pointers, the client application must allocate memory for the pointed-to nodes. For full pointers, on making the call, the client application must allocate memory for the pointed-to node. On return, the stub keeps track of whether each parameter is the original full pointer passed by the client, or a new pointer allocated by the server. If a pointer is unchanged, the returned data overwrites the existing pointed-to node. If a pointer is new, the stub allocates memory for the pointed-to node. When a parameter contains pointers, such as an element in a linked list, the stub keeps track of the chain of references, allocating nodes as necessary.

It is the client application's responsibility to free any memory allocated by the stub for new nodes. Clients can call the routine **rpc_sm_client_free( )** for this purpose.

If the server deletes or eliminates a reference to a pointed to node, an *orphaned* node may be created on the client side. It is the client application's responsibility to keep track of memory that it has allocated for pointed-to nodes and to deal with any nodes for which the server no longer has references.

Server Side Allocation

in parameters
: For all pointer types, the stub manages all memory for pointed-to nodes.

out parameters
: For reference pointers, the stub allocates memory for the pointed-to nodes as long as the size of the targets can be determined at compile time. When the manager routine is entered, such reference pointers point to valid storage. For parameters that contain full pointers, the server manager code must allocate memory for pointed-to nodes. Servers can call the routine **rpc_sm_allocate( )** for this purpose.

in, out parameters
For reference pointers, the stub allocates memory for pointed-to nodes if either the size of the pointed to nodes can be determined at compile time or the reference pointers point to values received from the client. When the manager routine is entered, such reference pointers point to valid storage. For full pointers, the stub allocates memory for the original pointed-to nodes. The server manager code must allocate memory if it creates new references. Servers can call the routine **rpc_sm_allocate( )** for this purpose.

The server stub automatically frees all memory allocated with **rpc_sm_allocate( )**.

RPC Data Types

IDL provides both a number of primitive data types—such as various sizes of integers and floats, bytes, and booleans—as well as pointers and a variety of constructed types based on the primitive types. The use of the primitive types is quite straightforward. The only important policy issues have to do with IDL data type to C data type mappings and with character handling. Pointers and the constructed types raise many more policy and style issues, and the bulk of this chapter is devoted to describing them.

IDL to C Type Mappings

Many of the primitive C data types represent items of different sizes on different machines. For example, an **int** may be 16 bits on one machine and 32 bits on another. These ambiguities can cause portability problems for some C programs, and they are intolerable for RPC programs. A parameter to an RPC call must represent the same size data item on both the client and server machine, whatever the machine architectures.

This means that when IDL declarations are compiled to generate C language headers and stubs, a given IDL type must always be declared in the corresponding C code as a C type of a specific length, no matter what machine the IDL compilation is done on.

To achieve this, the following must be true:

1. Each IDL primitive type is always represented in the generated C files, by a specific defined C type
2. Each of the specific defined C types is defined by the local implementation of DCE so that it represents a data type of the correct length.

For example, a parameter declared in the IDL as a **short**, will be declared in the IDL generated header file as the defined type **idl_short_int**. Each implementation of DCE then defines the **idl_shor_int** type correctly for the local C compiler and machine architecture to be an integer 16 bits long. For example, on a 32-bit machine, the **idl_short_int** type is typically defined as a **short int**.

When you write application code that refers to a parameter declared in the IDL, you must use a type that declares a data item of the same length. The safest policy is to use the same specific defined C type used in the headers and stubs. For example, your IDL file might declare the following:

```
void my_op([in,out] short var);
```

In this case, your server manager code would contain an function that looks something like this:

```
void my_op(idl_short_int var)
{
        .
        .
        .
}
```

On a 32-bit machine, your code could probably use a **short** safely (since that is how your implementation probably defines **idl_short_int**, but such usage is not portable to other machine types and is therefore not recommended.

The following table shows the IDL to C type mappings for the IDL primitive types.

Table 7. IDL/NDR/C Type Mappings: Part 1

IDL Type	NDR Type	Defined C Type	C Type
boolean	**boolean**	**idl_boolean**	**unsigned char**
char	**character**	**idl_char**	**unsigned char**
byte	**uninterpreted octet**	**idl_byte**	**unsigned char**
small	**small**	**idl_small_int**	**char**
short	**short**	**idl_short_int**	**short int**
long	**long**	**idl_long_int**	**long int**
hyper	**hyper**	**idl_hyper_int**	16-Bit or 32-Bit Machines: Big Endian: **struct { long high; unsigned long low; }** Little Endian: **struct { unsigned long low; long high; }** 64-Bit Machines: **long**
unsigned small	**unsigned small**	**idl_usmall_int**	**unsigned char**
unsigned short	**unsigned short**	**idl_ushort_int**	**unsigned short int**
unsigned long	**unsigned long**	**idl_ulong_int**	**unsigned long int**
unsigned hyper	**unsigned hyper**	**idl_uhyper_int**	16-Bit or 32-Bit Machines: Big Endian: **struct { unsigned long high; unsigned long low; }** Little Endian: **struct { unsigned long low; unsigned long high; }** 64-Bit Machines: **unsigned long**

Table 8. IDL/NDR/C Type Mappings: Part 2

IDL Type	NDR Type	Defined C Type	C Type
float	**float**	**idl_float**	**float**
double	**double**	**idl_double**	**double**
handle_t	**not transmitted**	**handle_t**	**void ***

Table 8. IDL/NDR/C Type Mappings: Part 2 (continued)

error_status_t	**unsigned long**	**idl_ulong_int**	**unsigned long int**
ISO_LATIN_1	**uninterpreted octet**	**ISO_LATIN_1**	**byte**
ISO_MULTI_LINGUAL	**(Note 1.)**	**ISO_MULTI_LINGUAL**	**struct{ byte row; byte column; }**
ISO_UCS	**(Note 1.)**	**ISO_UCS**	**struct{ byte group; byte plane; byte row; byte column; }**

In addition to the IDL primitive type mappings defined in the table, implementations provide a set of convenient typedefs that map the listed defined types into types that explicitly name amounts of storage. These are defined in IDL as follows:

```
typedef unsigned small  unsigned8;       /* positive 8-bit integer */
typedef unsigned short  unsigned16;      /* positive 16-bit integer */
typedef unsigned long   unsigned32;      /* positive 32-bit integer */
typedef small           signed8;         /* signed 8-bit integer    */
typedef short           signed16;        /* signed 16-bit integer   */
typedef long            signed32;        /* signed 32-bit integer   */
typedef unsigned32      boolean32;   /* a 32-bit boolean       */
```

They are defined in C as follows:

```
typedef idl_usmall_int unsigned8;
typedef idl_ushort_int unsigned16;
typedef idl_ulong_int unsigned32;
typedef idl_small_int signed8;
typedef idl_short_int signed16;
typedef idl_long_int signed32;
typedef unsigned32 boolean32;
```

As a matter of programming style, these types have the advantage that the size of the declared data items is explicitly stated. For this reason their use in both IDL declarations and application C code is recommended. Note also that the following IDL and C typedefs are also made available by implementations a convenient portable declaration for status parameters:

```
typedef unsigned long   error_status_t;

typedef idl_ulong_int error_status_t;
```

Character Handling

When passed as an RPC paramter, the IDL **char** type is automatically subject to ASCII-EBCDIC conversion, depending on the character encodings used by the client and server machines. Therefore, the contents of a **char** type may not be the same for the sender and receiver. This allows clients and servers to maintain the same semantics when passing characters between machines that use different encodings. For example, the character **a** is represented by a byte with the value 61h on an ASCII machine, and a byte with the value 81h on an EBCDIC machine. RPC automatically makes the conversion so that a character parameter that prints as **a** on the client machine also prints as **a** when recieved by the server.

However, if what your application really intends is to pass a byte with the value 61h from client to server, such translation is clearly not what you want. To avoid this potential problem, when passing byte data with noncharacter semantics, use the IDL **byte** type.

Also note that IDL provides three international character types for use with nonASCII, nonEBCDIC character sets: **ISO_LATIN_1**, **ISO_MULTI_LINGUAL**, and **ISO_UCS**. To ensure portability, your application should use these types to declare character data in one of these sets.

Pointers

RPC pointers differ from local pointers in one key respect: there is no shared address space between client and server. This means that the stubs need to marshall the pointed-to data itself. To do so, the stubs must be able to dereference any pointer passed as a parameter. This means that a pointer, *even if it does not point at useful data*, must be initialized either to NULL or to a valid address before it is passed as a parameter. This behavior may be counter-intuitive for programmers used to local procedure calls, where pointers may be freely passed whether they have been initialized or not, and is a common source of programming grief for remote procedure calls.

To be able to marshall pointer referents, the stubs need to know, either at compile time or at runtime, how much data to transmit; that is, they need to know the size of the pointed to object. This can require a good deal of work on the part of the stubs in the case of varying or conformant arrays and objects like linked lists.

One effect of this is that pointers only reference the marshalled data itself; that is, data of the size determined by the stub. For example, passing an **idl_char *** parameter causes the stub to marshall a single **idl_char**, since that is the size of the object pointed to by an **idl_char ***. Typically, a local procedure call passes a **char *** type in order to pass the address of an array of characters, not a single **char**; but a remote routine that tries to move such a pointer beyond the transmitted **char** will very likely find itself pointing to invalid storage and certainly not to the intended string.

A similar case is illustrated in the sample code: a client passes an array and an **[in, out, ptr]** pointer to an array element. If the server sets the pointer to point to some element of the passed array, then it will point to memory holding a copy of that element when the call returns to the client. It will not point to any part of the passed-in array itself, and any attempt to increment or decrement the pointer on the client side will leave it pointing to an invalid location.

This is one example of the fact that you cannot assume that the results of pointer arithmetic will be the same for a local and remote procedure call. To give another example, suppose a call passes two parameters: a data structure and a pointer to the type of the data structure, set to NULL. If the server application then sets the pointer to point to the data structure, the client stub will allocate new storage for the returned data structure and set the returned pointer to point to it. As a result, the returned pointer will not point to the original structure, but to a copy of it in stub maintained memory.

This may seem like an IDL limitation, but in fact, the real issue is that the client and server address spaces are different, and some operations in one address space cannot be reflected in the other. Specifically, the server application cannot meaningfully interpret an address in the client address space, and vice versa. So, as in the last example, the server cannot set a pointer to point to a structure in the client address space; it can only ask the client stub to mirror any changes made at the server.

Memory Allocation Routines

The stubs will do their best to allocate any new memory required for marshalled pointed-to nodes so that the marshalling is transparent to the application. On the server side, stub allocated memory exists for the scope of the manager routine call. The stub frees such memory once the nodes have been marshalled. On the client side, however, the stubs obviously cannot free the memory they have marshalled since they are returning the data to the client application. Therefore, in order to avoid memory leaks, when a client makes an RPC that results in the client stub allocating memory, the client application needs to call **rpc_sm_client_free( )** to free the pointed-to memory.

When a server manager routine needs to allocate new memory for a pointed-to node, it can do so either statically or by making a call to a memory allocation routine. In the latter case, however, the manager cannot deallocate the memory it has allocated, since the pointer must be valid when the call returns (so that the stubs can marshall the data.) Only the stub can free such memory. In order to permit this, server managers need to call **rpc_sm_allocate( )** to allocate memory for parameters. The stubs free all memory allocated by **rpc_sm_allocate( )** once they have marshalled the required data, thus avoiding memory leaks.

Pointer Types

For reasons of efficiency, IDL distinguishes between reference **[ref]**, full **[ptr]**, and unique **[unique]** pointers. As we saw above, even though pointers are used by applications to pass data by reference, the lack of shared address space means that the stubs have to pass the data by value and provide the receiver with a reference to the passed data.

In the simplest case, a pointer always points to the same memory: that is, its value does not change. In such a case the stubs always marshall the passed value from and to the same memory location on the sender and receiver respectively. This style of marshalling is provided by **[ref]** pointers.

When the value of a pointer changes during a call, the stubs have a more complex task. Suppose, for example, an **[in, out]** pointer is NULL before an RPC and is set by the server application to point to some data structure allocated by the server. As in the **[ref]** pointer case, the server stub needs to marshall the (new) referent and the client to unmarshall it, but the client stub also needs to do two more things: it needs to allocate space for the unmarshalled referent, and it needs to point the previously NULL pointer to it. Similarly, for a pointer that initially points to one memory location and is changed during an RPC to point to another, the client stub needs to allocate new memory to hold the unmarshalled value of the new referent and to change the pointer value accordingly. Not all of the extra work is confined to the client stub either. Obviously, the client stub needs to find out that the value of the pointer has changed, so the server has to marshall, and the RPC protocol to transmit, extra data to indicate this. This style of marshalling is provided by full (**[ptr]**) pointers, and it obviously requires more overhead than reference pointer marshalling.

Unique pointers provide for an intermediate case: a pointer that always points either to a single memory location or is NULL. Such a pointer may change from NULL to a nonnull value or from a nonnull value to NULL, but never has more than one nonnull value. Such a pointer is marshalled more efficiently than a full pointer, but not as efficiently as a reference pointer.

Applications should consider the **[ref]** and **[unique]** pointer types as optimizations. A full **[ptr]** pointer can always be used. The **[ref]** and **[unique]** pointer types may be used whenever the application is guaranteed to meet the restrictive conditions under which these types work.

As a guide to using the pointer types, there are a few general rules and a number of special cases, having mainly to do with embedded pointers and data of variable size. The rules are as follows:

- In passing parameters, you need to distinguish carefully between top-level and lower-level pointers. A top-level pointer is a pointer passed as an argument to a call. A lower-level pointer is one contained in the referent of a top-level pointer. The directional semantics **[out]** and **[in, out]** both require parameters to be passed by reference and hence always require a top-level pointer.

  The model is, essentially, that the client provides a container into which the returned value is written. In the **[out]** parameter case, the contents of the container are assumed to be unimportant on input and are not marshaled by the client stub. In the **[in, out]** case, the contents are assumed to be meaningful and are passed to the server.

  The top-level pointer is thus the address of the parameter container, and obviously, this value should not change during the course of the call. If it did, the return value would be written to some undetermined place in the client address space. Hence, the top level of **[out]** and **[in, out]** parameters have reference pointer semantics. The IDL compiler enforces this for **[out]** parameters by permitting only the **[ref]** attribute. It does not force this for **[in, out]** parameters, but the behavior is exactly the same. Remember, the actual parameters of an RPC call are always passed by value: hence a call cannot change the value of a top-level pointer. It can only change the value of something passed by reference.
- To pass an **[in]** parameter by reference, you can pass its address as a pointer of either style. The server stub will allocate and deallocate the required memory for the pointer referent. Since an **[in]** pointer has no reason to change its value, it is at least slightly more efficient to use a reference pointer in this case.
- Since **[out]** semantics do not consider the contents of such storage to be meaningful, an **[out]** parameter is not marshalled on the call. The server stub will allocate memory to hold the referent as long as the size of the referent is known at compile time. The stub obviously cannot allocate memory for referents whose size is determined arbitrarily by the the server application. For such parameters (such as linked lists) the server application must allocate space.

  One tricky case to consider is a linked list. The server stub allocates space for the head element, since it knows the size of such an element. The server manager then allocates space for the remaining elements and marshalls them back to the client. The client stub will allocate all necessary space for the server-created receive parameters.

  A server-created structure may contain reference pointers which the server may then set to point to objects it also allocates. All of this will be mirrored by the client stub. Note that this does not violate the rules for reference pointers, since the contained pointers do not change value during the call; they are created by the server application and passed back to the client exactly the same way that top-level reference pointers are created by clients and passed to servers.

  When an application wishes to have the callee allocate space for an **[out]** parameter, it needs to use two levels of indirection: a reference pointer to a full pointer to the data structure to be allocated. The client allocates the full pointer, setting it to NULL, and passes its address to the call. The server application then allocates the data structure and sets the full pointer to point to it. The client stub will then allocate space for the data structure on the return.

- An **[in, out]** reference pointer behaves exactly like the **[out]** reference pointer case, except that the server stub may be able to allocate space for a referent even if its size is not known at compile time. This will be the case when the client application creates an instance of a variable sized referent, such as a linked list. In such a case, the server stub will allocate sufficient space for the referent supplied by the client.
- You must take care when a server deallocates a **[ptr]** pointer referent. For an **[in, out]** parameter, the client-side stub does not deallocate the client-side referent, but the application should treat the referent as undefined, as if, in effect, the deallocated pointer referent had been unmarshalled by the client stub. By default, in the case of an **[in]** parameter, the value of the pointer referent remains unchanged on the client side. However, this default behavior can be modified by applying the **[reflect_deletions]** attribute to the operation. In this case, the client-side stub will deallocate the pointer referent. The client and server must use the **rpc_sm_** * routines to allocate and free memory for this reflection of deletions to work.
- For an **[in, out]** parameter which is a **[ptr]** pointer, if the server sets the parameter value to NULL, the client will no longer be able to dereference the pointer on return. If the client has no other means to reference the original pointed-to node, the node is said to be *orphaned*: the client will be unable to free it.

Pointer Examples

The following sample code demonstrates the basic properties of pointers. The first example demonstrates pointer arithmetic and how changes in the server address space can be reflected back to the client using full pointers. In the **.idl** file we declare a type that is an array of three integers, and a type that is a pointer to an integer. The operation takes the array as an **[in]** parameter and the pointer as an **[in, out]** parameter.

```
const unsigned32 ARRAY_SIZE = 3;

typedef unsigned32 num_array[ARRAY_SIZE];
typedef [ptr] unsigned32 *num_ptr;

void ptr_test1(
  [in] handle_t handle,
  [in, out] num_ptr *client_ptr,
  [in ] num_array client_array,
  [out] error_status_t *status
);
```

The server manager code to implement this points the client pointer to the beginning of the array and then increments it once:

```
void
ptr_test1(
    handle_t h,
    num_ptr *client_ptr,
    num_array client_array,
    error_status_t *status
)
{
    *status = 0;
    *client_ptr = client_array;
    ++(*client_ptr);
}
```

On return, the client's version of the pointer will point to memory that holds the second element of the array. It will not point to the array itself, however. The client code demonstrates this:

```
num_ptr client_ptr = NULL;
num_array client_array = {25, 50, 75};

ptr_test1(binding_h, &client_ptr, client_array, &status);

/*
 * The test function pointed the client pointer to the
 * second array element.  On return, this points to memory
 * that holds this value.
 */

printf("Client pointer points to %i", *client_ptr);

/* However, if we now increment the pointer, it
 * points to unintialized memory.  This shows the
 * limits of mirroring.
 * *** WARNING: You may dump core here !! ***
 */

client_ptr++;
printf("Client pointer now points to %i", *client_ptr);
```

What happens here is that the client stub allocates space for the new referent of **client_ptr** when the call returns. This space now holds the value in the second element of the array. The pointer no longer points to the original array but to this newly allocated space. You can see this clearly when the client attempts to increment the pointer. Instead of pointing to the third element of the array, it points to some undetermined place in memory, and the client may fail when it tries to dereference the pointer.

As an exercise, you could change the code to declare a pointer to the **num_array** defined type rather than to an integer. Then you could have the server manager point this to the input array and return it without incrementing the pointer. The returned pointer will now reference a copy of the original client array with all its elements. It will not, however reference the original array itself.

The second pointer example illustrates passing a linked list. The **.idl** declaration is as follows:

```
typedef struct link {
  unsigned32 value;
  [ptr] struct link *next;
} link_t;

void ptr_test2(
  [in] handle_t handle,
  [in, out, ref] link_t *head,
  [out] error_status_t *status
);
```

The server manager code is as follows:

```
void
ptr_test2(
    handle_t handle,
    link_t *head,
    error_status_t *status
)
```

```
{
    link_t *element;

    if (head)
    {
        element = head;
        while (element->next)
            element = element->next;
        /* Add another element to the list... */
        element->next = (link_t*) rpc_sm_allocate(sizeof(link_t), status);
        element->next->value = element->value * 2;
        element->next->next = NULL;
    }
    *status = error_status_ok;
};
```

The manager operation adds a new element to the end of the linked list. Note that the *head* parameter has **[in, out]** semantics here: we must pass in a pointer to a valid element. (The next example shows how to implement an *[out]* parameter that is allocated by the operation.)

In this and the following example, we use **rpc_sm_allocate( )** to allocate data on the server side. This gives the semantics you probably want for a dynamically allocated referent for a pointer parameter: on return, the data is automatically deallocated on the server, and further manager operations that access this data do so via a pointer parameter passed by the client. Memory leaks on the server are thus avoided.

An application must be very cautious if it attempts to use pointer parameters in a way that contradicts such semantics: for example, by returning a pointer to static global storage on the server. In such a case, the server and client versions of such storage can easily become inconsistent. A context handle, which the client must not modify, is typically what you want in such a case.

The client code for the linked list test is as follows:

```
link_t first, *element;
int i;
first.value = 2;
first.next = NULL;

for (i = 0; i < 8; i++)
    ptr_test2(binding_h, &first, &status);

element = &first;
while (element->next)
{
    printf("%i", ", element->value);
    element = element->next;
}
printf("%i", element->value);
```

The client passes in the head element, and then calls the server several times to add more elements to the list. Finally, the client prints out the list.

The next pointer example illustrates how the stubs automatically allocate memory for an **[out]** parameter. The client application allocates a NULL pointer to the data structure of interest and passes the address of this pointer as the **[out]** parameter. The server manager allocates a structure, and on return the client stub allocates it too, automatically.

The **.idl** declaration is as follows:

```
typedef struct {
    [ref] unsigned32 *value;
} number;

typedef [ptr] number *number_ptr;

void ptr_test3(
    [in] handle_t handle,
    [out, ref]  number_ptr *client_ptr,
    [out] error_status_t *status
);
```

The server manager operation is then as follows:

```
void
ptr_test3(
    handle_t handle,
    number_ptr *client_ptr,
    error_status_t *status
)
{
    number_ptr nptr;
    unsigned32 *nval;
    nptr = (number_ptr) rpc_sm_allocate(sizeof(number), status);
    nval = (unsigned32 *) rpc_sm_allocate(sizeof(unsigned32), status);
    *nval = 256;
    nptr->value = nval;
    *client_ptr = nptr;
    *status = error_status_ok;
};
```

The client test code looks like this:

```
number_ptr client_ptr = NULL;

ptr_test3(binding_h, &client_ptr, &status);
printf("Value = %i", (unsigned32)* (client_ptr->value));
```

Note the use of **[ref]** pointers here. The top-level **[ref]** pointer (the one passed as a parameter to the call) must point to valid storage when the call is made even though the pointer is not marshalled when the call is made. This follows the rules for **[ref]** pointers: they may not be NULL and may not change value during a call. The returned structure also contains a **[ref]** pointer, and the client stub does automatically allocate space for its referent when the call returns. This is an exception to the rule that an **[out] [ref]** pointer must point to valid storage when the call is made. In this case, the pointer is embedded in a structure which is created by the server. As long as the top-level pointer points to valid storage (to hold the returned structure), the client stub will allocate space for the referents of any newly-created **[ref]** pointers that it contains.

The final example illustrates node deletion. The **.idl** declaration is as follows:

```
[reflect_deletions] void ptr_test4(
                    [in] handle_t handle,
                    [in, out, ptr] unsigned32 *number,
                    [out] error_status_t *status);
```

The server code to implement this operation frees the memory pointed to by the input pointer and returns the pointer:

```
void
ptr_test4(
     handle_t h,
     unsigned32 *number,
     error_status_t *status
)
{
     *number = 32;
     rpc_sm_free(number, status);
     *status = error_status_ok;
};
```

The client code is as follows:

```
unsigned32 *num;

rpc_sm_enable_allocate(&status);
num = (unsigned32*) rpc_sm_allocate(sizeof(unsigned32), &status);
*num = 64;
ptr_test4(binding_h, num,
&status);
```

There are so many ways to use (and misuse) IDL pointers that it would be impossible to give a complete set of examples. The section on arrays contains more pointer examples.

Context Handles

Context handle semantics vary according to the application role. On the server side, the semantics are those of a full pointer. To the client application, a context handle has similar semantics to a fully bound server binding handle, except that the client may not perform any operations to modify it. To the client it represents a binding to context maintained by a specific a server instance. Because the context handle may also specify an object UUID, it may also bind to a specific type manager in the server instance; that is, a context handle refers to context maintained by a specific type manager in a specific server instance. It is valid over a series of calls within this scope. To enforce this, a context handle is intended to be passed as an explicit binding parameter for each operation that refers to the maintained context. Any attempt to use a context handle outside this scope will fail. Context handles are described in more detail in the *OSF DCE Application Development Guide—Core Components*.

Arrays

Array parameters provide an efficient way to pass contiguous blocks of data with little application overhead. The stubs take care of serializing and reassembling the passed data transparently to the application. When an application is interested in passing an entire buffer or some contiguous portion of a buffer synchronously—so that all of the data is made available to the receiver at the same time—arrays provide the most efficient mechanism. Pipes provide no advantage unless the data is to be processed asynchronously.

Arrays may be passed as RPC parameters, but, as in the case of other RPC data, the stubs need to know the size of data to be marshalled. The simple solution is to declare arrays of fixed size in the IDL. This can be inefficient however, since array sizes may vary at runtime, and since not all data in an array may need to be passed on every call. Therefore, IDL provides a variety of field attributes (**max_is**,

min_is, **size_is**, **last_is**, **first_is**, and **length_is**) to permit the size and bounds of the marshalled data to be determined at runtime. Note that passing a pointer to an array is not any more efficient as a way to deal with the problem of varying array sizes. Remember that marshalling a pointer requires marshalling the pointer's referent, so the array data will be marshalled anyway. Note also that the IDL language does not permit declaring a pointer to a varying array.

The size of the array data marshalled is determined in one of two ways. In a conformant array, the size of the array is not declared in the IDL declaration, and one of the **max_is** or **size_is** attributes is used to determine the size of the marshalled data at runtime. In a varying array, the size of the array is declared in the **.idl** file, but one or more of the other field attributes determines what range of elements is actually marshalled. Arrays may be both conformant and varying at the same time.

Each field attribute is associated with some variable whose value is known at runtime. The scope of this association is within either an operation declaration or a structure declaration. That is, when the array is a parameter of an operation, the field attribute variables must also be parameters of the same operation. Similarly, when the array is a member of a structure, the field attribute variables must be members of the same structure.

The following samples show a series of array declarations using some of the many possible forms:

```
/* An array of fixed size */

typedef char char5array[5];
typedef char5array *char5ptr;

void array_test1(
    [in] handle_t handle,
    [in] char5ptr a_pointer,
    [out] error_status_t *status);

/* A conformant array: the size is determined at runtime */

void array_test2(
    [in] handle_t handle,
    [in] unsigned32 size,
    [in, size_is(size)] char an_array[],
    [out] error_status_t *status);

/*
 * A varying array: the portion of the array transmitted is
 * determined at runtime
 */

typedef struct{
    unsigned32 first;
    unsigned32 length;
    [first_is(first), length_is(length)] char array[0..10];
}v_struct;

void array_test3(
    [in] handle_t handle,
    [in] v_struct v_array,
    [out] error_status_t *status);

/*
 * A conformant and varying array: both size and the portion
 * transmitted are determined at runtime
```

```
 */

typedef struct{
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
    [size_is(size), first_is(first), length_is(length)] char array[0..*];
}cv_struct;

void array_test4(
    [in] handle_t handle,
    [in] cv_struct *cv_array,
    [out] error_status_t *status);
```

The examples show clearly how field attribute variables are related to array declarations.

In the second operation declaration, a conformant array is declared as an operation parameter (*an_array*), so that the field attribute variable (*size*) must also be a parameter of the interface. In the third and fourth operations, varying and conformant-varying arrays are declared within structures, so that the field attribute variables (*size*, *first*, and *length*) must also be members of the same structures.

The server manager sample code to test these declarations is as follows:

```
void array_test1(
    handle_t handle,
    char5ptr a_pointer,
    error_status_t *status
)
{
    printf("Array test 1");
    printf("%c %c ", (*a_pointer)[0],(*a_pointer)[1]);
    *status = error_status_ok;
};

void array_test2(
    handle_t handle,
    unsigned32 size,
    idl_char an_array[],
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 2");
    for ( i = 0; i < size; i++)
    {
        printf("%c ",an_array[i]);
        printf("");
    }
    *status = error_status_ok;
}

void array_test3(
    handle_t handle,
    v_struct v_array,
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 3");
    for ( i = v_array.first; i < v_array.first + v_array.length; i++)
        printf("subscript %i value %c", i, v_array.array[i]);
```

```
    *status = error_status_ok;
}

void array_test4(
    handle_t handle,
    cv_struct *cv_array,
    error_status_t *status
)
{
    unsigned32 i;

    printf("Array test 4");
    for (i = (*cv_array).first; i < (*cv_array).first +
(*cv_array).length; i++)
        printf("subscript %i value %c", i, (*cv_array).array[i]);
    *status = error_status_ok;
}
```

The client sample code is as follows:

```
char5array fixed_array = {'a','b','c','d','e'};

v_struct varying_array = {3,4,{'a','b','c','d','e','f','g','h','i','j'}};

struct {
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
    char array[10];
}cv_array = {10, 4, 5, {'a','b','c','d','e','f','g','h','i','j'}};

array_test1(binding_h, &fixed_array, &status);

array_test2(binding_h, 5, fixed_array, &status);

array_test3(binding_h, varying_array, &status);

array_test4(binding_h, &cv_array, &status);
```

The server output will look like this:

```
Array test 1
a b
Array test 2
a
b
c
d
e
Array test 3
subscript 3 value d
subscript 4 value e
subscript 5 value f
subscript 6 value g
Array test 4
subscript 4 value e
subscript 5 value f
subscript 6 value g
subscript 7 value h
subscript 8 value i
```

Note that for the last test, the declared structure contains a conformant and varying array. The C language does not provide any intrinsic support for conformant arrays, and the actual IDL-generated header declaration for the type **cv_struct** looks as follows:

```
typedef struct  {
    unsigned32 size;
    unsigned32 first;
    unsigned32 length;
    idl_char array[1];
} cv_struct;
```

The declared structure contains an array only one element in length. When creating an instance of this type, the application must allocate a data structure of the correct size, either statically, as in the sample client code (the data item *cv_array*), or dynamically. The recipient of such a data structure (in this case, the server manager code), can then determine the actual size of the marshalled data by examining relevant field attribute variables (in this case, the structure's *size* member). Note also that IDL requires a structure containing a conformant array to be passed by reference; that is, as a pointer referent.

Conformant and varying arrays provide a way to pass blocks of contiguous data of varying sizes and ranges. However, there is no intrinsic mechanism for passing sparse arrays efficiently. Applications may, however, supply their own mechanisms for compressing and passing large, sparse arrays using the **[transmit_as]** mechanism.

There are a number of complications that can arise when using arrays of pointers. For example, an **[out]** or **[in, out]** conformant array of pointers, accompanied by an **[out]** or **[in, out]** field attribute variable, could potentially be of any size when returned to a caller. For **[ref]** pointers, which may not be NULL, the client must therefore ensure that all possible returned pointers in such an array actually point to valid storage. You can easily avoid such complications by sticking to the more straightforward array usages discussed here. However, if you find your application needs to use arrays in some more esoteric way, you should refer to the *AES/DC - RPC Volume*, "Chapter 4. Binding" on page 91, which contains a complete set of array and pointer usage rules.

Structures and Unions

There are no important policy issues relating to structures and unions as RPC parameters. Pointers and arrays as members of structures and unions are sometimes treated differently from separately declared types. By embedding pointers and arrays in structures and unions, you can sometimes achieve behavior that cannot be obtained by passing them as separate parameters.

Structures and unions can be used wherever they would be used in a non-RPC application. IDL structures differ from C language structures in one important respect: they may contain conformant arrays, which are not supported by C. A structure that contains a conformant array is itself conformant; that is, the size of the structure may not be determined until runtime. Applications need to do some extra work to determine the size of, and allocate, conformant structures. When structures are used to create linked lists and trees, the stubs do considerable work to insure that server allocated data is reflected back to the client.

IDL union syntax is quite different from C syntax, since IDL unions must be *discriminated* so that stubs can determine which of the contained data types to marshal. As with conformant and varying arrays, which use a field attribute variable to determine array size and bounds at runtime, IDL unions use a *discriminator* variable to determine which data type is marshalled.

IDL unions may be *encapsulated* or *non-encapsulated*. In an encapsulated union, the IDL compiler packages the union type and the discriminator in a structure. In a nonencapsulated union, the IDL switch_is] attribute is used to identify a discriminator variable. In this case, as in the case of array field attribute variables, the application must declare the discriminator and the union together, either as members of a structure or as parameters of an operation.

When a union is passed as a parameter, the value of the discriminator must either match one of the constants declared in the **switch** construct, or the switch must contain a **default** case. Otherwise, a stub marshalling error will occur.

Following are several examples of IDL union syntax. They are accompanied by the resulting IDL generated C header file declarations, and show how applications must refer to the union constructs declared in the IDL. The first example shows a set of declarations for an encapsulated union. The union holds either of two structures, one containing UUIDs, the other unsigned integers.

```
typedef struct two_uuid_s_t {
    uuid_t          uuid1;
    uuid_t          uuid2;
} two_uuid_t;

typedef struct two_uint_s_t {
    unsigned32      uint1;
    unsigned32      uint2;
} two_uint_t;

typedef enum {
    uuids,
    uints
} union_contents;

typedef union switch (union_contents type){
    case uuids:
        two_uint_t  integers;
    case uints:
        two_uuid_t  ids;
} test_union_t;
```

The resulting IDL generated C header declarations look like as follows:

```
typedef struct two_uuid_s_t{
    uuid_t uuid1;
    uuid_t uuid2;
}two_uuid_t;

typedef struct two_uint_s_t{
    unsigned32 uint1;
    unsigned32 uint2;
}two_uint_t;

typedef enum{
    uuids,
    uints
}union_contents;
typedef struct{
    union_contents type;
    union {
        /* case(s): 0 */
        two_uint_t integers;
```

```
        /* case(s): 1 */
        two_uuid_t ids;
    } tagged_union;
}test_union_t;
```

The IDL compiler packages the encapsulated union as a structure with the discriminant as the first member. To pass the union as an **[in]** or **[in, out]** parameter, the calling application must set the *type* field of this structure to either of the enumeration values **integers** or **ids**. To return the union as an **[out]** or **[in, out]** parameter, the callee must similarly be sure that the value of the *type* field is correctly set. To discover which data type was marshalled, the recipient can check the value of the *type* field.

The following is an example of non-encapsulated union usage. The **.idl** declaration is as follows:

```
typedef
    [switch_type(long)] union {
       [case (1,3)] float a_float;
       [case (2)] short b_short;
       [default]; /* An empty arm */
    } n_e_union_t;
```

The C header declaration of the non-encapsulated union generated by the IDL compiler is as follows:

```
typedef
    union {
        float a_float;
        short b_short;
    } n_e_union_t;
```

In this case, the discriminant must be separately declared in order for the union to be marshalled. The IDL **[switch_is]** attribute identifies the discriminant for an instance of the declared union type. Two examples of such **.idl** declarations follow:

```
/*
 * A structure that includes the union declared above and a member
 * that is used as the discriminant.  This structure can be passed
 * as an RPC parameter.
 */

typedef
    struct {
       long a;
       [switch_is(a)] n_e_union_t b;
    } a_struct;

/*
 * An operation declaration that passes the declared union type
 * along with a discriminant.
 */

void op1 (
    [in] handle_t h,
    [in, switch_is (s)] n_e_union_t u,
    [in] long s
);
```

Pipes

Pipes allow *application-level optimization* of bulk data transfer, by allowing the communication and processing of data to overlap. The actual data communications occur at about the same speed as arrays. However, pipes can reduce latency (how soon the application sees each “chunk” of data) and memory utilization. The intent is that the pipe routines should actually process the data and then get rid of it (for example, summarize it; write it to a file; pass it to another thread) rather than merely write it into an array. If an application desires to pass all of a stream of data and process it synchronously, then an array will probably be more efficient, since it entails considerably less processing overhead, as well as being simpler to program. For more on pipes as a topic in RPC application development, see the *OSF DCE Application Development Guide—Core Components*.

The transmit_as Attribute

The **[transmit_as]** attribute provides applications a way to do their own marshalling of data types. This is primarily useful as a way to deal with data structures that the stubs cannot marshall efficiently, such as sparse arrays. Following is an example of code to compress and reconstruct a large array by removing and then replacing all the zero-valued elements:

The **.idl** declarations are as follows:

```
/*
 * Transmit_as example: Here we turn a large sparse array into
 * a small conformant array for transmission.  The server is able
 * to reconstitute the sparse array.
 */

const long int S_ARRAY_SIZE = 32;

typedef struct{
    unsigned32 value;
    unsigned32 subscript;
} a_element;

typedef struct{
    unsigned32 size;
    [size_is(size)] a_element array[];
}compact_array_t;

typedef [transmit_as(compact_array_t)] unsigned32 sparse_array_t[S_ARRAY_SIZE];

void ship_array(
    [in] handle_t handle,
    [in] sparse_array_t *array,
    [out] error_status_t *status
);
```

All the callback routines are placed in a single module that is linked with both client and server (in this case, for the **test** interface). As an alternative, the appropriate callbacks could be declared separately within the client and server modules:

```
/*
 * test_xmit.c:
 *
 * The routines required to implement a [transmit_as] type.
 */
```

```
#include "test.h"

/* The to_xmit routine must allocate all space for the transmitted
 * type.  In general, the stubs have no way to determine how to allocate
 * space for the transmitted type.  Here, for example, the to_xmit
 * routine determines the size of a conformant array.
 */

void sparse_array_t_to_xmit(sparse_array_t *s_array,
                            compact_array_t **c_array
)
{
    unsigned32 i,j;
    unsigned32 csize;

    /* Count up the number of nonzero elements in the sparse array */

    for (i = 0, csize = 0; i < S_ARRAY_SIZE; i++)
    {
        if ((*s_array)[i] != 0)
        {
            csize++;
        }
    }

    /* Allocate a structure to hold the compact array */

    *c_array = (compact_array_t *)calloc(csize*2 + 1, sizeof(unsigned32));
    ((compact_array_t)**c_array).size = csize;

    /* Fill in the compact array from the nonzero elements */

    for (i = 0, j = 0; i < S_ARRAY_SIZE; i++)
    {
        if ((*s_array)[i] != 0)
        {
            ((compact_array_t)**c_array).array[j].value = (*s_array)[i];

            ((compact_array_t)**c_array).array[j++].subscript = i;
        }
    }
}
/*
 * The from_xmit routine may not have to allocate any space for the
 * presented type.  The presented type is always of a definite size
 * (conformant, varying, etc. types are not permitted), so the stub
 * provides an instance of the top level of the presented type.  In
 * this case, for example, s_array points to an instance of a sparse
 * array.  If the presented type contains any pointers, the from_xmit
 * routine needs to allocate space for the referents and the free_inst
 * routine needs to free them.
 */

void sparse_array_t_from_xmit(compact_array_t *c_array,
                              sparse_array_t *s_array)
{
    unsigned32 i,j;
    for (i = 0; i < ((compact_array_t) * c_array).size; i++)
    {
        j = ((compact_array_t)*c_array).array[i].subscript;
        (*s_array)[j] = ((compact_array_t)*c_array).array[i].value;
    }
}

/* This routine is called to free anything allocated by the
 * to_xmit routine.
 */
```

```
void sparse_array_t_free_xmit(compact_array_t *c_array)
{
    free(c_array);
}

/* This routine is called to free anything allocated by the
 * from_xmit routine.  Since from_xmit doesn't allocate anything
 * this is a null routine.
 */

void sparse_array_t_free_inst(sparse_array_t *s_array)
{

}
```

The client code to excercise the sparse array transmitted type is as follows:

```
sparse_array_t test_array;

/* Create a sparse array with only three nonzero members */

memset(test_array,0,sizeof(unsigned32)*S_ARRAY_SIZE);
test_array[0] = 2;
test_array[20] = 4;
test_array[31] = 8;

/*
 * When compressed, this array requires 7 32-bit integers, as opposed
 * 32 32-bit integers for the uncompressed array.  If you don't care
 * about reconstructing the sparse array on the server side, you can
 * get even more efficiency.
 */

ship_array(binding_h, &test_array, &status);
```

The server manager code is as follows:

```
void ship_array(
    handle_t binding_h,
    sparse_array_t *array,
    error_status_t *status
)
{
    int i;

    /*
     * Print the elements of the sparse array.
     */

    for (i = 0; i < S_ARRAY_SIZE; i++)
    {
        printf("%i", (*array)[i]);
    }
    *status = error_status_ok;
}
```

Note that the **free_inst** routine will not be needed if the transmitted type does not contain pointers. However, the routine is called by the stub automatically in any case, so at least a null routine must be provided. As an exercise, you might add **printf( )**s to each callback to see when it is called. You could also add code to show the format of the transmitted array before it is reconstructed by the **from_xmit** routine. Finally, you can create an even more efficient compression by not attempting to reconstruct the original array on the server side.

Chapter 7. Errors and Messaging

Applications should adopt a consistent and portable error handling style. This includes methods for returning errors from remote procedure calls, generation of application-specific status codes, and the generation and display of error text. This chapter recommends a set of techniques for handling all of these issues. The chapter also makes recommendations about error logging for applications that choose to use the logging facilities.

Error Handling

By default, the RPC runtime generates exceptions for RPC remote and communications errors. However, the default exception handler dumps core, which is not a very useful client response to such errors as failure to connect with a server that is down. Although the default error handling model attempts to treat an RPC call as a single continuous thread, propagating server errors back to the client, it will probably be more useful for most applications to contain the effects of server errors on the server side of the application. In this model, the client will output an error message when, for example, the server dumps core.

Thus, it is recommended that applications establish some explicit error handling mechanism for RPC calls. The *AES/DC* recommends the use of status returns as being a more portable way of handling errors than using exceptions. This recommendation is also consistent with the error-handling model for the RPC API.

You can have remote calls' communications and remote runtime errors reported through a status parameter by specifying the **[comm_status]** and **[fault_status]** attributes for the calls in the application's **.acf** file. The IDL compiler does not require that a status parameter be explicitly declared in the interface declaration, since it will add such a parameter implicitly. The **comm_status** and **fault_status** parameters are not really part of the remote interface: they are supplied by the client stub as one way of handling remote exceptions.

However, server managers need to report application-specific errors as well. Although such errors can be reported through function return values or a separate application error status parameter, the most consistent method is to use a single status parameter to report all errors. In this way a client need not concern itself with two or three separate error parameters, and can use a consistent error handling scheme for both API and application RPC errors. In order to return application-specific errors, such a status parameter must be part of the IDL specification of the interface. The recommended method is therefore to declare a *status* parameter as part of the application's **.idl** file declarations, and then add the **[com_status]** and **[fault_status]** attributes to the parameter by declarations in the **.acf** file.

Of course, if application and DCE runtime errors are to share the same status parameter, they must use disjoint error number spaces. The DCE messaging facilities provide a means to do this.

Messaging Facilities

The need to acquire an error number space is related to the larger issue of messaging in general. Applications typically need messages both for error reporting and for other status reporting and informational purposes. The recommended practice is to use messages catalogs for all messaging.

Message catalogs permit applications to deal with messages as numerical constants and to keep all associated text separate from the application itself. This is especially important for internationalization requirements; applications deal only with generic error numbers; locale-specific message text is kept in separate catalogs.

Message numbers are partitioned by technology, component, and code. The technology and component fields select a message catalog, and the code indexes messages within a catalog. The requirement to have a unique application-specific error number space can be met by adopting a unique component field within a given technology. To facilitate this, OSF makes component numbers available in two ways.

1. Within the **dce** technology (the default core component technology field used by DCE implementations), OSF sets aside two components guaranteed not to be used by DCE implementations.
2. OSF sets aside an ISV technology and maintains a registry of component numbers which may be assigned to ISVs.

This makes two levels of uniqueness available to applications. An *unregistered* application can guarantee that its message number space does not conflict with that of DCE implementations or of any registered ISV components by using one of the reserved component numbers within the **dce** technology space. This does not, of course, guarantee that the application's message number space does not conflict with that of other unregistered applications. This is a sufficient guarantee only for applications that do not communicate or share application-specific message catalogs with other unregistered applications. For example, the client side of an unregistered application may encounter error number conflicts if it makes RPCs to the server side of an application that uses an overlapping error number space.

Note: A related restriction on such non-registered applications is that they must install their message catalogs in some application-specific place. Since message catalog names depend on component numbers, other applications may be using the same message catalog names.

Applications that need to guarantee a unique error number space among all DCE applications should use a registered component number obtained from OSF. This is the recommended procedure for applications that have public interfaces that are likely to be called by other applications.

DCE Errors and DCE Messages

An application that wishes to use the DCE message facilities must organize all of its message text into a separate file which is compiled by the **sams** utility to generate a message catalog. The result of the **sams** compilation is that a set of DCE-consistent, application-specific codes for all the messages (not only errors) is generated. Use of the DCE facilities thus guarantees that application-specific status codes will be disjoint from those used by DCE for **fault** and **comm** status values,

and for API calls. The application can then use exactly the same error handling and reporting strategy for application RPC calls as for API calls.

Of course, generating the message catalogs is only one aspect of using the DCE facilities. The DCE routines that access the message catalogs to output the messages must also be used.

DCE Application Message APIs

Message generation by distributed programs can be divided into two broad kinds:

- Normal (often user-prompted, client-generated) messages
- Server event messages, containing information about server activity, either normal or extraordinary

Similarly, DCE makes available to applications two messaging APIs:

- The DCE messaging interface
- The DCE serviceability interface

The DCE serviceability interface is designed specifically to output messages of the second (server event) type. Messages in the first category can be output using the DCE general purpose application messaging routines.

Although the two interfaces, broadly speaking, do the same general thing (that is, write messages), their functionality was designed to serve different needs, both of which occur in most distributed applications. Nevertheless, either interface can be used more or less exclusively of the other, if desired. Both interfaces use message catalogs (for the most part) to generate output; the catalogs themselves are generated by **sams** during compilation, as mentioned earlier.

The following sections describe some aspects of using the serviceability interface. Full discussions of both interfaces can be found in the *OSF DCE Application Development Guide—Core Components*.

Serviceability and Logging

The DCE serviceability facilities allow server applications to display or log messages, to control message routing, and to associate actions with messages. A remote serviceability interface also makes it possible to control server message routing and filtering via **dcecp** or from application management clients.

The serviceability mechanism is designed to be used mainly for server informational and error messaging—that is, for messages that are of interest to those who are concerned with server maintenance and administration (in the broadest sense of these terms). The essential idea of the mechanism is that all server events that are significant for maintaining or restoring normal operation should be reported in messages that are made to be self-documenting, so that (provided all significant events have been correctly identified and reported) users and administrators will by definition always be able to learn what action they should take whenever anything out of the ordinary occurs. User-prompted, interactive, client-generated messaging should be handled through the DCE messaging interface.

Serviceability is also used by the DCE components (for example, DTS, CDS, and so forth) themselves. Consistent use of the same message mechanism by DCE implementations and applications should result in simplified DCE administration.

DCE components use the serviceability facilities according to the following guidelines; it is recommended that DCE applications use them also.

- All servers should report when they are started, and when they have completed their initialization and are ready to perform work. They should also indicate when they are going off-line.
- All program exits should be reported as fatal errors. Similarly, all calls to **abort( )** should be replaced by calls to **dce_svc_printf( )** with the **svc_c_action_abort** action attribute specified.
- Errors which make it impossible for the application to proceed should be reported as close as possible to the point of occurrence. This includes such conditions as: failure to allocate memory, failure to open a configuration file for reading, or a log file for writing, and so on.
- Conditions which may indicate system-level malfunction or poor performance must be reported.
- Routine administrative actions should be reported as informational messages. This includes: creation, modification and deletion of tickets, threads, files, sockets, RPC endpoints, or other objects; message transfer, including name lookup, binding, and forwarding; and database maintenance, including replication or synchronization.

The severity level attribute for each message can be determined according to the following criteria:

- Fatal error exit (**svc_c_sev_fatal_error**). An unrecoverable error has occurred requiring special manual recovery actions to take place, such as database restoration. The program usually terminates immediately.
- Error detected (**svc_c_sev_error**). An unexpected event that is nonterminal or is correctable via human intervention has occurred, such as a timeout. The program continues although some functions or services may not be available. This may also be used to indicate that a particular request or action could not be completed.
- Correctable error (**svc_c_sev_warning**). An error occurred that was automatically corrected, such as a configuration file was not found so that defaults were used. This may also be used to indicate a condition that may be an error if the effects are undesireable, such as removing all files when a nonempty directory is removed. This may also be used to indicate a condition that if not corrected will eventually result in an error, such as when a printer is running out of paper.
- Informational notice (**svc_c_sev_notice**). A predetermined major event has occurred, such as a server started.
- Verbose information notice (**svc_c_sev_notice_verbose**). A predetermined event has occurred, such as a directory entry was removed.
- Debug level 1 (**svc_c_debug1**) through debug level 9 (**svc_c_debug9**). Messages in the nine debug levels would not normally appear in production code.

An appropriate action may be associated with an error message by ORing one of the **svc_c_action. . .** values with the message attribute. Note that the **svc_c_action_abort** action, which results in a call to **abort( )**, does not provide any reliable means to clean up and should only be used where the default **abort( )** action, which is typically to dump core, is appropriate. Cleanup for the **svc_c_action_exit** action can implemented by supplying an **atexit( )** handler.

In addition to these guidelines, a persistent server application that does message logging should consider exporting the remote serviceability interface as a means to simplify server administration.

Sample Code

The sample server application source compiles a message catalog as well as the required auxiliary **.c** and **.h** files from a **sams** file. In part, the file looks like the following (for the full file, see “Chapter 10. A Sample Application” on page 177):

```
# Part I
component       smp
table           smp__table
technology      dce
####################################################################
# Part II
serviceability  table smp_svc_table handle smp_svc_handle
start
    subcomponent smp_s_server   "server"       smp_i_svc_server
    subcomponent smp_s_manager  "manager"      smp_i_svc_manager
    subcomponent smp_s_binder   "binder"       smp_i_svc_binder
end

####################################################################
# Part III
# Note that defining the "sub-component" and "attributes" fields
#  will result in a convenience macro's being generated for the
#  message in question...

start
code              sign_on
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Starting up"
explanation       ""
action            "None required."
end

start
code              cleanup
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Cleaning up"
explanation       "Starting server cleanup"
action            "None required."
end

start
code              server_exit
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Exiting"
explanation       ""
action            "None required."
end

start
code              signal_catcher
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Spawning signal handler thread"
explanation       ""
action            "None required."
```

```
end

start
code             no_signal_catcher
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Spawn signal handler failed"
explanation      "RPC runtime error. pthread_create() failed."
action           ""
end

start
code             bad_entryname_count
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Bad entryname count"
explanation      "Count of entrynames doesn't match count of object uuids"
action           ""
end

start
code             cannot_resolve_name
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Can't resolve name"
explanation      "ACL manager resolver failed to resolve name"
action           "The ACL databases may be corrupt and need to be regenerated."
end

start
code             cannot_manage_keys
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Can't spawn key management thread."
explanation      "RPC runtime error."
action           ""
end

start
code             no_acl_dbs
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "ACL databases not found, creating them from scratch"
explanation      ""
action           "None required."
end

start
code             exporting_to
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Exporting to %s"
explanation      "Exporting to CDS entry"
action           "None required."
end
start
code             unexporting_from
subcomponent     smp_s_server
attributes       "svc_c_sev_notice"
text             "Unexporting from %s"
explanation      "Unexporting from CDS entry"
action           "None required."
end


start
code             importing_from
```

```
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Importing from %s"
explanation       "Importing from CDS entry"
action            "None required."
end

start
code              auth_set_client
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Beginning client authentication setup"
explanation       ""
action            "None required."
end

start
code              bindings_received
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Nr of %s bindings received == %d"
explanation       "Server diagnostic message."
action            "None required."
end

start
code              full_binding
subcomponent      smp_s_server
attributes        "svc_c_sev_notice"
text              "Full %s binding in string form == %s"
explanation       "Server diagnostic message."
action            "None required."
end
start
code              server_error
subcomponent      smp_s_server
attributes        "svc_c_sev_fatal"
text              "%s: %s"
explanation       "general error message"
action            "?"
end

start
code              no_permissions
subcomponent      smp_s_manager
attributes        "svc_c_sev_notice"
text              "No permissions"
explanation       "Client does not have permissions for operation"
action            "None required."
end

start
code              object_not_found
subcomponent      smp_s_manager
attributes        "svc_c_sev_error"
text              "Object not found"
explanation       "object was not found in UUID-indexed database"
action            "None required."
end

start
code              manager_error
subcomponent      smp_s_manager
attributes        "svc_c_sev_fatal"
text              "%s: %s"
explanation       "general error message"
action            "?"
```

```
end

start
code            binder_error
subcomponent    smp_s_binder
attributes      "svc_c_sev_fatal"
text            "%s: %s"
explanation     "general error message"
action          "?"
end
####################################################################
# Part IIIa
# Messages for serviceability table
#
#  Note that there has to be one of these for each of
#   the subcomponents declared in the second part of
#   the file (above)...


start           !intable undocumented
code            smp_i_svc_server
text            "Sample server"
end

start           !intable undocumented
code            smp_i_svc_binder
text            "Sample object binder"
end

start           !intable undocumented
code            smp_i_svc_manager
text            "Sample manager"
end
```

The server **main()** function then establishes the required serviceability context and defines a message table with the following calls:

```
/* Set the program name for serviceability messages...        */
dce_svc_set_progname(argv[0], &status);

/* Get serviceability handle...                               */
smp_svc_handle = dce_svc_register(smp_svc_table,
                        (idl_char*)"smp",
                        &status);

/* Set up in-memory serviceability message table...           */
dce_msg_define_msg_table(smp__table,
                        sizeof smp__table / sizeof smp__table[0],
                       &status);
```

The following fragments illustrate remote error handling using a common *status* parameter. The **.idl** file for the **sample** interface includes the following declarations:

```
void sample_call(
        [in] handle_t binding,
        [out] long *status,
        [in,out] error_status_t
*remote_status);
```

This is matched in the **.acf** file by the following:

```
interface sample
{
    sample_call([comm_status,fault_status] remote_status);
}
```

Then, for example, the server implementation of the **sample_call( )** remote call can return the **smp_s_no_perms** status code on authorization failure:

```
void
sample_call(
    rpc_binding_handle_t binding,       /* Client binding.          */
    idl_long_int *status,
    error_status_t *remote_status)
{

    extern uuid_t sample_acl_mgr_uuid, sample_acl_uuid;
    boolean32 authorized = 0;

    /* We have to explicitly initialize the remote status value;    */
    /*  otherwise, if no error occurs in the transmission (which    */
    /*  would cause the runtime to assign an error value to this    */
    /*  variable), its value will be whatever it happened to be     */
    /*  when the RPC was made by the client...                      */
    *remote_status = rpc_s_ok;

    DCE_SVC_DEBUG((smp_svc_handle,
               smp_s_manager,
               svc_c_debug6,
               "Entering sample_call()..."));

    /* Check whether client is authorized or not...                 */
    DCE_SVC_DEBUG((smp_svc_handle,
               smp_s_manager,
               svc_c_debug6,
               "Calling dce_acl_is_client_authorized()..."));
    dce_acl_is_client_authorized(
        binding,              /* Client's binding handle.       */
        &sample_acl_mgr_uuid,   /* ACL manager type UUID.         */
        &sample_acl_uuid,      /* The ACL UUID.                  */
        NULL,                 /*  Pointer to owner's UUID.      */
        NULL,                 /* Pointer to owner's group's UUID.*/
        sec_acl_perm_read,     /* The desired privileges.        */
        &authorized,           /* Will be TRUE or FALSE on return. */
        remote_status);

    if (*remote_status != error_status_ok)
    {
        print_manager_error("dce_acl_is_client_authorized()",
                      *remote_status);
        return;
    }

    if (authorized)
    {
        DCE_SVC_DEBUG((smp_svc_handle,
                   smp_s_manager,
                   svc_c_debug8,
                   "Call authorized"));

        /* HERE'S WHERE WE SHOULD ACTUALLY DO SOMETHING!        */

    *status = error_status_ok;
    }
    else
    {
```

```
            DCE_SVC_DEBUG((smp_svc_handle,
                           smp_s_manager,
                           svc_c_debug8,
                           "Call not authorized"));

            /* Return no permissions status to client              */
            *status = no_permissions;
    }

    DCE_SVC_DEBUG((smp_svc_handle,
                   smp_s_manager,
                   svc_c_debug6,
                   "Successfully exiting sample_call()"));

}
```

The client making the **sample_call( )** remote call can then check both RPC **comm** and **fault** status and application-specific status and display any error messages with the same code:

```
sample_call(binding_h, &rpc_status, &rpc_remote_status);
if (rpc_remote_status != error_status_ok)
{
        print_error("sample_call()", rpc_remote_status);
        exit(1);
}
```

Chapter 8. Object-Oriented Applications with Distributed Objects

The DCE Interface Definition Language (IDL) compiler includes support for C++ language syntax features that provide a distributed object framework. This chapter describes some terms and techniques for developing object-oriented DCE applications. Before you start developing your applications, read this chapter and then see the *OSF DCE Application Development Guide—Core Components* for more information.

In general, using C++ for your DCE applications should be easier than using C because the DCE mechanisms can be better hidden from the developer. The following are some examples:

- The C++ compiler forces the server implementation of all interface operations. Although all DCE applications must do this to work properly, there is nothing to prevent you from writing an incomplete server in C. An added benefit of C++ is that the IDL compiler automatically generates the manager class with all the function signatures defined.
- The DCE function table known as an entry point vector (EPV) is generated automatically in object code by a C++ compiler, rather than in C code by the IDL compiler. In addition, you never have to construct a manager EPV as you might in your C code.
- C++ automatically provides a mechanism for grouping objects into types with its class data structure. This means it is unnecessary to use the DCE **rpc_object_set_type()** routine and associated routines.
- For DCE servers, interfaces are automatically registered with the DCE runtime.

These and other features make for easier and faster development.

DCE supplies exception-handling macros to use in distributed applications. You should use the DCE macros in your applications, instead of the standard C++ macros, to be sure exceptions are propagated correctly from servers to clients.

Distributed applications traditionally use the client/server model, in which a client application binds to a server and makes a request, to which the server responds. In this model, the distinction between a client and server is strong from both a development and runtime perspective. The client/server model is still convenient when developing and describing distributed, object-oriented applications, but an application is not distinctly a client or server. Whether an application is a client or server is more a condition of its execution than a characteristic of its development. It is convenient to say that a client requests the use of a distributed object and that objects are maintained by servers. However, clients also create and maintain their own objects that may be used by other applications, and servers may need to use objects maintained by other applications.

Kinds of Objects

In addition to the local objects that servers and clients each have, there are two major kinds of distributed objects: dynamic objects and named objects. Distributed objects exist in the server's memory space, and clients refer to them via a pointer data structure known as an *object reference*.

Distributed dynamic objects are created on a server by a client request for the exclusive use of the invoking client. These objects reside on the server, but clients initiate the object's creation using an object creator function defined in the interface. In C++, dynamic objects are simply local objects created at runtime. In the context of DCE, we use the term *dynamic objects* to mean distributed dynamic objects. Dynamic objects are typically short-lived; that is, in the life span of the client rather than that of the server.

Well-known objects represent specific resources supported by one or more servers and are called *named objects* when they are identified by a name stored in a name service. Each server that supports the resource creates and maintains its own object representing the resource.

Clients control dynamic objects and servers control named objects. The server can give out a dynamic object reference only to the initiating client and cannot give out the reference to any other client. However, the client that initiated the creation of a dynamic object can become a server to another client and give out a reference to the dynamic objects it owns. Servers do not automatically delete dynamic objects when a client exits (unlike other RPC features such as a context handle). This allows clients to pass object references to other programs to use. This also means that it is the *responsibility of the clients* to delete all their dynamic object references, thereby causing remote procedure calls to delete the object on the server. On the other hand, a server may pass object references for named objects to more than one client, and the server has control over when a named object is deleted. Clients delete only their local reference to named distributed objects; they cannot delete the actual object on the server.

Persistent objects are known objects (usually named) whose characteristics are maintained independently from a specific server, usually on disk.

Reference Counting: How Objects Keep Track of Multiple Clients

Reference counting is simply a record that each dynamic object maintains internally of how many clients have a reference to it. Dynamic object reference counting on a server is handled for you by DCE. Reference counting is maintained so that a server knows when to finally release the resources of an object. An object's reference count increases when the client passes a dynamic object to another application, and an object's reference count decreases when each object reference is deleted. This is why we have stressed that it is the responsibility of clients to delete their object references.

In addition, if one client passes a dynamic object to another and the original client then exits or dies, the second client's object is perfectly valid with respect to the server.

With the volatility of a distributed environment, even well-designed clients could lose a connection or exit unexpectedly, leaving the reference count in an incorrect state. Robust servers can delete any dynamic objects they wish, but they run the risk of causing an unexpected failure on clients. Depending on the application, this may be acceptable. For example, in a trading system where dynamic objects may be valid only during a single day, nightly purging of stale objects makes sense.

Using Interface Definitions to Design Classes

Since RPC is the mechanism DCE clients and servers use to become distributed applications, this mechanism must be introduced into C++ applications in an object-oriented way. When IDL compiles an interface definition, it generates a class hierarchy for clients and one for servers that provide the transition from RPCs to object-oriented applications. This is shown in the following figure.

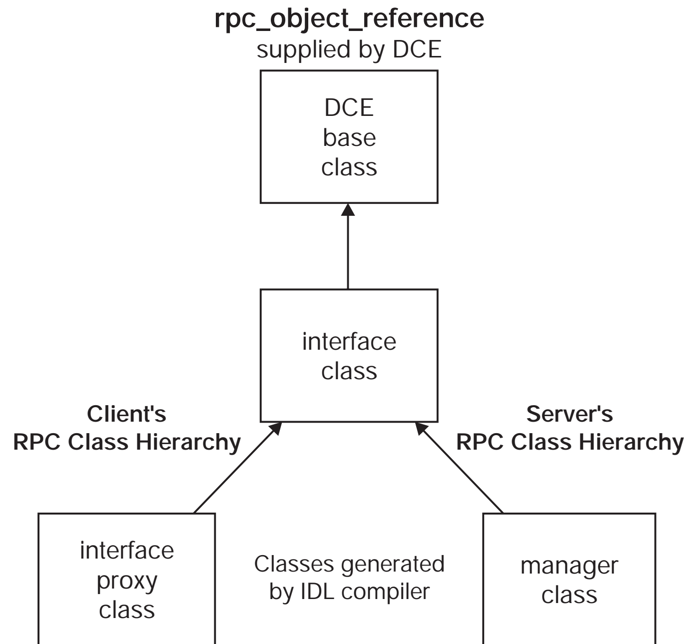


Figure 15. The RPC Class Hierarchy

At the top of each RPC hierarchy is a DCE-supplied class called **rpc_object_reference**. This *DCE base class* encapsulates the RPC mechanisms and provides a framework for identifying, distributing, and tracking objects. It is used by the generated stubs and not directly by your application code.

The IDL compiler also generates an *interface class* derived from the DCE base class. The interface class contains the public member functions specified in the interface definition. The interface class provides the link between the RPC mechanisms of the DCE base class and your application.

Although the interface class is the same for both the client and server hierarchy, the client and server must implement these functions quite differently. The server implements application-specific functions that operate on the actual object, while the client uses **idl**-generated functions that provide the distributed mechanisms. Polymorphism provides the capability to hide the implementation details of the interface class. In C++, this flexibility is easily provided by creating the interface class as an *abstract class* with all *pure virtual functions*. This means that none of the functions of an interface class are implemented directly, so no objects of the interface class can be created directly. This apparent restriction simply means that to implement the member functions, another class must be derived from the interface class.

At this point, the client and server RPC hierarchies diverge. In the client, the IDL compiler automatically derives a *proxy class* from the interface class. The proxy class implements in the client stub all the pure virtual member functions of the interface class. These functions handle all the underlying RPC code necessary for your client to access distributed objects. Thus, when a client calls the interface class's member functions, the polymorphic behavior of the C++ class hierarchy causes automatic invocation of the appropriate proxy class function in the stub. In the server, the IDL compiler automatically generates a *manager class* with empty function bodies.

You can choose to implement the server's interface operations in two ways:

- Implement the functions in the generated header file. However, be aware that under normal circumstances, a new IDL compilation of the interface will overwrite the header file with empty functions. (You can override this IDL compiler behavior with the **-no_cxxmgr** option.
- Derive a new implementation class from the automatically generated manager class.

Using Static Functions in Interface Design

Your IDL interfaces must be designed to make it clear which functions are static, because it may not be intuitive to first-time users. If something is static it doesn't move or change; it retains its state. In C++ the term *static* has a little more meaning than this. C++ uses the term to restrict the definition of a variable or function to within a specific scope of a program, and it causes values to be retained when program execution comes back to that scope. A variable or function that is static is like a refined global variable or function. Both static and global variables give a program a convenient way for one part to easily communicate with another without using parameters, but a static variable's scope is confined to a portion of the program, rather than the scope of the entire program. A static class member is a data member or member function that is in the scope of all objects of a particular class, but not in the scope of any other class objects or code of the program. A class's objects use static members to share data and functionality in a way that saves storage. A static data member acts as a class-global variable because there is only one instance of it, no matter how many instances of the class exist.

Programs and interfaces can declare a member function as static. Unlike nonstatic class members, static class members can be accessed or invoked directly, even if an instance of the class doesn't exist. For this reason, it is useful to developers if the interface designer identifies all operations in the interface that are to be static member functions by doing all of the following:

- Identifying static operations with comments in the IDL file.
- Grouping static operations together in the IDL file.
- Specifying static operations in the interface by using the **static** keyword. This will self-document the static member functions. However, this imposes a restriction that requires that the interface be compiled for C++ only and that servers implement the member function as static. If you do not want to impose this in the interface definition, use the **cxx_static** attribute in an Attribute Configuration File (ACF) to specify a static member function.

Adding an Interface Rather than Changing One

An interface can inherit the features of another interface; thus not only do you have class hierarchies, but you also can have interface hierarchies as well. We suggest

that you add more interfaces rather than change an existing one, even though IDL provides for different interface versions. Reasons for this include the following:

- Adding an interface will self document new features. There is no way for a developer to know which operations are old and which are new unless they compare the old and new interfaces or the interface designer has documented where the new operations begin.
- Adding an interface will not break the interface hierarchy. Adding additional operations to a base class can be a problem for applications.
- Adding an interface allows you to bypass the tedious procedure for changing an interface and its version numbers.

Binding to Distributed Objects Rather than Servers

When binding to objects, binding information is hidden from the application by encapsulating it in the **rpc_object_reference** base class that every DCE object inherits. An application refers to a distributed object via an *object reference*. An application initially obtains an object reference by calling an appropriate operation defined in an interface definition or a special **bind( )** function generated when the interface definition is compiled. Once an object reference is obtained, the application refers to the distributed object by calling its member functions, which are other operations defined in the interface definition. Since the location of the object (binding information) is already known and completely hidden, it is not appropriate to use explicit binding handles on these member functions.

Traditional DCE RPC object models limit what an object represents. In these models, objects represent resources on various servers or multiple servers on individual hosts. These models do not allow for objects to represent many different things in a single application. In addition, these models use a mechanism to group objects into types so that different sets of manager routines can be implemented, but this mechanism is unnecessary because it is built in with the C++ features of IDL. Using the object-oriented model with C++ instead is an elegant way to accomplish these things.

Clients Manipulate Objects Maintained on Servers

The following figure simplifies the relationship between an object-oriented application and a remote procedure call.

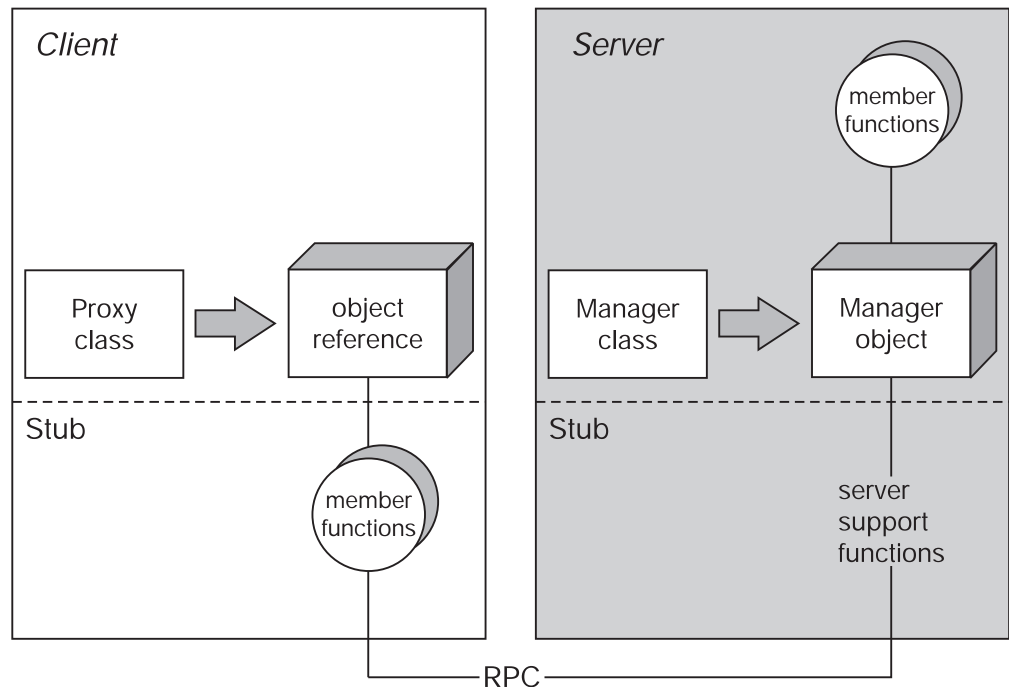


Figure 16. Distributed Objects and a Remote Procedure Call

A client first obtains an object reference to the remote object (manager object). An object reference is in the client's address space to act as a proxy for the remote object in the server's address space. In fact, an object reference is an instance of the client's proxy class.

When a member function is called, the function in the client stub executes to handle the RPC. The stub of the server managing the object receives the RPC and dispatches the call to the appropriate manager object code and its member function implementation. The actual object is an instance of the manager class.

Naming Objects

This section discusses the model used to advertise named objects. The model stores binding information for the object's hosts in the name service (for example, the Cell Directory Service, CDS). The model also stores the object's server location information in the endpoint map of each host that has one of the running servers.

There are many ways to use the name service to store information. Typical servers create and export information to a name service entry representing the server itself. In the named object model, an additional name service entry is also created for each named object. Thus, as the following figure shows, if five persistent objects are supported by two servers, there are a total of seven entries in the namespace. These allow clients a choice to bind by way of server entry names or objects names.

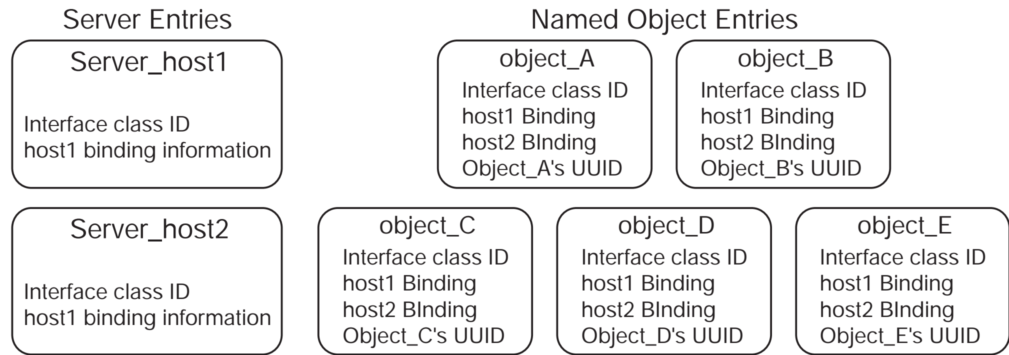


Figure 17. Server and Object Names in the Name Service

The object name is application specific, and each server exports its binding information to both its server entry and the object entries it supports. The name service entries for each object associate the object name with the combination of interface class ID, binding information (protocol sequence and host name) for each supported server, and object identifier (a UUID).

Object entries are different than server entries. It is important to understand that object entries represent objects, not servers. A server entry should contain binding information for only one server because it represents that server. On the other hand, object entries contain binding information for one (or more) servers because multiple servers may support the object.

As the number of named objects for an interface grows, management of the name service may become necessary. This could be done in the code of each server, but a more consistent and practical approach is to create scripts that use **dcecp**, or management applications that use the name service application programming interface routines. For example, the performance of the name service may degrade as the number of entries in a directory becomes very large. Place object names in a directory structure appropriate for the application.

If the application allows all clients to know the object names, the practical number of objects may be limited. However, an application can be designed to name a service object that clients look up, and pass in the specific object name (or identifying number). In this case, the servers can implement (and even later change) the object name location any way preferred: by using CDS, a commercial database, or an application-specific database.

Chapter 9. Server Management

Every DCE server requires some management. At a minimum, servers need to be started and stopped. In addition, servers usually provide generic server information such as the server principal name and an indication that the server is listening for remote calls. Servers may also permit other kinds of management operations while they are running; it is perfectly feasible to have a server reinitialize or even unregister and reregister endpoints while it is running.

From the management perspective, servers are thought of as either *on-demand* or *persistent*. In the on-demand model, a server only starts (thus occupying system resources) when it is needed. When an on-demand server is installed, a startup configuration is also installed with **dced**. Such a server would then use the configuration (obtained by a call to the **dce_server_inq_server()** routine) when it is auto-started by **dced** on receipt of an RPC request for an interface, operation, or object registered for that server.

A persistent server is one that runs continuously. Starting, stopping and otherwise managing such a server are typically considered priviledged operations. In general, a robust persistent server should provide a separate application control program that calls the DCE management interfaces (APIs for **dced**, RPC, and the like) and the application's own management interface (if one is provided). Of course, a server cannot start itself, but an application control client program can start the server via the **dced**. The model looks something like that shown in the following figure:

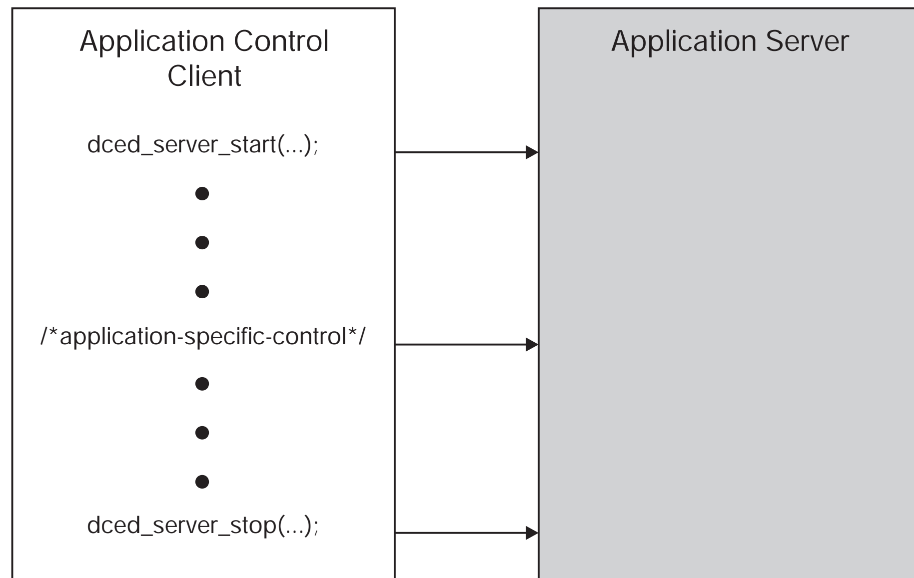


Figure 18. Managing a Server with a Control Client

In addition to starting and stopping the server, **dced**'s management routines provide other control operations. For example, the control program can use **dced_server_disable_if( )** and **dced_server_enable_if( )** to disable and reenable specific interfaces offered by the server. Application-specific management operations can be used to exert even finer control than is possible with the DCE-provided services.

Application Support for Server Management

Applications can support server management at three levels. At a minimum, every server automatically supports the RPC management API (routines the begin with **rpc_mgmt_**). By attaching an authorization function to the management interface (via a call to **rpc_mgmt_set_authorizaton_fn( )**), a server can set nondefault access to the generic management functions. Although these routines give a management program some control of the server, some of these routines only work locally, so the controlling client must run on the same host as the server.

At the second level, all servers should permit themselves to be managed from remote hosts via the **dced**. The requirements in the server's initialization code are minimal:

- The server should establish a security state using the **dce_server_sec_begin( )** call. This call establishes the server's identity with the RPC runtime such that clients can make authenticated remote procedure calls to it. The call also establishes with the security service the server's identity so that it can make authenticated remote procedure calls to other servers.

  Server writers should also give the **dced** (which runs with the host's principal identity) permission to control the server. Since the default is to disable remote control, the server must provide a nondefault authorization function that gives the machine principal access. An example of such an authorization function is given in "Chapter 3. Security" on page 51.
- The server must register as a DCE server using the **dce_server_register( )** call. This call fulfills the majority of the server initialization tasks including creating bindings, registering interfaces with the RPC runtime, registering endpoints with dced's endpoint mapper service, and advertising in the name service.

All servers should take these steps to operate correctly in DCE.

Finally, applications can provide application-specific server management. This would typically be done for a persistant server that provides access to some shared resource such as a database. Such a server can provide a set of privileged management operations—such as database maintainance—as a seperate application-specific management interface. Such an interface can be accessed by an application management client that can also call the DCE management interfaces. This type of management client is shown in the previous figure.

Manager Initialization

Server initialization tasks can typically be divided between essentially generic initialization—creating bindings, establishing security state, exporting to a name service, and listening for calls, among other things—and manager-specific initialization. (Remember that *management* refers to a set of tasks to control a server while a *manager* is a server's implementation of a set of operations from one or more interfaces.)

Once the server has called **rpc_server_listen( )**, the manager operations may be called asynchronously. The application may, however, need to perform some initialization before any manager operations are performed. For example, the sample storage manager (code example **context_manager.c**) needs to initialize its tables before any storage can be allocated out of them. An application has three choices about manager initialization policy:

1. The server can perform manager initialization before calling **rpc_server_listen( )**.
2. The server can have the first instance of a manager operation thread perform manager initialization, using the **pthread_once( )** facility. Although initilizing everything prior to listening for remote procedure calls is more straight-forward programming, some applications might benefit from this threaded approach. For example, those operations that do not need the initialization could forgo use of the **pthread_once( )** facility. This is the approach demonstrated in the sample storage manager.
3. The server can export manager initialization operations as part of its application-specific management interface, and have a management client perform the initialization.

Options 1 and 2 have similar effects and are appropriate for most servers. Option 3 might be appropriate for a persistent server where reinitialization of the running server is a useful operation. Such an operation is a perfect candidate for inclusion in an application-specific management interface for a persistent server.

Chapter 10. A Sample Application

This chapter presents the complete code for a generic sample application that illustrates the recommended policies. The code is as generic as possible in the sense that it demonstrates things that most servers need to do. This generic server code is contained in the **sample_server.c** and **sample_server.h** modules. The application-specific portion consists of a set of simple examples to illustrate various styles of RPC data usage, including pointers, pipes, and context handles. These illustrations are contained in **sample_manager.c** (the server side) and **sample_client.c** (the client side). **sample.idl** contains a set of sample interface definitions for the illustrated usages.

The Generic Server

The generic server implemented by **sample_server.c** demonstrates a variety of tasks that most servers need to carry out, such as exporting bindings, creating an authentication identity, establishing an ACL manager, and handling asynchronous signals. As much as possible, the bulk of each task is implemented as one or more separate functions. This modularity makes it easier to understand the requirements for coding each task since each function or related set of functions can be studied separately. Also, because the tasks performed are fairly generic, the functions should be reusable in something close to the form presented here by many servers.

The IDL file **sample.idl** is included here mainly to demonstrate the data type declarations used for the ACL manager. A more complete IDL file is given in “Object Bind Interface” on page 228 to show how the illustrated RPC data types are declared.

```
/****************************************************************************/
/* [27.VI.94]                                                              */
/*                                                                         */
/* sample.idl --                                                           */
/*                                                                         */
/*                                                                         */
/*                                                                         */
/*                              -77 cols-                                  */
/****************************************************************************/



[
uuid(002d70b2-671b-1d24-a1da-0000c0d4de56),
version(1.0)
]
interface sample
{

const long int TEXT_SIZE = 100;


void sample_call(
[in] handle_t binding,
[out] long *status,
[in,out] error_status_t *remote_status);

void sample_get_text(
[in] handle_t binding,
[in] uuid_t object_uuid,
[in,out,string] char text[TEXT_SIZE],
```

```
[out] long *status,
[in,out] error_status_t *remote_status);

void sample_put_text(
[in] handle_t binding,
[in] uuid_t object_uuid,
[in,out,string] char text[TEXT_SIZE],
[out] long *status,
[in,out] error_status_t *remote_status);


}


/****************************************************************************/
/* [27.VI.94]                                                              */
/*                                                                         */
/* sample.acf --                                                           */
/*                                                                         */
/*                                                                         */
/*                                                                         */
/*                            -77 cols-                                    */
/****************************************************************************/


interface sample
{
    sample_call([comm_status,fault_status] remote_status);

    sample_get_text([comm_status,fault_status] remote_status);

    sample_put_text([comm_status,fault_status] remote_status);


}
```

The IDL file **sample_db.idl** and the ACF file **sample_db.idl** are required to generate a server-only stub for the database serialization routines used by the ACL manager.

```
/****************************************************************************/
/* [27.VI.94]                                                              */
/*                                                                         */
/* sample_db.idl -- Here we declare a "serialization" function for the     */
/*                  sample object                                          */
/*                                                                         */
/*                                                                         */
/*  This file contains the declarations for the data type that will contain */
/*   the data that we will be storing "in" our sample objects.             */
/*                                                                         */
/*  The declarations are done in an IDL file because the data is sent      */
/*   across the wire by the ACL and Backing Store routines.                */
/*                                                                         */
/*                                                                         */
/*                                                                         */
/*                                                                         */
/*  The instructions for how to set up the IDL and ACF files to generate   */
/*   serialization procedures for backing store data types can be found in  */
/*   the dce_db_open.3dce manpage. This file and its accompanying .acf file */
/*   are written in conformance with the examples there.                 */
/*                                                                         */
/*                                                                         */
/*                            -77 cols-                                    */
/****************************************************************************/
```

```
[ uuid(00312933-403d-1d3d-a469-0000c0d4de56),
version(1.0) ]

interface sampledb
{
import "dce/database.idl";
import "sample.idl";


/*** FROM dce/database.idl:*************************************************/
/* This is the standard header for each "object" in the database.   */
/* IMPORTANT:
/* The header struct cannot have any variable-length data      */
/*      (e.g., char *). This is because when fetching (and un-      */
/*  marshalling) just the header, the variable part is at the   */
/*  end of the application's entire data object, not at the end */
/*  of the header.                                              */
/*
        THE FOLLOWING IS FILLED IN BY A CALL TO dce_db_std_header_init().
For an example of how these fields are accessed, see the routine
sample_resolve_by_name() in sample_server.c. Note that the fields
are automatically filled in by the ACL library; we only have to
read them.

typedef struct dce_db_dataheader_s_t {
    uuid_t uuid;              [...Object UUID.]
    uuid_t owner_id;
    uuid_t group_id;
    uuid_t acl_uuid;
    uuid_t def_object_acl;
    uuid_t def_container_acl;
    unsigned32 ref_count;
    THE FOLLOWING FIELDS ARE PRIVATE TO THE LIBRARY:
    utc_t created;
    utc_t modified;
    unsigned32 modified_count;
} dce_db_dataheader_t;

typedef enum {
    dce_db_header_std,
    dce_db_header_acl_uuid,
    dce_db_header_none
} dce_db_header_type_t;


WHICH ONE OF THE FOLLOWING YOU GET DEPENDS ON A FLAG PASSED TO
THE dce_db_open() ROUTINE...
typedef union switch (dce_db_header_type_t type) tagged_union {
    case dce_db_header_none: NONE ;
    case dce_db_header_std: dce_db_dataheader_t h;
    case dce_db_header_acl_uuid: uuid_t acl_uuid;
} dce_db_header_t;

*/
/*** ...END OF FROM dce/database.idl *****************************************/

/* Currently there is no actual object data in this structure.      */
/*  We just use it to hold the object piece of the                  */
/*  name->object->acl mapping...      [OLD NOTE]              */

typedef struct sample_record_s_t {
[string]   char message[TEXT_SIZE];
} sample_record_t;

typedef struct sample_data_s_t {
dce_db_header_t s_hdr;
```

```
sample_record_t s_data;
} sample_data_t;


void sample_data_convert(
[in] handle_t h,
[in,out] sample_data_t *data,
[in,out] error_status_t *st
);


void uu_convert(
[in] handle_t h,
[in,out] uuid_t *u,
[in,out] error_status_t *st
);

}


/****************************************************************************/
/* [27.VI.94]                                                               */
/*                                                                          */
/* sample_db.acf -- This makes the sample db interface into one that        */
/*                  does pickling.                                          */
/*                                                                          */
/*                                                                          */
/*                            -77 cols-                                     */
/****************************************************************************/


interface sampledb {
[encode,decode] sample_data_convert();
[encode,decode] uu_convert();
}
```

The generic server is then implemented by **sample_server.h** and **sample_server.c**.

```
/****************************************************************************/
/* [26.IX.94]                                                               */
/*                                                                          */
/* sample_server.h --                                                       */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*                            -77 cols-                                     */
/****************************************************************************/
/*                                                                          */
/* The following is passed via server_acl_mgr_setup() to the calls to      */
/*  server_create_acl(), where it is used to get a UUID to put in the       */
/*  ACLs we are creating, which will identify a user. In other words, at    */
/*  present the application is set up in such a way as to allow only the   */
/*  Cell Administrator principal to be a user with any kind of permissions  */
/*  at all on the objects we are creating. That's why the client can only   */
/*  be run successfully by a user dce_login'd as "cell_admin". The reason   */
/*  for doing things this way is that it allows us to have a user principal */
/*  we can always rely on being present, and thus avoid having to set up    */
/*  some user principals ourselves. Not that this would be so hard...       */
#define SAMPLE_OWNER "cell_admin"

/* Keytab file name:                                                        */
#define KEYTAB "FILE:/tmp/sample_keytab"

/* Default leaf name and length for the server entry:                       */
```

```
#define NAMELEN 20
#define DEFNAME "sample_server_entry"

#define IF_ANNOTATION "Sample interface, version 1.0"

/* At present we set up only two objects...                                */
/* A "well-known" residual name for the management "object":               */
#define MGMT_OBJ_NAME "server_mgmt"
/*                                                                         */
/* A residual name for a sample object:                                    */
#define SAMPLE_OBJECT_NAME "sample_object"

/* Relative pathname at which to locate newly-created backing store       */
/*  databases. Note that this is interpreted as the name of a subdirectory: */
#define ACL_DB_PATH "db_sample_acl"
/* Maximum length of a database pathname string...                         */
#define MAX_ACL_PATH_SIZE 50

/* Time allowed for cleanup of name space:                                 */
#define CLEANUPTIME 60

/* Maximum number of serviceability routings allowed:                      */
#define MAX_ROUTES 10

/* Maximum number of separate debug levels allowed:                        */
#define MAX_LEVELS 9



/* Data structure for holding server entry names to pass to                */
/*  server_export_objects():                                               */
typedef struct {
unsigned32 count;
unsigned_char_t *name[1];
} entryname_vector_t;




/* Handle for serviceability calls                                         */
dce_svc_handle_t smp_svc_handle;

/* Sample server-specific definitions:                                     */

/* Used by remote bind interface...                                        */
/* These are the backing store database handles:                           */
dce_db_handle_t db_acl, db_object, db_name;

#define mgmt_perm_inq_if sec_acl_perm_unused_00000080
#define mgmt_perm_inq_pname sec_acl_perm_unused_00000100
#define mgmt_perm_inq_stats sec_acl_perm_unused_00000200
#define mgmt_perm_ping sec_acl_perm_unused_00000400
#define mgmt_perm_kill sec_acl_perm_unused_00000800



/* The constants below come from aclbase.h (aclbase.idl)...                */
#define OBJ_OWNER_PERMS sec_acl_perm_read | sec_acl_perm_write \
| sec_acl_perm_delete | sec_acl_perm_test \
| sec_acl_perm_control | sec_acl_perm_execute

#define ALL_MGMT_PERMS mgmt_perm_inq_if | mgmt_perm_inq_pname \
| mgmt_perm_inq_stats | mgmt_perm_ping \
| mgmt_perm_kill | sec_acl_perm_test \
| sec_acl_perm_control

/* These are the two entry point vectors that are explicitly initialized:   */
```

```
extern rdaclif_v1_0_epv_t dce_acl_v1_0_epv;

extern sample_bind_v1_0_epv_t sample_bind_epv;

/* This global boolean records whether the backing store databases have    */
/*  actually been opened or not. This allows us to avoid calling the       */
/*  server_acl_mgr_close() routine when there are no open databases;call-  */
/*  the dce_db_close() routine on an unopened database will result in a    */
/*  core dump.                                                             */
boolean32 databases_open;


/****************************************************************************/
/* [27.VI.94]                                                               */
/*                                                                          */
/* sample_server.c -- Main program for "sample" server: initialization     */
/*                    and cleanup.                                          */
/*                                                                          */
/*  Note that no remote calls are defined in this file; for those, refer   */
/*   to either sample_manager.c or sample_bind.c.                           */
/*                                                                          */
/*                              -77 cols-                                   */
/****************************************************************************/

#define DCE_DEBUG

#include <stdio.h>
#include <malloc.h>
#include <time.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/param.h>


#include <dce/dce.h>
/* #include <dce/stubbase.h>  */
#include <dce/dce_cf.h>
#include <dce/dce_error.h>
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/keymgmt.h>
#include <dce/uuid.h>
#include <dce/exc_handling.h>
#include <dce/dce_msg.h>
#include <dce/dbif.h>
#include <dce/aclif.h>
#include <dce/dceacl.h>
#include <dce/pgo.h>
#include <dce/dced.h>


#include <dce/dcesvcmsg.h>
#include <dce/svcremote.h>

#include "dcesmpsvc.h"
#include "dcesmpmsg.h"
#include "dcesmpmac.h"

/* sample-specific includes:                                                */
#include "sample.h"
#include "sample_db.h"
#include "sample_bind.h"
#include "sample_server.h"
```

```
/******************************************************************************/
/* ANSI-C style prototypes for functions private to this module...          */

int do_command_line(int,
char **,
unsigned_char_t **,
entryname_vector_t *);

void signal_handler(char *);

void server_register_get_bindings(rpc_if_handle_t,
rpc_binding_vector_t **,
unsigned32 *);

void server_export_objects(rpc_if_handle_t,
rpc_binding_vector_t *,
uuid_vector_t *,
entryname_vector_t *,
unsigned_char_t *,
unsigned32 *);

void server_cleanup_objects(rpc_if_handle_t,
rpc_binding_vector_t *,
uuid_vector_t *,
entryname_vector_t *,
unsigned32 *);

void managekey(char *);

void server_get_identity(unsigned_char_p_t,
sec_login_handle_t *,
unsigned_char_p_t,
unsigned32 *);

void server_renew_identity(unsigned_char_p_t,
sec_login_handle_t,
unsigned_char_p_t,
unsigned32,
unsigned32 *);

void server_create_dflt_acl(dce_db_handle_t,
unsigned_char_t *,
void(*)(),
boolean32,
sec_acl_t *,
uuid_t *,
unsigned32 *);

void server_get_local_principal_id(unsigned_char_t *,
uuid_t *,
unsigned32 *);

void server_create_acl(uuid_t,
sec_acl_permset_t,
unsigned_char_t *,
sec_acl_t *,
uuid_t *,
unsigned32 *);

void server_store_acl(dce_db_handle_t,
dce_db_handle_t,
dce_db_handle_t,
sec_acl_t *,
uuid_t *,
uuid_t *,
unsigned_char_t *,
void *,
```

```
boolean32,
unsigned32 *);

void server_acl_mgr_setup(unsigned_char_t *,
dce_acl_resolve_func_t,
uuid_t,
uuid_t,
unsigned_char_t *,
sec_acl_permset_t,
unsigned_char_t *,
boolean32,
dce_db_handle_t *,
dce_db_handle_t *,
dce_db_handle_t *,
uuid_t *,
uuid_t *,
unsigned32 *);

void server_acl_mgr_close(dce_db_handle_t *,
dce_db_handle_t *,
dce_db_handle_t *,
unsigned32 *);

void server_rdacl_export(rpc_binding_vector_t *,
uuid_vector_t *,
unsigned32 *);

void server_rdacl_cleanup(rpc_binding_vector_t *,
uuid_vector_t *,
unsigned32 *);

void server_bind_cleanup(rpc_binding_vector_t *,
uuid_vector_t *,
unsigned32 *);

boolean32 sample_mgmt_auth(rpc_binding_handle_t,
unsigned32,
unsigned32 *);

dce_acl_resolve_func_t sample_resolve_by_name(handle_t,
sec_acl_component_name_t,
sec_acl_type_t,
uuid_t *,
boolean32,
void *,
uuid_t *,
error_status_t *);

void sample_bind_export(rpc_binding_vector_t *,
uuid_vector_t *,
unsigned32 *);

void create_server_uuid(uuid_t *,
uuid_vector_t *);

void print_server_error(char *,
error_status_t);

/******************************************************************************/


/* Default routing information:                                              */
#define SAMPLE_SVC_FATAL_DEFAULT_ROUTE 0
#define SAMPLE_SVC_ERROR_DEFAULT_ROUTE 1
#define SAMPLE_SVC_WARNING_DEFAULT_ROUTE 2
#define SAMPLE_SVC_NOTICE_DEFAULT_ROUTE 3
#define SAMPLE_SVC_VERBOSE_NOTICE_DEFAULT_ROUTE 4
```

```
#define MAX_DEFAULT_ROUTES 5
unsigned_char_t *default_routes[MAX_DEFAULT_ROUTES] =
{"FATAL:STDERR:",
 "ERROR:STDERR:",
 "WARNING:STDERR:",
 "NOTICE:STDERR:",
 "NOTICE_VERBOSE:STDERR:"};
/*
{"FATAL:TEXTFILE:/tmp/smp_svc_%ld",
 "ERROR:TEXTFILE:/tmp/smp_svc_%ld",
 "WARNING:TEXTFILE:/tmp/smp_svc_%ld",
 "NOTICE:TEXTFILE:/tmp/smp_svc_%ld",
 "NOTICE_VERBOSE:/tmp/smp_svc_%ld"};
*/

/* Default debug level and route...                                     */
unsigned_char_t *default_debug_route = (unsigned_char_t *)"smp:*.9:STDERR:-";
unsigned_char_t *default_debug_level = (unsigned_char_t *)"*.9";

/*                                                                      */
/* The debug level scheme is at present roughly as follows:             */
/*                                                                      */
/*     svc_c_debug1 -- not used.                                        */
/*     svc_c_debug2 -- not used.                                        */
/*     svc_c_debug3 -- not used.                                        */
/*     svc_c_debug4 -- messages announcing calls to system (DCE or OS)  */
/*                     routines.                                        */
/*     svc_c_debug5 -- messages announcing calls to local routines.     */
/*     svc_c_debug6 -- messages announcing entry/exit to/from server remote */
/*                     routines (e.g., remote bind, remote svc, sample in- */
/*                     terface operations, etc.). This is also the level */
/*                     for enabling announcements of library calls from  */
/*                     within these routines.                            */
/*     svc_c_debug7 -- messages announcing entry/exit to/from server local- */
/*                     ly called routines.                               */
/*     svc_c_debug8 -- messages displaying various debugging information. */
/*     svc_c_debug9 -- not used.                                        */
/*                                                                      */

int routes_G[MAX_ROUTES];

int debug_routes_G[MAX_LEVELS];

int debug_levels_G[MAX_LEVELS];


/* These two UUIDs could be treated as "well known": i.e. applications  */
/*   that use the same ACL manager for mgmt operations can use these... */

uuid_t mgmt_acl_mgr_uuid = {/* 0060f928-bbf3-1d35-8d7d-0000c0d4de56     */
0x0060f928, 0xbbf3, 0x1d35, 0x8d, 0x7d, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};

uuid_t mgmt_object_uuid = {/* 00573b0e-bcc2-1d35-a73e-0000c0d4de56      */
0x00573b0e, 0xbcc2, 0x1d35, 0xa7, 0xe3, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};


/* These UUIDs are specific to this server...                           */
/* Some ACL UUIDs that will be globally used:                           */
uuid_t mgmt_acl_uuid;
uuid_t sample_acl_uuid;

/* The UUID of the sample ACL manager:                                  */
uuid_t sample_acl_mgr_uuid = { /* 001a15a9-3382-1d23-a16a-0000c0d4de56  */
0x001a15a9, 0x3382, 0x1d23, 0xa1, 0x6a, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};
```

```
/* A UUID for a sample object:                                          */
uuid_t sample_object_uuid = {/* 00415371-f29a-1d3d-b8c8-0000c0d4de56      */
0x00415371, 0xf29a, 0x1d3d, 0xb8, 0xc8, 0x00, 0x00, 0xc0, 0xd4, 0xde, 0x56
};

/* The mgmt printstrings could be treated as standard for                */
/*   a standard mgmt ACL manager...                                      */
sec_acl_printstring_t mgmt_info = {"mgmt", "Management Interface"};


/* Note that we don't need to use the unused bits here;                  */
/*  it's just less confusing this way...                                 */

sec_acl_printstring_t mgmt_printstr[] = {
{ "i", "m_inq_if", mgmt_perm_inq_if      },
{ "n", "m_inq_pname", mgmt_perm_inq_pname    },
{ "s", "m_inq_stats", mgmt_perm_inq_stats    },
{ "p",  "m_ping",       mgmt_perm_ping        },
{ "k",  "m_kill",       mgmt_perm_kill        },
{ "c", "control", sec_acl_perm_control   },
{ "t", "test", sec_acl_perm_test      }
};

sec_acl_printstring_t sample_info = {"sample", "Sample RPC Program"};

sec_acl_printstring_t sample_printstr[] = {
{ "r", "read", sec_acl_perm_read },
{ "w", "write", sec_acl_perm_write },
{ "d", "delete", sec_acl_perm_delete },
{ "c", "control", sec_acl_perm_control },
{ "t", "test", sec_acl_perm_test },
{ "x", "execute", sec_acl_perm_execute    }
};


/***************
 *
 *
 * main --
 *
 *
 *
 *
 **************/
int
main(
int argc,
char *argv[]
)
{

unsigned32 status;   /* For status returned from library calls.     */
rpc_binding_vector_t *binding_vector; /* For bindings from RPC run- */
/*  time.                           */
unsigned_char_t *string_binding; /* For string binding conversions. */
unsigned_char_t *server_principal_name; /* Our server principal     */
/*  name, read from command line.   */
unsigned_char_t *uuid_string;   /* For UUID string conversions.     */
entryname_vector_t entryname_vector; /* List of server entry names, */
/*  read from command line.         */
sec_login_handle_t login_context; /* Our login context, for server- */
/*  assumed identity.               */
pthread_t sigcatcher;           /* Handle to signal catcher thread. */
uuid_vector_t server_uuid_v;    /* Array of server instance UUIDs.  */
/* At present there is only one of these,   */
/*  and it is generated dynamically at the  */
```

```
/*  beginning of the program. This ends up  */
/*  being used as an object UUID for the    */
/*  server's exported bindings and regis-   */
/*  tered mappings.                          */
int i;                  /* Utility index variable.                */
int route_error;        /* Condition variable for setting up de-  */
/*  fault serviceability routings. Not used */
/*  yet, however.                            */
uuid_t server_uuid;  /* The UUID that identifies this server in-  */
/*  stance; i.e., the object UUID that      */
/*  identifies our server's bindings. It is */
/*  created by us, in create_server_uuid(), */
/*  and goes into server_entry_ptr->id in   */
/*  make_server_entry(). It actually ends   */
/*  up being identical to the configuration */
/*  UUID, if one is created (by calling the */
/*  dced_server_create() routine).          */
dce_error_string_t error_string;  /* Used to directly retrieve er-  */
/*  ror message strings in cases where the  */
/*  serviceability call can't be used, be-  */
/*  cause the svc table hasn't been regis-  */
/*  tered yet.                               */
int print_status;       /* Used to return status from the         */
/*  dce_error_inq_text() routine.           */


databases_open = FALSE;

/* Process the command line...                                       */
do_command_line(argc,
argv,
&server_principal_name,
&entryname_vector);


/* The following calls set up default routing of serviceability     */
/*  messages. Note that these must be called before                 */
/*  dce_svc_register()...                                            */
for (i = 0, route_error = FALSE; (i < MAX_DEFAULT_ROUTES) &&(!route_error); i++)
{
fprintf(stdout, "          Setting default route %s ...\n",
default_routes[i]);
dce_svc_routing(default_routes[i], &status);
if (status != svc_s_ok)
{
print_server_error("dce_svc_routing(default_routes[i])",
status);
status = svc_s_ok;
}
}

/* Set the default serviceability debug level and route...          */
dce_svc_debug_routing(default_debug_route, &status);

if (status != error_status_ok)
{
dce_error_inq_text(status, error_string, &print_status);
fprintf(stdout, "dce_svc_routing(): %s\n", error_string);
/* exit(1);    */
}

/* Get serviceability handle...                                      */
smp_svc_handle = dce_svc_register(smp_svc_table,
(idl_char*)"smp",
&status);
if (status != error_status_ok)
{
```

```
print_server_error("dce_svc_register()", status);
exit(1);
}

/* Set up in-memory serviceability message table...                */
dce_msg_define_msg_table(smp__table,
sizeof smp__table / sizeof smp__table[0],
&status);
if (status != error_status_ok)
{
print_server_error("dce_msg_define_msg_table()", status);
exit(1);
}

dce_svc_printf(SIGN_ON_MSG);


/* Create an object UUID for this server instance...                */
create_server_uuid(&server_uuid, &server_uuid_v);

/* Register the interface and get bindings...                       */

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_register_get_bindings()"));
server_register_get_bindings(sample_v1_0_s_ifspec,
&binding_vector,
&status);
if (status != error_status_ok)
{
print_server_error("server_register_get_bindings()",
status);
exit(1);
}

/* Register server authentication information...                    */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_register_auth_info()"));
rpc_server_register_auth_info(server_principal_name,
rpc_c_authn_dce_secret,
NULL,
KEYTAB,
&status);
if (status != error_status_ok)
{
print_server_error("rpc_server_register_auth_info()",
status);
exit(1);
}

/* Assume new identity...                                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_get_identity()"));
server_get_identity(server_principal_name,
&login_context,
(unsigned_char_p_t)KEYTAB,
&status);
if (status != error_status_ok)
{
print_server_error("server_get_identity()", status);
exit(1);
}

/* Spin off a thread to wait for signals...                         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Spawning signal handler thread"));
if (pthread_create(&sigcatcher,
pthread_attr_default,
```

```
(pthread_startroutine_t)signal_handler,
(void*)0))
{
dce_svc_printf(NO_SIGNAL_CATCHER_MSG);
exit(1);
}

/* Export objects to namespace and register them in end-            */
/*  point map...                                                     */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_export_objects()"));
server_export_objects(sample_v1_0_s_ifspec,
binding_vector,
&server_uuid_v,
&entryname_vector,
(unsigned_char_t *)IF_ANNOTATION,
&status);
if (status != error_status_ok)
{
print_server_error("server_export_objects()", status);
goto CLEANUP_EXIT;
}

/* Register the remote binding interface...                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling sample_bind_export()"));
sample_bind_export(binding_vector, &server_uuid_v, &status);
if (status != error_status_ok)
{
print_server_error("sample_bind_export()", status);
exit(1);
}

/* Create a default ACL manager...                                   */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_acl_mgr_setup()"));
server_acl_mgr_setup(
(unsigned_char_t *)ACL_DB_PATH, /* Pathname for     */
/*  database files.                */
(dce_acl_resolve_func_t)sample_resolve_by_name,
/* Our name->ACL UUID resolution    */
/*  function.                       */
sample_acl_mgr_uuid, /* UUID of our ACL manager;    */
/*  hard-coded at top of this file. */
sample_object_uuid,  /* UUID of our sample object;  */
/*  hard-coded at top of this file. */
(unsigned_char_t *)SAMPLE_OBJECT_NAME, /* Name of   */
/*  our sample object.              */
OBJ_OWNER_PERMS, /* Owner's permissions on sample   */
/*  object.                         */
(unsigned_char_t *)SAMPLE_OWNER,   /* Principal     */
/*  name of sample object owner.    */
0,              /* TRUE => object is a container.   */
&db_acl,    /* Will contain ACL UUID store handle.  */
&db_object, /* Will contain obj UUID store handle.  */
&db_name,   /* Will contain name store handle.      */
&sample_acl_uuid, /* Will contain object ACL UUID.  */
&mgmt_acl_uuid,   /* Will contain mgmt ACL UUID.    */
&status);
if (status != error_status_ok)
{
print_server_error("server_acl_mgr_setup()", status);
goto CLEANUP_EXIT;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
```

```
uuid_to_string(&mgmt_acl_uuid, &uuid_string, &status);
if (status != uuid_s_ok)
{
print_server_error("uuid_to_string()", status);
return(0);
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of mgmt_acl_uuid == %s", uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, &status);
if (status != rpc_s_ok)
{
print_server_error("rpc_string_free()", status);
return(0);
}


/* Register the remote ACL interface (at the endpoint map,  */
/*  but not in the namespace)...                            */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_rdacl_export()"));
server_rdacl_export(binding_vector, &server_uuid_v, &status);
if (status != error_status_ok)
{
print_server_error("server_rdacl_export()", status);
goto CLEANUP_EXIT;
}


/* Start listening for calls...                                     */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_listen()"));
rpc_server_listen(rpc_c_listen_max_calls_default, &status);
if (status != error_status_ok)
{
print_server_error("rpc_server_listen()", status);
/* exit(1);     */
}

CLEANUP_EXIT:

/******************************************************************/
/*                                                                */
/* Cleanup code -- Reached either because the server listen       */
/*                 returned, or because we never got to listen in */
/*                 the first place due to some runtime error.     */
/*                                                                */
/*                                                                */
/******************************************************************/

/* Close the ACL databases...                                     */
if (databases_open)
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_acl_mgr_close()"));
server_acl_mgr_close(&db_acl, &db_object, &db_name, &status);
if (status != error_status_ok)
{
print_server_error("server_acl_mgr_close()", status);
}
}

/* Be sure credentials are still valid before we try to           */
/*  cleanup...                                                    */
dce_svc_printf(CLEANUP_MSG);
```

```
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_renew_identity()"));
server_renew_identity(server_principal_name,
login_context,
(unsigned_char_p_t)KEYTAB,
CLEANUPTIME,
&status);
if (status != error_status_ok)
{
print_server_error("server_renew_identity()", status);
}

/* Unexport server objects from namespace and from endpoint         */
/*  map...                                                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_cleanup_objects()"));
server_cleanup_objects(sample_v1_0_s_ifspec,
binding_vector,
&server_uuid_v,
&entryname_vector,
&status);
if (status != error_status_ok)
{
print_server_error("server_cleanup_objects()", status);
exit(1);
}

/* Unregister the remote ACL interface from the endpoint            */
/*  map...                                                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_rdacl_cleanup()"));
server_rdacl_cleanup(binding_vector, &server_uuid_v, &status);
if (status != error_status_ok)
   {
print_server_error("server_rdacl_cleanup()", status);
}

/* Unregister the remote bind interface from the endpoint           */
/*  map...                                                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_bind_cleanup()"));
server_bind_cleanup(binding_vector, &server_uuid_v, &status);
if (status != error_status_ok)
{
print_server_error("server_bind_cleanup()", status);
}

/* Print server exit message...                                     */
dce_svc_printf(SERVER_EXIT_MSG);

}

/******
 *
 * do_command_line -- Get and interpret arguments and options from the
 *                    command line, and do other setup related to the
 *                    command line's contents.
 *
 *                    Returns 0 if normal invocation, 1 if setup.
 *
 *    Called from main().
 *
 ******/

int do_command_line(
int argc,
char *argv[],
```

```
unsigned_char_t **server_principal_name,
entryname_vector_t *entryname_vector
)
{
dce_error_string_t error_string;
int print_status;
unsigned32 status;

/* Note that the code expects you to type as the second argument    */
/*  a slash-terminated full CDS directory name, to which it will    */
/*  then concatenate the entryname. It is this name that is then    */
/*  passed to the server_export_objects() routine later on.         */

/* Check the command line...                                        */
if ( (argc == 2) && (( strcmp(argv[1], "setup") == 0) || \
( strcmp(argv[1], "unsetup") == 0)) )
return 1;

else if (argc < 3)
{
fprintf(stdout, "\n Usage:\n");
fprintf(stdout, "       %s <principal_name><CDS_dir_name>/\n\n", argv[0]);
exit(1);
}

/* Get the server's principal name from the command line...         */
*server_principal_name = (unsigned_char_p_t)malloc(strlen(argv[1]));
strcpy((char *)*server_principal_name, (char *)argv[1]);

/* Get the list of server entry names from the command line...      */
entryname_vector->count = 1;
entryname_vector->name[0] = (unsigned_char_p_t)malloc(strlen(argv[2]) + NAMELEN);
strcpy((char *)entryname_vector->name[0], argv[2]);
strcat((char *)entryname_vector->name[0], DEFNAME);

/* Set the program name for serviceability messages...              */
dce_svc_set_progname(argv[0], &status);
if (status != error_status_ok)
{
dce_error_inq_text(status, error_string, &print_status);
fprintf(stdout, "dce_svc_set_progname(): %s\n", error_string);
exit(1);
}

return 0;
}




/******
 *
 * server_register_get_bindings --  Register an interface:
 *
 *         Set up only one type manager
 *         Use all protocol sequences
 *         Return the bindings
 *
 *    Called from main().
 *
 ******/

void server_register_get_bindings(
rpc_if_handle_t interface,             /* Interface to register.          */
rpc_binding_vector_t **binding_vector, /* To return bindings.             */
unsigned32 *status)                    /* To return status.               */
{
```

```
unsigned_char_t *string_binding;
int i;

*status = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_register_get_bindings()"));

/* Register the default interface, default epv, and nil type        */
/*  UUID...                                                         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_register_if()"));
rpc_server_register_if(interface, NULL, NULL, status);
if (*status != error_status_ok)
{
print_server_error("rpc_server_register_if()", *status);
return;
}

/* Use all available protocol sequences...                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_use_all_protseqs()"));
rpc_server_use_all_protseqs(rpc_c_protseq_max_reqs_default,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_server_use_all_protseqs()", *status);
return;
}

/* Get the binding handles generated by the runtime...              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_inq_bindings()"));
rpc_server_inq_bindings(binding_vector, status);
if (*status != error_status_ok)
{
print_server_error("rpc_server_inq_bindings()", *status);
return;
}

/*********************************************************************
*
* The following shows how to convert a vector of bindings into
*    string bindings, and to print them out...
*
*
*****/

dce_svc_printf(BINDINGS_RECEIVED_MSG, "sample",
(**binding_vector).count);

for (i = 0; i < (**binding_vector).count; i++)
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_binding_to_string_binding()"));
rpc_binding_to_string_binding((**binding_vector).binding_h[i],
&string_binding,
status);
if (*status != rpc_s_ok)
{
print_server_error("rpc_binding_to_string_binding()",
*status);
exit(1);
}

dce_svc_printf(FULL_BINDING_MSG, "sample", string_binding);
```

```
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&string_binding, status);
if (*status != rpc_s_ok)
{
print_server_error("rpc_string_free()", *status);
exit(1);
}

}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_register_get_bindings()"));

}


/******
 *
 *
 * server_export_objects -- Set up a simple object-based binding scheme
 *                          for the server, i.e.:
 *
 *          Register bindings and objects in the endpoint map,
 *          Then export each object to a separate name space entry.
 *
 *    The function uses a vector of entry names that correspond one-to-one
 *     with the objects in the object uuid vector. The server must have
 *     export permission to CDS in order to successfully execute this
 *     function.
 *
 *    Called from main().
 *
 ******/
void server_export_objects(
rpc_if_handle_t interface,            /* The interface specification.     */
rpc_binding_vector_t *binding_vector, /* The server's binding handles.    */
uuid_vector_t *object_uuid_vector,    /* Server instance UUID, created in  */
       /*  main.                            */
entryname_vector_t *entryname_vector, /* Server entry names, from command  */
       /*  line.                            */
unsigned_char_t *annotation,          /* Annotation string for endpoint   */
       /*  map entry.                       */
unsigned32 *status)                   /* To return status               */
{


uuid_vector_t object_uuid; /* Used to hold object UUIDs to be       */
/* passed to rpc_ns_binding_export().      */
int i;                     /* Index variable.                       */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_export_objects()"));

*status = error_status_ok;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_register()"));
rpc_ep_register(interface, binding_vector, object_uuid_vector,
annotation, status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_register()", *status);
return;
}

if (object_uuid_vector)
{
if (entryname_vector->count != object_uuid_vector->count)
```

```
{
dce_svc_printf(BAD_ENTRYNAME_COUNT_MSG);
return;
}

object_uuid.count = 1;

/* Export objects one at a time to CDS entries... */
for (i = 0; i < entryname_vector->count; i++)
{
dce_svc_printf(EXPORTING_TO_MSG, entryname_vector->name[i]);
object_uuid.uuid[0] = object_uuid_vector->uuid[i];
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ns_binding_export()"));
rpc_ns_binding_export(rpc_c_ns_syntax_default,
entryname_vector->name[i],
interface,
binding_vector,
(uuid_vector_t*)&object_uuid,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ns_binding_export()",
*status);
return;
}
}
}
else
{

dce_svc_printf(EXPORTING_TO_MSG, entryname_vector->name[0]);
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ns_binding_export()"));
rpc_ns_binding_export(rpc_c_ns_syntax_default,
entryname_vector->name[0],
interface,
binding_vector,
NULL,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ns_binding_export()",
*status);
return;
}
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_export_objects()"));

}



/******
 *
 * server_cleanup_objects -- Unexport and unregister all server objects.
 *
 *      The server must have valid credentials for this routine to
 *       successfully execute.
 *
 *    Called from main().
 *
 ******/

void server_cleanup_objects(
```

```
rpc_if_handle_t interface,            /* Interface to unregister.        */
rpc_binding_vector_t *binding_vector,  /* Server bindings to delete.      */
uuid_vector_t *object_uuid_vector,     /* Server instance UUID(s).        */
entryname_vector_t *entryname_vector,  /* Server entry names.             */
unsigned32 *status)                    /* To return status.               */
{

struct {
unsigned32 count;
uuid_t *uuid[1];
} object_uuid;

int i;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_cleanup_objects()"));

*status = error_status_ok;

/* Get rid of the endpoints...                                     */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_unregister()"));
rpc_ep_unregister(interface, binding_vector, object_uuid_vector,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_unregister()", *status);
return;
}

/* Get rid of the server instance UUID(s). However, note that at    */
/*  present there is only one of these, and it's hard-coded below.  */
if (object_uuid_vector)
{
if (entryname_vector->count != object_uuid_vector->count)
{
dce_svc_printf(BAD_ENTRYNAME_COUNT_MSG);
return;
}
object_uuid.count = 1;

/* Unexport objects one at a time from CDS entries... */
for (i = 0; i < entryname_vector->count; i++)
{
dce_svc_printf(UNEXPORTING_FROM_MSG,
entryname_vector->name[i]);

object_uuid.uuid[0] = object_uuid_vector->uuid[i];
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ns_binding_unexport()"));
rpc_ns_binding_unexport(rpc_c_ns_syntax_default,
entryname_vector->name[i],
interface,
(uuid_vector_t*)&object_uuid,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ns_binding_unexport()",
*status);
return;
}
}
}
else
/* I.e., there is only one server instance to unexport...          */
{
dce_svc_printf(UNEXPORTING_FROM_MSG, entryname_vector->name[0]);
```

```
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ns_binding_unexport()"));
rpc_ns_binding_unexport(rpc_c_ns_syntax_default,
entryname_vector->name[0],
interface,
NULL,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ns_binding_unexport()",
*status);
return;
}
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_cleanup_objects()"));

}

/******
 *
 * managekey -- Make sure the server principal's key is changed before
 *              it expires.
 *
 *      The key management thread which runs this function is created
 *       in server_get_identity(), below.
 *
 *
 ******/

void managekey(char *prin_name){          /* Server principal name
   */
unsigned32 status;

status = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering managekey()"));

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_key_mgmt_manage_key()"));
sec_key_mgmt_manage_key(
rpc_c_authn_dce_secret,   /* Authentication protocol.      */
KEYTAB,                   /* Local key file.               */
(idl_char *)prin_name,    /* Principal name.               */
&status);
if (status != error_status_ok)
print_server_error("sec_key_mgmt_manage_key()", status);

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting managekey()"));

}


/******
 *
 * server_get_identity -- Establish a new server identity with valid
 *                        credentials. This includes setting up a key
 *                        management thread.
 *
 *
 *    Called from main().
 *
 ******/
```

```
void server_get_identity(
unsigned_char_p_t prin_name,            /* Server principal name.         */
sec_login_handle_t *login_context,      /* Returns server's login context.  */
unsigned_char_p_t keytab,               /* Local key file.                */
unsigned32 *status)
{

pthread_t keymgr;
sec_passwd_rec_t *keydata;
sec_login_auth_src_t auth_src;
boolean32 reset_pwd;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_get_identity()"));

*status = error_status_ok;

/* Spin off thread to manage key for specified principal...         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling pthread_create()"));
if (pthread_create(&keymgr,            /* Thread handle.           */
pthread_attr_default,  /* Specifies default thread  */
/*  attributes.            */
(pthread_startroutine_t)managekey, /* Start rou-    */
/*  tine; see above.       */
(void*)prin_name))     /* Argument to pass to start */
       /*  routine: server princi-  */
       /*  pal name.                */
{
dce_svc_printf(CANNOT_MANAGE_KEYS_MSG);
return;
}

/* Create a context and get the login context...                    */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_setup_identity()"));
sec_login_setup_identity(prin_name,
sec_login_no_flags,
login_context,
status);
if (*status != error_status_ok)
{
print_server_error("sec_login_setup_identity()", *status);
return;
}

/* Get secret key from the keytab file...                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_key_mgmt_get_key()"));
sec_key_mgmt_get_key(rpc_c_authn_dce_secret,
keytab,
prin_name,
0,
(void**)&keydata,
status);
if (*status != error_status_ok)
{
print_server_error("sec_key_mgmt_get_key()", *status);
return;
}

/* Validate the login context...                                    */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_validate_identity()"));
sec_login_validate_identity(*login_context,
keydata,
&reset_pwd,
```

```
&auth_src,
status);
if (*status != error_status_ok)
{
print_server_error("sec_login_validate_identity()", *status);
return;
}

/* Finally, set the context...                                         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_set_context()"));
sec_login_set_context(*login_context, status);
if (*status != error_status_ok)
{
print_server_error("sec_login_set_context()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_get_identity()"));

}


/******
 *
 * server_renew_identity -- Make sure that credentials are still valid, and
 *                          enew them if they are not.
 *
 *
 *     This routine is called (with the current credentials) whenever a task
 *     is about to be attempted that requires valid credentials. For an ex-
 *     ample, see the cleanup code in "main()" above. A valid credential will
 *     nevertheless be considered invalid if it will expire within time_left
 *     seconds. This gives a margin of time between the validity check that
 *     occurs here and the actual use of the credential.
 *
 *    Called from main() (but can be called from elsewhere).
 *
 ******/

void server_renew_identity(
unsigned_char_p_t prin_name,      /* Server's principal name.             */
sec_login_handle_t login_context, /* Server's login context.              */
unsigned_char_p_t keytab,         /* Local key file.                      */
unsigned32 time_left,             /* Amount of "margin" -- see above.     */
unsigned32 *status)               /* To return status.                    */
{
signed32 expiration;
time_t current_time;
sec_passwd_rec_t *keydata;
sec_login_auth_src_t auth_src;
boolean32 reset_pwd;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_renew_identity()"));

*status = error_status_ok;

/* Get the lifetime for the server's Ticket-Granting-Ticket (TGT).  */
/* Note that sec_login_get_expiration() returns a non-zero          */
/*  status for an uncertified login context. This is not            */
/*  an error. Hence the special error checking...                   */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_get_expiration()"));
sec_login_get_expiration(login_context,
&expiration,
```

```
status);
if (*status != sec_login_s_not_certified)
{
print_server_error("sec_login_validate_identity()", *status);
return;
}

/* Get current time...                                            */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling time()"));
time(&current_time);

/* Now, if the expiration time is sooner than the desired "time     */
/*  left"...                                                        */
if (expiration < (current_time + time_left))
{
/* Refresh the server's authenticated identity...          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_refresh_identity()"));
sec_login_refresh_identity(login_context,
    status);
if (*status != error_status_ok)
{
print_server_error("sec_login_refresh_identity()", *status);
return;
}

/* Get key from local file...                              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_key_mgmt_get_key()"));
sec_key_mgmt_get_key(rpc_c_authn_dce_secret,
keytab,
prin_name,
0,
(void**)&keydata,
status);
if (*status != error_status_ok)
{
print_server_error("sec_key_mgmt_get_key()", *status);
return;
}

/* Validate the login context...                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_login_validate_identity()"));
sec_login_validate_identity(login_context,
keydata,
&reset_pwd,
&auth_src,
status);
if (*status != error_status_ok)
{
print_server_error("sec_login_validate_identity()", *status);
return;
}
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Successfully exiting server_renew_identity()"));

}


/******
 *
 * server_create_dflt_acl -- Create a default ACL; i.e., get the initial
 *                           container ACL and copy it (instead of con-
```

```
 *                            structing an ACL whole, as below), and create
 *                            a UUID for the ACL.
 *
 *    Not called from anywhere. NOT YET TESTED.
 *
 ******/

void server_create_dflt_acl(
dce_db_handle_t db_acl,           /* Backing store handle.  */
unsigned_char_t *container,  /* Object we want the ACL of.  */
void (*resolver)(),  /* ACL name-to-UUID resolver function; */
/*  i.e., sample_resolve_by_name(). */
boolean32 is_container,      /* Is the object a container?  */
sec_acl_t *acl,              /* ACL will be returned here.  */
uuid_t *acl_uuid,     /* ACL's UUID will be returned here.  */
unsigned32 *status)
{

sec_acl_type_t sec_acl_type;     /* To contain ACL type specifier.  */
uuid_t iacl_uuid;                /* To contain initial container    */
/*  ACL's UUID.            */

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_create_dflt_acl()"));

/* Create the UUID for the new ACL...                                */
*status = error_status_ok;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_create()"));
uuid_create(acl_uuid, status);
if (*status != error_status_ok)
{
print_server_error("uuid_create()", *status);
return;
}

if (is_container)
sec_acl_type = 2;
else
sec_acl_type = 1;

/* Now get the initial container's ACL UUID.                         */
/* This is a call to sample_resolve_by_name(); see below...          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling (*resolver)()"));
(*resolver)(NULL,  /* No client binding handle; this isn't a re-    */
/* mote call.                             */
container, /* The object whose ACL's UUID we want;  */
/* here, the initial container.           */
sec_acl_type, /* Type of ACL we want UUID for.      */
NULL, /* No manager type specified.                 */
0,    /* Dummy parameter for us.                    */
NULL, /* No need to specify a special backing       */
/*  store handle.                         */
&iacl_uuid,  /* Initial container ACL's UUID is re- */
/*  turned here.                          */
status);
if (*status != error_status_ok)
{
print_server_error("resolver function(*)", *status);
return;
}

/* Now get the initial container ACL...                              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_fetch_by_uuid()"));
dce_db_fetch_by_uuid(db_acl,          /* ACL UUID-indexed database. */
```

```
&iacl_uuid,      /* The initial container ACL UUID. */
acl,                  /* The ACL is returned here.  */
status);
if (*status != error_status_ok)
{
print_server_error("dce_db_fetch_by_uuid()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_create_dflt_acl()"));

}




/******
 *
 * server_get_local_principal_id -- Get (from the local cell registry) the
 *                                  UUID corresponding to a principal name.
 *
 *
 *    Called from server_create_acl() and server_acl_mgr_setup().
 *
 ******/

void server_get_local_principal_id(
unsigned_char_t *p_name,    /* Simple principal name.       */
uuid_t *p_id,               /* UUID returned here.          */
unsigned32 *status)         /* Status returned here.        */
{

char *cell_name;                    /* For local cell name.         */
sec_rgy_handle_t rhandle;           /* For registry server handle.  */

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_get_local_principal_id()"));

/* First, get the local cell name...                                */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_cf_get_cell_name()"));
dce_cf_get_cell_name(&cell_name, status);
if (*status != error_status_ok)
{
print_server_error("dce_cf_get_cell_name()", *status);
return;
}

/* Now bind to the cell's registry...                               */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sec_rgy_site_open()"));
sec_rgy_site_open(cell_name, &rhandle, status);
if (*status != error_status_ok)
{
print_server_error("sec_rgy_site_open()", *status);
return;
}

/* Free the string space we got the cell name in...                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling free()"));
free(cell_name);

/* Now get from the registry the UUID associated with the principal */
/*  name we got in the first place...                               */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
```

```
"Calling sec_rgy_pgo_name_to_id()"));
sec_rgy_pgo_name_to_id(rhandle,
sec_rgy_domain_person,
p_name,
p_id,
status);
if (*status != error_status_ok)
{
print_server_error("sec_rgy_pgo_name_to_id()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_get_local_principal_id()"));

}



/******
 *
 * server_create_acl -- Create an ACL with some specified set of permissions
 *                      assigned to some principal user.
 *
 *
 *    Called from server_acl_mgr_setup().
 *
 ******/

void server_create_acl(
uuid_t mgr_type_uuid,     /* Manager type of ACL to create. */
sec_acl_permset_t perms,  /* Permission set for ACL.        */
unsigned_char_t *user,    /* Principal name for new entry.  */
sec_acl_t *acl,           /* To return the ACL entry in.    */
uuid_t *acl_uuid,         /* To return the ACL's UUID in.   */
unsigned32 *status)       /* To return status in.           */
{

static uuid_t u;  /* For the principal's UUID (from the registry).  */

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_create_acl()"));

*status = error_status_ok;

/* Create a UUID for the ACL...                                     */
/* Note that the new UUID doesn't get associated with the entry in  */
/*  this routine. It must happen in server_acl_mgr_setup()...       */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_create()"));
uuid_create(acl_uuid, status);
if (*status != error_status_ok)
{
print_server_error("uuid_create()", *status);
return;
}

/* Create an initial ACL object with default permissions for the    */
/*  designated user principal identity...                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_init()"));
dce_acl_obj_init(&mgr_type_uuid, acl, status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_init()", *status);
return;
}
```

```
/* Get the specified principal's UUID...                               */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_get_local_principal_id()"));
server_get_local_principal_id(user, &u, status);
if (*status != error_status_ok)
{
print_server_error("server_get_local_principal_id()", *status);
return;
}

/* Now add the user ACL entry to the ACL...                            */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_add_user_entry()"));
dce_acl_obj_add_user_entry(acl, perms, &u, status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_add_user_entry()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_create_acl()"));

}


/******
 *
 * server_store_acl -- Store ACL-related data.    *
 *
 *      The data is stored in databases that support a
 *      name->object_uuid->acl_uuid style of ACL lookup.
 *
 *
 *    Called from server_acl_mgr_setup().
 *
 ******/
/******************************************************************************/
/*                                                                            */
/* There are three databases (this can be seen also in the                    */
/*  server_acl_mgr_close() routine):                                          */
/*                                                                            */
/*   db_acl:ACL (UUID)-indexed:        (used to store the ACLs themselves) */
/*   db_object:Object (UUID)-indexed:   (used to store the object data     */
/*                                       itself)                              */
/*   db_name:Name ("Residual")-indexed: (used to store the simple names of  */
/*                                       the objects)                         */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/******************************************************************************/

void server_store_acl(
dce_db_handle_t db_acl,     /* ACL (UUID)-indexed store.    */
dce_db_handle_t db_object,  /* Object (UUID)-indexed store. */
dce_db_handle_t db_name,    /* Name-indexed store.          */
sec_acl_t *acl,             /* The ACL itself.              */
uuid_t *acl_uuid,           /* ACL UUID.                    */
uuid_t *object_uuid,        /* Object UUID.                 */
unsigned_char_t *object_name, /* The name of the object.    */
void *object_contents,  /* The actual object data contents. */
/* NOTE: NOT USED NOW.              */
boolean32 is_container,  /* Are we storing a container ACL? */
unsigned32 *status)         /* To return status.            */
{
```

```
/* These two variables are used to hold UUIDs for the ACLs we will  */
/*  need to create if we have a container ACL on our hands...       */
static uuid_t def_object, def_container;
static sample_data_t object_data;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_store_acl()"));

*status = error_status_ok;

/* Null the contents of the object_data variable...                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling bzero()"));
bzero((char *)&object_data, sizeof object_data);

/* If we have a container ACL, then we have to create and store the */
/*  special stuff associated with it-- namely, the container ACL    */
/*  itself, and a default object ACL to go with it...               */
if (is_container)
{
/* Create a UUID for the default object ACL...              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_create()"));
uuid_create(&def_object, status);
if (*status != error_status_ok)
{
print_server_error("uuid_create()", *status);
return;
}
/* Create a UUID for the default container ACL...           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_create()"));
uuid_create(&def_container, status);
if (*status != error_status_ok)
{
print_server_error("uuid_create()", *status);
return;
}

/* Store the default object ACL into UUID-indexed store...  */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_store_by_uuid()"));
dce_db_store_by_uuid(db_acl, &def_object, acl, status);
if (*status != error_status_ok)
{
print_server_error("dce_db_store_by_uuid()", *status);
return;
}

/* Store the default container ACL into UUID-indexed        */
/*  store...                                                */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_store_by_uuid()"));
dce_db_store_by_uuid(db_acl, &def_container, acl, status);
if (*status != error_status_ok)
{
print_server_error("dce_db_store_by_uuid()", *status);
return;
}

        }

/* Store the plain object ACL into ACL UUID-indexed store...        */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_store_by_uuid()"));
dce_db_store_by_uuid(db_acl, acl_uuid, acl, status);
if (*status != error_status_ok)
```

```
{
print_server_error("dce_db_store_by_uuid()", *status);
return;
}

/* Store the ACL UUID(s) into a standard object header...          */

/* Observe how this all seems to work: the following call sets up   */
/*  some general info in the object-indexed database that is asso-  */
/*  ciated with the Object and ACL UUIDs passed. This is where the  */
/*  ACL on an object actually gets connected with that object. Up   */
/*  above the information connected with the ACL UUID was stored    */
/*  in a sec_acl_t structure, but it's the following call that ac-  */
/*  tually associates this structure with some object. Afterward    */
/*  the object data itself (what is being "ACL'd") gets stored via  */
/*  the dce_db_store_by_uuid() call, and finally the object UUID    */
/*  itself is stored by name via the dce_db_store_by_name() call.   */
/*  So the reverse process (beginning with the name) will be:       */
/*                                                                  */
/*    1. Look up the object UUID by name by calling                 */
/*       dce_db_fetch_by_name().                                    */
/*                                                                  */
/*    2. Look up the data (i.e., object data) for the object        */
/*       by calling dce_db_fetch_by_uuid().                         */
/*                                                                  */
/*    3. Extract the ACL UUID from the correct field in the object  */
/*       data structure.                                            */
/*                                                                  */
/*  ...These steps can be seen in sample_resolve_by_name(), the     */
/*  purpose of which is to return an ACL UUID when given an object  */
/*  name; the permission lists in effect for the object can then be */
/*  accessed and checked against some set of permissions presented  */
/*  by a prospective accessor.                                      */
/*                                                                  */
/* Once the ACL library has gotten from us the UUID that identifies */
/*  the ACL on the object it wants to investigate the permissions   */
/*  on, it's up to it to go on to retrieve the ACL itself, using    */
/*  the UUID to do so. It is able to do this because we have reg-   */
/*  istered our ACL database via the dce_acl_register_object_type() */
/*  call (this is also, by the way, where our *(resolver)() routine */
/*  is registered). So the runtime can extract the ACL information, */
/*  compare it with the permissions presented by the entity that's  */
/*  trying to access the object in question, and allow, or not al-  */
/*  low, the operation to proceed accordingly.                      */
/*                                                                  */
/* The way to test whether our sample ACLs have been set up cor-    */
/*  rectly or not would be to try to do various things to them via  */
/*  acl_edit.                                                       */
/*                                                                  */
/* Note that the registration procedures described here are only to */
/*  set up an application's ACL manager so that it is accessible    */
/*  via acl_edit (and, I suppose, dced and dcecp). In situations    */
/*  where a client in contact with the application server itself is */
/*  trying to perform some operation, it is the responsibility of   */
/*  the application code itself to check the client's authorization */
/*  and make the correct decision as to access. Note though that it */
/*  does this through the dce_acl_is_client_authorized() call,      */
/*  which again can work only if you have correctly registered the  */
/*  application's manager. For an example of using this call see    */
/*  the sample_call() code in sample_manager.c; it is also called   */
/*  by sample_mgmt_auth(), below in this file.                      */
/*                                                                  */
/*                                                                  */
/* To sum up, then, there are basically three avenues of access     */
/*  that an application has to provide for when setting up an ACL   */
/*  manager:                                                        */
/*                                                                  */
```

```
/*      1. Access by clients to the server via the remote mgmt      */
/*         interface. This is handled by setting up a mgmt call-    */
/*         back routine that will be automatically invoked by the   */
/*         runtime whenever a remote mgmt access is attempted. Our  */
/*         callback routine is sample_mgmt_auth(), below, and it is */
/*         registered by a call to rpc_mgmt_set_authorization().    */
/*                                                                  */
/*      2. Access by entities of any kind via acl_edit. This is     */
/*         handled by the mechanisms described above, which are     */
/*         set up by the call to dce_acl_register_object_type().    */
/*                                                                  */
/*      3. Access by clients in contact with the server. This is    */
/*         handled by the server code itself, as described above.   */
/*                                                                  */
/*                                                                  */
/*                                                                  */
/* Note that the use of the three databases given here is necessar- */
/*  ily true only of the ACL databases we are setting up here. The  */
/*  object data stored in databases is strictly up to the applica-  */
/*  tion; that is why this parameter is defined as (void *).        */
/*  In other words, the backing store library can be used for any-  */
/*  thing.                                                          */
/*                                                                  */
/*  In the sample_db.idl file can be seen the object data type de-  */
/*  fined for this sample application, which is stored in the ob-   */
/*  ject UUID-indexed database...                                   */


DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_std_header_init()"));
dce_db_std_header_init(
db_object,      /* Object database.         */
&(object_data.s_hdr), /* Object data hdr.   */
object_uuid,    /* Object UUID.             */
acl_uuid,       /* ACL UUID.                */
&def_object,    /* Default object ACL.      */
&def_container, /* Default container ACL.   */
0,              /* Reference count.         */
status);

if (*status != error_status_ok)
{
print_server_error("dce_db_std_header_init()", *status);
return;
}

/* Now store the object data keyed by object UUID...               */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_store_by_uuid()"));


/* This is perhaps a not very nice way to do this, but it will have */
/*  to serve for the time being...                                  */
if (strcmp((char *)object_name, SAMPLE_OBJECT_NAME) == 0)
strcpy((char *)object_data.s_data.message,
"THIS IS AN OFFICIAL SAMPLE OBJECT TEXT!");
else if (strcmp((char *)object_name, MGMT_OBJ_NAME) == 0)
strcpy((char *)object_data.s_data.message,
"THIS IS AN OFFICIAL MGMT OBJECT SAMPLE TEXT!");
else
strcpy((char *)object_data.s_data.message,
"I DON'T KNOW WHAT THIS IS!");

dce_db_store_by_uuid(db_object,
object_uuid,
(void *)&object_data,
status);
```

```
if (*status != error_status_ok)
{
print_server_error("dce_db_store_by_uuid()", *status);
return;
}


/* Finally, store the object UUID keyed by the object ("residual")  */
/*  name...                                                         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_store_by_name(%s)", object_name));
dce_db_store_by_name(db_name, (char *)object_name, object_uuid,
status);
if (*status != error_status_ok)
{
print_server_error("dce_db_store_by_name()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_store_acl()"));

}


/******
 *
 * server_acl_mgr_setup -- Open and, if necessary, create the ACL-related
 *                         databases, i.e.:
 *
 *        1. Set up a default ACL manager for the management interface.
 *
 *        2. Create an initial ACL. For servers that dynamically create
 *           objects, this ACL is intended to be used as the ACL on the
 *           "container" in which objects are created. If the server
 *           manages static objects, this ACL can be used for some other
 *           purpose.
 *
 *           Note that all the container-related code is actually in the
 *           server_store_acl() routine above.
 *
 *    Called from main().
 *
 ******/

void server_acl_mgr_setup(
unsigned_char_t *db_acl_path,    /* Pathname for databases. */
dce_acl_resolve_func_t resolver, /* sample_resolve_by_name. */
uuid_t acl_mgr_uuid,             /* ACL manager UUID.       */
uuid_t object_uuid,              /* Object UUID.            */
unsigned_char_t *object_name,    /* Object name.            */
sec_acl_permset_t owner_perms,   /* Owner permission set.   */
unsigned_char_t *owner,          /* Owner name.             */
boolean32 is_container,      /* Is this a container object? */
/* == TRUE from main().     */
/* [out] parameters: */
dce_db_handle_t *db_acl,    /* ACL-indexed store handle.    */
dce_db_handle_t *db_object, /* Object-indexed store handle. */
dce_db_handle_t *db_name,   /* Name-indexed store handle.   */
uuid_t *object_acl_uuid,    /* Object ACL UUID.             */
uuid_t *mgmt_acl_uuid,      /* Mgmt ACL UUID.               */
unsigned32 *status)
{

uuid_t machine_princ_id;
unsigned_char_t machine_principal[MAXHOSTNAMELEN + 20];
unsigned_char_t *uuid_string;
```

```
sec_acl_t *new_obj_acl, *new_mgmt_acl;
boolean32 need_init;
unsigned32 dbflags;
static sample_data_t object_data;
unsigned_char_t *acl_path_string;
sec_acl_permset_t permset = (sec_acl_permset_t) 0;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_acl_mgr_setup()"));

new_obj_acl = (sec_acl_t *)malloc(sizeof(sec_acl_t));
new_mgmt_acl = (sec_acl_t *)malloc(sizeof(sec_acl_t));

*status = error_status_ok;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling bzero()"));
bzero((char *)&object_data, sizeof object_data);

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_create_nil()"));
uuid_create_nil(object_acl_uuid, status);
if (*status != error_status_ok)
{
print_server_error("uuid_create_nil()", *status);
return;
}

need_init = 0;

/* Build the full pathname string for the db_acl database...        */
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy((char *)acl_path_string, (char *)db_acl_path);
strcat((char *)acl_path_string, (char *)"/");
strncat((char *)acl_path_string, "db_acl", strlen("db_acl"));

/* Find out if the database already exists...                       */
if (access((char *)acl_path_string, R_OK) != 0)
if (errno == ENOENT)
need_init = 1;

/*******************************************************************/

/* Create the indexed-by-UUID databases. There are two of these:    */
/*       One for the ACL UUID-indexed store, and                    */
/*       One for the Object UUID-indexed store...                   */

dbflags = db_c_index_by_uuid;

/* If the thing doesn't exist yet, then we need to do some init-    */
/*  ialization...                                                   */
if (need_init)
dbflags |= db_c_create;

/* Open (or create) the "db_acl" ACL UUID-indexed backing store.    */
/*  Note that no header type is specified among the dbflags, so the */
/*  database will be created with no header-- that's the default... */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_open()"));

dce_db_open(
(char *)acl_path_string,  /* Filename of backing store.     */
NULL,      /* Backing store "backend type" default == hash. */
dbflags,   /* We already specified index by UUID for this.  */
(dce_db_convert_func_t)dce_rdacl_convert,  /* Serialization */
/* function (generated by IDL).            */
db_acl,    /* The returned backing store handle.           */
status);
```

```
if (*status != error_status_ok)
{
print_server_error("dce_db_open()", *status);
free(acl_path_string);
return;
}
/* Set the global variable that records whether we actually have    */
/*  opened the databases; this enables us to avoid calling the      */
/*  dce_db_close() routine for unopened databases, which will cause */
/*  a core dump...                                                  */
databases_open = TRUE;

/* For the object database, we need standard backing store headers  */
/*  to hold UUIDs for all the various ACLs...                       */
dbflags |= db_c_std_header;
if (need_init)
dbflags |= db_c_create;

/* Now open (or create) the "db_object" store... */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_open()"));

/* Build the full pathname string for the database...               */
free(acl_path_string);
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy((char *)acl_path_string, (char *)db_acl_path);
strcat((char *)acl_path_string, (char *)"/");
strncat((char *)acl_path_string, "db_object", strlen("db_object"));

dce_db_open(
(char *)acl_path_string, /* Filename of backing store.      */
NULL,      /* Backing store "backend type" default == hash. */
dbflags,   /* Specifies index by UUID, and include standard */
/* headers.                              */
(dce_db_convert_func_t)sample_data_convert, /* Serializa-   */
/* tion function for object data.        */
db_object,    /* The returned backing store handle.        */
status);
if (*status != error_status_ok)
{
print_server_error("dce_db_open()", *status);
free(acl_path_string);
return;
}


/* Create the indexed-by-name database...                           */

dbflags = db_c_index_by_name;
if (need_init)
dbflags |= db_c_create;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_open()"));

/* Build the full pathname string for the database...               */
free(acl_path_string);
acl_path_string = malloc(MAX_ACL_PATH_SIZE);
strcpy((char *)acl_path_string, (char *)db_acl_path);
strcat((char *)acl_path_string, (char *)"/");
strncat((char *)acl_path_string, "db_name", strlen("db_name"));

dce_db_open(
(char *)acl_path_string,  /* Filename of backing store.     */
NULL,      /* Backing store "backend type" default == hash. */
dbflags,   /* Specifies index by name.                      */
(dce_db_convert_func_t)uu_convert, /* Serialization func-   */
```

```
/* tion for name data.                      */
db_name,   /* The returned backing store handle.            */
status);
if (*status != error_status_ok)
{
print_server_error("dce_db_open()", *status);
free(acl_path_string);
return;
}
free(acl_path_string);

/*******************************************************************/

/* Now register our ACL manager's object types with the ACL        */
/*  library...                                                     */

/* Register for the mgmt ACL...                                    */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_register_object_type()"));
dce_acl_register_object_type(
*db_acl,  /* Backing store where ACLs are to be stored.     */
&mgmt_acl_mgr_uuid,     /* Type of ACL manager: this one is */
/* for mgmt ACL operations; the UUID is defined     */
/* globally at the top of this file.                */
/* Why do we need this parameter? Well, the way     */
/*  that the ACL library keeps track of the differ- */
/*  ent "sets" of ACL databases is by manager UUID. */
/*  The manager UUID is what the library will use   */
/*  to figure out which ACL database to open and    */
/*  retrieve a requested ACL's contents from.       */
/*  Essentially what we are doing here is setting   */
/*  up things so that calls to the library routine  */
/*  dce_acl_is_client_authorized() can be made to   */
/*  check our ACLs, giving only the ACL UUID and a  */
/*  manager UUID to get the desired result.         */

sizeof mgmt_printstr/sizeof mgmt_printstr[0], /* Number of  */
/* items in mgmt_printstr array.            */
mgmt_printstr,  /* An array of sec_acl_printstring_t struc- */
/*  tures containing the printable repre-   */
/*  sentation of each specified permission. */
&mgmt_info,     /* A single sec_acl_printstring_t contain-  */
/*  ing the name and short description for  */
/*  the given ACL manager.                  */
sec_acl_perm_control,  /* Permission set needed to change   */
/* an ACL. Constants like these are defined */
/* in <dce/aclbase.h>.                      */
sec_acl_perm_test, /* Permission set needed to test an ACL. */

resolver,  /* Server function to get ACL UUID for a given   */
/*  object; for us it's the                 */
/*  sample_resolve_by_name() call, below.   */
/* This routine is for the use of acl_edit: */
/*  it allows acl_edit to receive an object */
/*  name and come up with the ACL UUID; at  */
/*  least that's what I think it's for.     */
NULL,           /* Argument to pass to resolver function.   */
0,              /* Flags -- none here.                      */
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_register_object_type()", *status);
return;
}

/* Now register for the regular ACL...                             */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
```

```
"Calling dce_acl_register_object_type()"));
dce_acl_register_object_type(
*db_acl, /* Backing store where ACLs are to be stored.      */
&sample_acl_mgr_uuid,  /* Hard-coded at top of this file.   */
sizeof sample_printstr/sizeof sample_printstr[0], /* Number */
/* of items in our printstring array.       */
sample_printstr, /* An array of sec_acl_printstring_t       */
/* structures containing the printable rep- */
/* resentation of each specified permis-    */
/* sion set.                                */
&sample_info,   /* A single sec_acl_printstring_t contain-  */
/*  ing the name and short description for  */
/*  the manager we're registering.          */
sec_acl_perm_control, /* Permission set needed to change an */
/*  ACL.                                    */
sec_acl_perm_test,    /* The permission you need to test an */
/* ACL maintained by this manager.  */

resolver,       /* Application server function that gives   */
/*  the ACL UUID for a given object, when   */
/*  presented with that object's name; for  */
/*  us it's the sample_resolve_by_name()    */
/*  routine, below.                         */
NULL,           /* Argument to pass to resolver routine;    */
/*  identified as the "resolver_arg" in the */
/*  code to that function below.            */
0,              /* Flags -- none here.                      */
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_register_object_type()", *status);
return;
}

/* If we're initializing, then we have to create all this stuff...  */
if (need_init)
{

dce_svc_printf(NO_ACL_DBS_MSG);
/* Create the mgmt interface ACL...                         */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_create_acl()"));
server_create_acl(
mgmt_acl_mgr_uuid, /* Create mgmt manager type ACL. */
ALL_MGMT_PERMS, /* Permission set for new ACL.      */
owner,          /* Principal name for new entry.    */
new_mgmt_acl,   /* This will contain the new ACL.   */
mgmt_acl_uuid,  /* This will contain the ACL UUID.  */
status);
if (*status != error_status_ok)
{
print_server_error("server_create_acl()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(mgmt_acl_uuid, &uuid_string, status);
if (*status != uuid_s_ok)
{
print_server_error("uuid_to_string()", *status);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of mgmt_acl_uuid == %s",
uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
```

```
rpc_string_free(&uuid_string, status);
if (*status != rpc_s_ok)
{
print_server_error("rpc_string_free()", *status);
}


/***********************************************************/
/* For the management ACL we must add a default entry for   */
    /*  the machine principal so dced can manage the server.    */

/* Construct the name entry string...                       */
strcpy((char *)machine_principal, "hosts/");
gethostname((char *)(machine_principal + 6), MAXHOSTNAMELEN + 1);
strcat((char *)machine_principal, "/self");

/* Get the machine principal's UUID...                      */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_get_local_principal_id()"));
server_get_local_principal_id(machine_principal,
&machine_princ_id,
status);
if (*status != error_status_ok)
}
print_server_error("server_get_local_principal_id()",
*status);
return;
}

/* Add a user entry for the machine principal to the new    */
/*  ACL...                                                  */
permset = ALL_MGMT_PERMS;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_add_user_entry()"));
dce_acl_obj_add_user_entry(new_mgmt_acl,
permset,
&machine_princ_id,
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_add_user_entry()",
*status);
return;
}

/* By default everybody must be able to get the principal   */
/*  name. They should be able to ping too. So add an appro- */
/*  priate unauthenticated permissions entry to the ACL...  */
permset = mgmt_perm_inq_pname | mgmt_perm_ping;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_add_unauth_entry()"));
dce_acl_obj_add_unauth_entry(
new_mgmt_acl,
permset,
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_add_unauth_entry()",
*status);
return;
}

/* Add permissions for the any_other entry in the ACL...    */
permset = mgmt_perm_inq_pname | mgmt_perm_ping;
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_add_any_other_entry()"));
dce_acl_obj_add_any_other_entry(
```

```
new_mgmt_acl,
permset,
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_add_any_other_entry()",
*status);
return;
}

/* Store the mgmt ACL...                                   */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_store_acl()"));
server_store_acl(
*db_acl,     /* The ACL UUID-indexed store.        */
*db_object,  /* The object UUID-indexed store.     */
*db_name,  /* The name ("residual")-indexed store.  */
new_mgmt_acl,       /* The ACL itself.             */
mgmt_acl_uuid,      /* The mgmt ACL UUID.          */
&mgmt_object_uuid,  /* The mgmt object UUID.       */
(unsigned_char_t *)MGMT_OBJ_NAME,  /* The mgmt ob-  */
/*  ject name.            */
/* (void*) */ &object_data, /* The object contents. */
0,                  /* Not a container ACL.        */
status);

if (*status != error_status_ok)
{
print_server_error("server_store_acl()", *status);
return;
}


/*****************************************************************/
/* Object ACL creation code...                                    */

/* Now create the object ACL...                            */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_create_acl()"));
server_create_acl(
sample_acl_mgr_uuid,  /* Create an ACL with this    */
/*  manager type.                   */
owner_perms,   /* Give it these permissions.       */
owner,         /* Make this the principal name.    */
new_obj_acl,   /* This will contain new ACL.       */
object_acl_uuid, /* This will contain new ACL UUID. */
status);
if (*status != error_status_ok)
{
print_server_error("server_create_acl()", *status);
return;
}

/* Null the data header...                                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling bzero()"));
bzero((char *)&object_data, sizeof object_data);

/* Store the object ACL...                                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling server_store_acl()"));
server_store_acl(
*db_acl,     /* The ACL UUID-indexed store.        */
*db_object,  /* The object UUID-indexed store.     */
*db_name,  /* The name ("residual")-indexed store.  */
new_obj_acl,        /* The ACL itself.             */
object_acl_uuid,    /* The object ACL UUID.        */
```

```
&object_uuid,       /* The object UUID.             */
object_name,        /* The object name.             */
/* (void*) */ &object_data, /* The object contents. */
/* is_container */  0,  /* Is this a container ACL? */
status);
if (*status != error_status_ok)
{
print_server_error("server_store_acl()", *status);
return;
}

/* Finally, free the space we were using...                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_obj_free_entries()"));
dce_acl_obj_free_entries(new_obj_acl, status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_obj_free_entries()",
*status);
return;
}

/* ...end of object ACL creation code.                                */
/*******************************************************************/

}
        else /* ACL databases already exist; get the two ACL UUIDs...
   */
{

/* This is a call to sample_resolve_by_name() (see below);  */
/*  it gives us the UUID of the ACL of the object whose     */
/*  name we pass it...                                      */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling (*resolver)()"));
(*resolver)(
NULL,         /* No client bind handle; local call. */
object_name,  /* Object whose ACL UUID we want.     */
0,            /* Type of ACL we want UUID of.       */
&sample_acl_mgr_uuid, /* Object's manager type.     */
0,            /* Ignored as far as we're concerned. */
NULL,         /* "resolver_arg"; unused.            */
object_acl_uuid, /* Will contain object ACL UUID.   */
status);
if (*status != error_status_ok)
{
print_server_error("resolver function (*)", *status);
return;
}


DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug5,
"Calling (*resolver)()"));
(*resolver)(
NULL,         /* No client bind handle; local call. */
(sec_acl_component_name_t)MGMT_OBJ_NAME, /* We want */
/* mgmt object's ACL UUID.  */
0,             /* Type of ACL we want UUID of.      */
&mgmt_acl_mgr_uuid, /* Object's manager type=mgmt.  */
0,            /* Ignored as far as we're concerned. */
NULL,         /* "resolver_arg"; ignored.           */
mgmt_acl_uuid, /* Will contain mgmt ACL UUID.       */
status);
if (*status != error_status_ok)
{
print_server_error("resolver function (*)", *status);
return;
```

```
}

   }


DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(mgmt_acl_uuid, &uuid_string, status);
if (*status != uuid_s_ok)
{
print_server_error("uuid_to_string()", *status);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of mgmt_acl_uuid == %s", uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, status);
if (*status != rpc_s_ok)
{
print_server_error("rpc_string_free()", *status);
}


/* Set up remote management authorization to use the ACL manager.   */
/*  Note that the first parameter to this call is the address of a  */
/*  management authorization callback routine, which is defined     */
/*  later in this file...                                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_mgmt_set_authorization_fn()"));
rpc_mgmt_set_authorization_fn(sample_mgmt_auth, status);
if (*status != error_status_ok)
{
print_server_error("rpc_mgmt_set_authorization_fn()", *status);
return;
}

/* Finally, register the rdacl interface with the runtime...        */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_register_if()"));
rpc_server_register_if(
rdaclif_v1_0_s_ifspec, /* Interface to register.    */
NULL,                  /* Manager type UUID.        */
(rpc_mgr_epv_t) &dce_acl_v1_0_epv, /* Entry point   */
/* vector.          */
status);
if (*status != error_status_ok)
{
print_server_error("rpc_server_register_if()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_acl_mgr_setup()"));

}


/******
 *
 * server_acl_mgr_close --  Called at cleanup time to close
 *                          the three ACL databases.
 *
 *
 *    Called from main().
 *
 ******/
```

```
void server_acl_mgr_close(
dce_db_handle_t *db_acl,          /* ACL UUID-indexed database.
   */
dce_db_handle_t *db_object,       /* Object UUID-indexed database.
   */
dce_db_handle_t *db_name,         /* Name-indexed database.
   */
unsigned32 *status)
{

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_acl_mgr_close()"));

*status = error_status_ok;

/* Close the ACL UUID-indexed database...                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_close()"));
dce_db_close(db_acl, status);
if (*status != error_status_ok)
{
print_server_error("dce_db_close()", *status);
return;
}

/* Close the Object UUID-indexed database...                       */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_close()"));
dce_db_close(db_object, status);
if (*status != error_status_ok)
{
print_server_error("dce_db_close()", *status);
return;
}

/* Close the name-indexed database...                              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_close()"));
dce_db_close(db_name, status);
if (*status != error_status_ok)
{
print_server_error("dce_db_close()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_acl_mgr_close()"));

}


/******
 *
 * server_rdacl_export -- Make the rdacl interface available
 *                        for ACL editors.
 *
 *
 *     Note that we don't export to the namespace. Instead, the ACL editor
 *      will typically bind to the server via some other entry that holds
 *      the application-specific interface bindings. For our application,
 *      that entry is:
 *
 *             /.:/sample_server_entry
 *
 *      ...This entry (the "junction" to the object "entries") must hold at
 *      least one object UUID, and the same UUID must be put into the end-
 *      point map too. If not, ACL editors will have no way to distinguish
```

```
 *      the endpoints of this server from those of other servers on the same
 *      host that also export the rdacl interface.
 *
 *    Called from main().
 *
 ******/

void server_rdacl_export(
rpc_binding_vector_t *binding_vector, /* Binding handles from RPC
runtime.  */
uuid_vector_t *object_uuid_vector,    /* Server instance UUID(s).          */
unsigned32 *status)
{

uuid_vector_t my_uuids;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_rdacl_export()"));

*status = error_status_ok;

/* Register the server's endpoints with the rdacl interface at the  */
/*  local endpoint map...                                            */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_register()"));
rpc_ep_register(rdaclif_v1_0_s_ifspec,
binding_vector,    /* Our binding handles from RPC runtime. */
object_uuid_vector,  /* Server instance UUID (only one).    */
(unsigned_char_p_t) "rdacl interface", /* Annotation.       */
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_register()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_rdacl_export()"));

}


/******
 *
 * server_rdacl_cleanup -- Called at cleanup time to
 *                         unregister the rdacl interface.
 *
 *
 *    Called from main().
 *
 ******/

void server_rdacl_cleanup(
rpc_binding_vector_t *binding_vector,  /* Binding handles from RPC
runtime. */
uuid_vector_t *object_uuid_vector,     /* Server instance UUID(s).
   */
unsigned32 *status)
{

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_rdacl_cleanup()"));

*status = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_unregister()"));
```

```
rpc_ep_unregister(rdaclif_v1_0_s_ifspec,
binding_vector,
object_uuid_vector,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_unregister()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_rdacl_cleanup()"));

}



/******
 *
 * server_bind_cleanup -- Called at cleanup time to
 *                        unregister the remote bind interface.
 *
 *
 *    Called from main().
 *
 ******/

void server_bind_cleanup(
rpc_binding_vector_t *binding_vector,  /* Binding handles from RPC runtime. */
uuid_vector_t *object_uuid_vector,     /* Server instance UUID(s).         */
unsigned32 *status)
{

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering server_bind_cleanup()"));

*status = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_unregister()"));
rpc_ep_unregister(sample_bind_v1_0_s_ifspec,
binding_vector,
object_uuid_vector,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_unregister()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting server_bind_cleanup()"));

}



/******************************************************************************/
/* End of server init and cleanup functions                                   */
/******************************************************************************/



/******
 *
```

```
 * signal_handler -- Thread to handle asynchronous interrupts.
 *
 *  Catch and handle SIGINT and SIGTERM. Note that we
 *  don't use sigaction() here because it won't work with
 *  asynchronous signals. Also note that signals must be
 *  blocked prior to being waited for.
 *
 *
 *    The thread that runs this function is started in main().
 *
 ******/

void signal_handler(char *arg)
{
sigset_t signals;  /* Set of signals available to the application.  */
int sig;
unsigned32 status;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering signal_handler()"));

status = error_status_ok;

/* Initialize the signal set...                                      */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigemptyset()"));
sigemptyset(&signals);

/* Add SIGINT to signal set...                                       */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigaddset()"));
sigaddset(&signals, SIGINT);

/* Add SIGTERM to signal set...                                      */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigaddset()"));
sigaddset(&signals, SIGTERM);

/* Set the current signal mask...                                    */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigprocmask()"));
sigprocmask(SIG_BLOCK, &signals, NULL);

/* And now wait for the signals...                                   */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigwait()..."));
while (1)
{
sig = sigwait(&signals);
switch (sig)
{
case SIGINT:
case SIGTERM:
/* SIGNAL-SPECIFIC ACTIONS GO HERE...       */
break;
default:
continue;
}
break;
}

/* Unset the signal mask...                                          */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling sigprocmask()"));
sigprocmask(SIG_UNBLOCK, &signals, NULL);

/* Terminate server: cause the main thread listen loop to return    */
```

```
/*  and go to cleanup. Obviously, if we're not listening yet, this  */
/*  will fail...                                                     */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_mgmt_stop_server_listening()"));
rpc_mgmt_stop_server_listening(NULL, &status);
if (status != error_status_ok)
{
print_server_error("rpc_mgmt_stop_server_listening()", status);
exit(1);
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting signal_handler()"));

}



/******
 *
 * sample_mgmt_auth -- Management authorization callback function.
 *
 *    This is the routine that is implicitly called to test authorization
 *     whenever someone tries to use the mgmt interface to tinker with us
 *     or our ACLs.
 *
 *
 *
 *    The callback is set up by a call to rpc_mgmt_set_authorization() in
 *    server_acl_mgr_setup().
 *
 ******/

boolean32 sample_mgmt_auth(
rpc_binding_handle_t client_binding, /* Client's binding, whoever he is.   */
unsigned32 requested_mgmt_operation, /* What client is attempting to do.   */
unsigned32 *status)
{
boolean32 authorized = 0;
sec_acl_permset_t perm_required;
unsigned_char_t *uuid_string;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering sample_mgmt_auth()"));

*status = error_status_ok;

/* Discover what permission is required in order to do what the     */
/*  client is trying to do...                                        */
switch (requested_mgmt_operation)
{
case rpc_c_mgmt_inq_if_ids:
perm_required = mgmt_perm_inq_if;
break;
case rpc_c_mgmt_inq_princ_name:
perm_required = mgmt_perm_inq_pname;
break;
case rpc_c_mgmt_inq_stats:
perm_required = mgmt_perm_inq_stats;
break;
case rpc_c_mgmt_is_server_listen:
       perm_required = mgmt_perm_ping;
break;
case rpc_c_mgmt_stop_server_listen:
perm_required = mgmt_perm_kill;
break;
default:
```

```
/* This should never happen, but just in case...            */
return(0);
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(&mgmt_acl_uuid, &uuid_string, status);
if (*status != uuid_s_ok)
{
print_server_error("uuid_to_string()", *status);
return(0);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of mgmt_acl_uuid == %s", uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, status);
if (*status != rpc_s_ok)
{
print_server_error("rpc_string_free()", *status);
return(0);
}

/* Okay, now check whether the client is authorized or not...       */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_acl_is_client_authorized()"));
dce_acl_is_client_authorized(
client_binding,         /* Client's binding handle.       */
&mgmt_acl_mgr_uuid,      /* ACL manager type UUID.          */
&mgmt_acl_uuid,          /* The ACL UUID.                   */
NULL,                   /*        Pointer to owner's UUID. */
NULL,                   /* Pointer to owner's group's UUID. */
perm_required,          /* The desired privileges.         */
&authorized,            /* Will be TRUE or FALSE on return. */
status);
if (*status != error_status_ok)
{
print_server_error("dce_acl_is_client_authorized()", *status);
return(0);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting sample_mgmt_auth()"));

if (authorized)
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"AUTHORIZED!"));
else
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"NOT AUTHORIZED!"));

/* Return the result to the caller...                               */
return(authorized);
}


/******
 *
 * sample_resolve_by_name -- take the name of an object, and return the
 *                           UUID of the object's ACL.
 *
 * The address of this function is passed (via the call to
 * server_acl_mgr_setup()) to the dce_acl_register_object_type() call. So
 * it gets implicitly called anytime someone tries to retrieve the ACL of
 * an object managed by the ACL manager we've set up.
 *
 * Basically, the most a server needs is one resolve-by-name routine and
 * one resolve-by-UUID routine; the former gets you the desired object's
```

```
 * UUID; and the latter then will get you the object data itself (the way
 * this works can be seen in the body of this routine below). In most
 * cases, these routines will share the same name and UUID databases; if
 * they don't, the resolver_arg can be used to point to the correct other
 * database. Typically, the only difference between the managers is that
 * they use different print strings.
 *
 * For the official statement of the signature of a dce_acl_resolve_func_t,
 * see the dce_acl_resolve_by_uuid() manpage; that routine has the same
 * type.
 *
 * NOTE that all this routine really has to do is look up the object
 *      UUID, get the ACL UUID from the object header, then extract the
 *      ACL and check its manager type with the manager_type passed, and,
 *      if the manager types match, return the ACL UUID; otherwise, return
 *      an error. Everything else is superfluous, though (perhaps) inter-
 *      esting.
 *
 *******/
dce_acl_resolve_func_t sample_resolve_by_name(
handle_t h,                     /* Client binding handle passed into the   */
/*  server stub. sec_acl_bind() is used to  */
/*  create this handle.                     */
sec_acl_component_name_t name,  /* The object whose ACL's UUID we want.    */
sec_acl_type_t sec_acl_type,    /* The type of ACL whose UUID we want.     */
uuid_t *manager_type,           /* The object's manager type.              */
/* NOTE that this parameter isn't used be-  */
/*  low.                                    */
boolean32 writing, /* "This parameter is ignored in OSF's im-  */
/*  plementation" (from the manpage for     */
/*  dce_acl_resolve_by_uuid()).             */
void *resolver_arg,             /* This is the app-defined argument passed  */
/*  to dce_acl_register_object_type(); it   */
/*  should be a handle for a backing store  */
/*  indexed by UUID. Note that it isn't     */
/*  used here though.                       */
uuid_t *acl_uuid,               /* To return ACL's UUID in.                */
error_status_t *st              /* To return status in.                    */
)
{
uuid_t u, *up; /* To hold the retrieved object UUID, and to         */
       /*  take a pointer to it.                            */
unsigned_char_t *uuid_string;
sec_acl_t retrieved_acl;
uuid_t owner_uuid, group_uuid;

/* The definition of the following is in the sample.idl file.       */
/*                                                                  */
/* See the "Examples" section in the dce_db_open() manpage,         */
/*  where the skeleton IDL interface for a server's backing         */
/*  store is given. The data type definition (which is what         */
/*  sample_data_t is) is there prescribed as consisting of a        */
/*  dce_db_header_t, plus whatever server-specific data is          */
/*  quired, all in a single structure.                              */
/*                                                                  */
/*  Essentially it's a dce_db_header_t structure (with an           */
/*  application-defined message string tacked on); this is          */
/*  the object header data structure that is returned, e.g.,        */
/*  by dce_db_header_fetch(); in other words, this is the           */
/*  thingie that actually contains the data "in" an object          */
/*  held in an object store. At least that's what I think it        */
/*  is...                                                           */
sample_data_t object_data;


DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering sample_resolve_by_name()"));
```

```
*st = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Object name == %s", name));

/* Check for non-existence of object name...                          */
if (!name || !*name)
{
dce_svc_printf(CANNOT_RESOLVE_NAME_MSG);
return;
}

/* Get the object's UUID, which will be the key that we will use to */
/*  fetch this particular object's data in the call following this  */
/*  one...                                                           */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_fetch_by_name()"));
dce_db_fetch_by_name(db_name, (char *)name, /* (void *) */ &u, st);
if (*st != error_status_ok)
{
print_server_error("dce_db_fetch_by_name()", *st);
return;
}

up = &u;     /* ...take the pointer to the key.                    */

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(up, &uuid_string, st);
if (*st != uuid_s_ok)
{
print_server_error("uuid_to_string()", *st);
return(0);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of retrieved key UUID == %s",
uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, st);
if (*st != rpc_s_ok)
{
print_server_error("rpc_string_free()", *st);
return(0);
}

/* Using the UUID "key" that we just retrieved, get the object_data */
/*  for the desired object (note that the data that one retrieves   */
/*  with this routine can be anything; it depends on what we are    */
/*  using the backing store for)...                                 */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_fetch_by_uuid()"));
dce_db_fetch_by_uuid(db_object, up, /* (void *) */ &object_data,
st);
if (*st != error_status_ok)
{
print_server_error("dce_db_fetch_by_uuid()", *st);
return;
}

/* Now, depending on the kind of ACL we're hunting for (i.e. ob-    */
/*  ject, container, etc.), extract its UUID from the object's      */
/*  header structure...                                             */
switch (sec_acl_type)
{
case 1:
```

```
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Case == 1"));
*acl_uuid = object_data.s_hdr.tagged_union.h.def_object_acl;
break;
case 2:
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Case == 2"));
*acl_uuid = object_data.s_hdr.tagged_union.h.def_container_acl;
break;
default:
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Case == default"));
*acl_uuid = object_data.s_hdr.tagged_union.h.acl_uuid;
}

/* Find out some other interesting stuff...                        */

owner_uuid = object_data.s_hdr.tagged_union.h.owner_id;
group_uuid = object_data.s_hdr.tagged_union.h.group_id;

uuid_to_string(&owner_uuid, &uuid_string, st);
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Owner UUID == %s", uuid_string));
rpc_string_free(&uuid_string, st);

uuid_to_string(&group_uuid, &uuid_string, st);
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"Group UUID == %s", uuid_string));
rpc_string_free(&uuid_string, st);

/* Here it might be interesting to try retrieving the ACL itself,   */
/*  and e.g seeing what its manager type is...                      */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling dce_db_fetch_by_uuid()"));
dce_db_fetch_by_uuid(db_acl,
acl_uuid,
&retrieved_acl,
st);
if (*st != error_status_ok)
{
print_server_error("dce_db_fetch_by_uuid()", *st);
return(0);
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(&(retrieved_acl.sec_acl_manager_type),
&uuid_string, st);
if (*st != uuid_s_ok)
{
print_server_error("uuid_to_string()", *st);
return(0);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"sec_acl_manager_type == %s", uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, st);
if (*st != rpc_s_ok)
{
print_server_error("rpc_string_free()", *st);
}

/* We are handling two ACL managers through this function, so we    */
/*  have to make sure that we've extracted from the single ACL      */
/*  database the correct ACL: i.e., one whose manager type UUID is  */
/*  identical to the manager_type parameter we were passed: this is */
```

```
/*  the manager whose ACL the runtime is trying to bind to. The     */
/*  point is that the ACL library is going to call all its regis-   */
/*  tered resolvers successively with the SAME ACL UUID, until it   */
/*  finds one that works. If we just return the ACL UUID without    */
/*  checking whether the right manager_type is being asked for,     */
/*  we'll only cause an error in the ACL library when it discovers  */
/*  that the types don't match up. This will prevent acl_edit from  */
/*  working. So do the checking here...                             */
if ((manager_type != NULL) && (!uuid_equal(manager_type,
&(retrieved_acl.sec_acl_manager_type),
st)))
{
/* Return a bad status...                                   */
*st = acl_s_bad_manager_type;
/* And no ACL UUID...                                       */
acl_uuid = NULL;
return(0);
}


DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling uuid_to_string()"));
uuid_to_string(acl_uuid, &uuid_string, st);
if (*st != uuid_s_ok)
{
print_server_error("uuid_to_string()", *st);
return(0);
}
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"sec_acl_type == %d", (int)sec_acl_type));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug8,
"String form of retrieved ACL UUID == %s",
uuid_string));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_string_free()"));
rpc_string_free(&uuid_string, st);
if (*st != rpc_s_ok)
{
print_server_error("rpc_string_free()", *st);
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting sample_resolve_by_name()"));

}




/******
 *
 * sample_bind_export -- Register the interface specification
 *                       and endpoints for the remote binding inter-
 *                       face.
 *
 *    Called from main().
 *
 ******/

void sample_bind_export(
rpc_binding_vector_t *binding_vector,
uuid_vector_t *uuid_vec,
unsigned32 *status)
{

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Entering sample_bind_export()"));
```

```
*status = error_status_ok;

/* Register sample_bind interface...                                */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_server_register_if()"));
rpc_server_register_if(sample_bind_v1_0_s_ifspec,
NULL,
(rpc_mgr_epv_t) &sample_bind_epv,
status);
if (*status != error_status_ok)
{
print_server_error("rpc_server_register_if()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug4,
"Calling rpc_ep_register()"));
rpc_ep_register(sample_bind_v1_0_s_ifspec,
binding_vector,
uuid_vec,
(unsigned_char_p_t) "sample_bind interface",
status);
if (*status != error_status_ok)
{
print_server_error("rpc_ep_register()", *status);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_server, svc_c_debug7,
"Exiting sample_bind_export()"));

}


/*****
*
* create_server_uuid -- Create server instance UUID.
*
*
*  Called from main(), make_server_entry().
*
*****/

void
create_server_uuid(
uuid_t *server_uuid,
uuid_vector_t *server_uuid_v
)
{
unsigned32 status;
dce_error_string_t error_string;
int print_status;

/*******************************************************************/
/* Create a UUID to identify this server instance; this will go in- */
/*  to the namespace and endpoint map so that clients of such gen-  */
/*  eric interfaces as rdacl can find this server's endpoints.      */
/*  Without such a UUID, these clients can't distinguish among      */
/*  servers on the same host that also export the generic inter-    */
/*  faces. This could be a well-known UUID, but here we will        */
/*  generate one on the fly. Clients binding to us by name will get */
/*  this UUID without having to know what it is.                    */
/*                                                                  */
/*******************************************************************/

fprintf(stdout, "Entering create_server_uuid()...\n");
```

```
/* Create and save server instance UUID...                          */
fprintf(stdout, "Calling uuid_create()...\n");
uuid_create(server_uuid, &status);
if (status != error_status_ok)
{
dce_error_inq_text(status, error_string, &print_status);
fprintf(stdout, "uuid_create(): %s\n", error_string);
exit(1);
}

server_uuid_v->uuid[0] = server_uuid;
server_uuid_v->count = 1;

fprintf(stdout, "...Exiting create_server_uuid()\n");
}


/*****
*
* print_server_error-- Server version. Prints text associated with
*                      bad status code.
*
*
*****/

void
print_server_error(
char *caller,          /* Routine that received the error.              */
error_status_t status)  /* Status we want to print the message for.       */
{
dce_error_string_t error_string;
int print_status;

dce_error_inq_text(status, error_string, &print_status);
dce_svc_printf(SERVER_ERROR_MSG, caller, error_string);

}
```

Note that the server code contained in these files is nearly all generic. In the ACL manager, the only application specific elements are the type of data stored in the object database, declared in **sample.idl**, and the name and object UUID for the initial object created during ACL manager setup. The export objects operation uses application-specific names and object uuids. The signal catcher thread installs application-specific handling for asynchronous signals, although the actual signal handling code simply causes the listen loop to return and invoke the generic cleanup operations.

Object Bind Interface

```
/***************************************************************************/
/* [27.VI.94]                                                              */
/*                                                                         */
/* sample_bind.c -- The remote binding interface implementation            */
/*                          code.                                          */
/*                                                                         */
/*                                                                         */
/*                                                                         */
/* The code below is built and linked into the server object; meanwhile    */
/* the sample_bind.idl file is processed and the output of that is         */
/* a set of client and server stubs for the implementation. The serverstub */
/* is generated with the -no_mepv option, which allows us to call ourim-   */
/* plementation by our own names, and explicitly initialize the entryoint */
```

```
/* vector structure with it (see the end of this file for how that hap-    */
/* pens). The client of course calls the routines by its standard name,    */
/* as generated in the client stub from sample_bind.idl.                    */
/*                                                                          */
/* In order to make the call remotely accessible, the server has to go      */
/* through the steps of registering the sample_bind interface (sep-         */
/* arately from all other interfaces, of course) with the name service,     */
/* and of registering its endpoints with the sample_bind interface (and     */
/* the "sample_bind_epv" vector) with the runtime. Then the client          */
/* has to import bindings to the sample_bind interface separately as        */
/* well. How all this is done can be seen in sample_client.c and            */
/* sample_server.c.                                                         */
/*                                                                          */
/*                                                                          */
/*                                                                          */
/*                            -77 cols-                                     */
/****************************************************************************/

#define DCE_DEBUG

#include <stdio.h>
#include <malloc.h>
#include <time.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/param.h>

#include <dce/dce.h>
#include <dce/dce_cf.h>
#include <dce/dce_error.h>
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/keymgmt.h>
#include <dce/uuid.h>
#include <dce/exc_handling.h>
#include <dce/dce_msg.h>
#include <dce/dbif.h>
#include <dce/aclif.h>
#include <dce/dceacl.h>
#include <dce/pgo.h>

#include <dce/dcesvcmsg.h>
#include <dce/svcremote.h>

/* Serviceability sams-generated header files...                            */
#include "dcesmpsvc.h"
#include "dcesmpmsg.h"
#include "dcesmpmac.h"

/* Following is our IDL-generated header...                                 */
#include "sample_bind.h"

#include "sample_server.h"

/* Declaration of the bind interface's routines' entry point vector. The    */
/*  actual addresses are filled in at the bottom of this file...            */
struct sample_bind_v1_0_epv_t sample_bind_epv;



/******
 *
 * name_to_object -- The remote bind operation implementation code:
 *                   receives a name, returns an object UUID.
 *
```

```
 *     Essentially what this routine is is a remote operation that doesn't
 *     actually "do" anything; it just returns a given object's UUID.
 *
 *
 ******/
void
name_to_object(handle_t binding_h, /* The binding that got us here.        */
unsigned_char_t *component,   /* The backing store's key.                 */
uuid_t *object_uuid,          /* For the UUID we will return.             */
uuid_t *mgr_type_uuid,        /* Type Manager UUID.                       */
unsigned32 *st                /* Status.                                  */
)

{

dce_error_string_t error_string;
int print_status;


*st = error_status_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_binder, svc_c_debug6,
"Entering name_to_object()..."));

if (!component || !*component)
{
dce_svc_printf(CANNOT_RESOLVE_NAME_MSG);
return;
}

/* dce_db_fetch_by_name() retrieves data from the string-indexed    */
/*  backing store identified by the handle parameter, which was     */
/*  obtained from dce_db_open(). It is a specialized retrieval      */
/*  routine for backing stores that are indexed by string, as sel-  */
/*  ected by the db_c_index_by_name bit in the flags parameter to   */
/*  dce_db_open() when the backing store was created.               */
/* Here it's the object_uuid that is to be returned...              */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_binder, svc_c_debug6,
"Calling dce_db_fetch_by_name()..."));
dce_db_fetch_by_name(
db_name,  /* Name-indexed database, globally-known handle.  */
(char *)component, /* Pointer to the key we're using, i.e.  */
/*  the name.                        */
object_uuid, /* What we're hoping to get, i.e. object UUID. */
st);
if (*st != error_status_ok)
{
dce_error_inq_text(*st, error_string, &print_status);
dce_svc_printf(BINDER_ERROR_MSG, "dce_db_fetch_by_name()",
error_string);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_binder, svc_c_debug6,
"Successfully exiting name_to_object()"));
}


/* The bind interface's routines' entry point vector. Here the actual ad-   */
/*  dresses are filled in...                                                 */
sample_bind_v1_0_epv_t sample_bind_epv = {
name_to_object
};


/****************************************************************************/
/* [27.VI.94]                                                               */
```

```
/*                                                                           */
/* sample_bind.idl --                                                        */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*                            -77 cols-                                      */
/*****************************************************************************/


[
uuid(006868ca-6064-1d49-9829-0000c0d4de56),
version(1.0)
]
interface sample_bind
{

void rs_bind_to_object(
[in] handle_t binding,
[in, string] char *component,
[out] uuid_t *object_uuid,
[out] uuid_t *mgr_type_uuid,
[out] error_status_t *status
);
}


/*****************************************************************************/
/* [27.VI.94]                                                                */
/*                                                                           */
/* sample_bind.acf --                                                        */
/*                                                                           */
/*                                                                           */
/*                                                                           */
/*                            -77 cols-                                      */
/*****************************************************************************/


interface sample_bind
{
rs_bind_to_object([comm_status,fault_status] status);
}
```

Manager and Client Illustrations

Most of the application-specific server code is contained in **sample_manager.c**. Since generic client tasks are so simple, the whole client is contained in **sample_client.c**.

```
/*****************************************************************************/
/* [27.VI.94]                                                                */
/*                                                                           */
/* sample_manager.c -- Implementation of "sample" interface.                 */
/*                                                                           */
/*   Routines in this file consist only of remote calls or routines that     */
/*    are internal to those calls.                                           */
/*                                                                           */
/*                                                                           */
/*                            -77 cols-                                      */
/*****************************************************************************/

#define DCE_DEBUG


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

```
#include <time.h>
#include <pthread.h>
#include <errno.h>
#include <unistd.h>
#include <signal.h>
#include <sys/param.h>

#include <dce/nbase.h>
#include <dce/dce.h>
#include <dce/dce_cf.h>
#include <dce/dce_error.h>
#include <dce/rpc.h>
#include <dce/sec_login.h>
#include <dce/keymgmt.h>
#include <dce/uuid.h>
#include <dce/exc_handling.h>
#include <dce/dce_msg.h>
#include <dce/dbif.h>
#include <dce/aclif.h>e <dce/dceacl.h>
#include <dce/pgo.h>

#include <dce/dcesvcmsg.h>
#include <dce/svcremote.h>

#include "dcesmpsvc.h"
#include "dcesmpmsg.h"
#include "dcesmpmac.h"

#include "sample.h"
#include "sample_db.h"
#include "sample_bind.h"
#include "sample_server.h"

void sample_call(rpc_binding_handle_t,
idl_long_int *,
error_status_t *);

void sample_get_text(rpc_binding_handle_t,
uuid_t,
idl_char *,
idl_long_int *,
error_status_t *);

void sample_put_text(rpc_binding_handle_t,
uuid_t,
idl_char *,
idl_long_int *,
error_status_t *);

void print_manager_error(char *,
error_status_t);



/******
 *
 * sample_call -- It don't do too much right now...
 *
 *
 *
 *
 ******/
/****************************************************************************/
void
sample_call(
rpc_binding_handle_t binding,       /* Client binding.              */
idl_long_int *status,
```

```
error_status_t *remote_status)
{

extern uuid_t sample_acl_mgr_uuid, sample_acl_uuid;
boolean32 authorized = 0;
/******************************************************************************/
/* We have to explicitly initialize the remote status value;        */
/*  otherwise, if no error occurs in the transmission (which        */
/*  would cause the runtime to assign an error value to this        */
/*  variable), its value will be whatever it happened to be         */
/*  when the RPC was made by the client...                          */
*remote_status = rpc_s_ok;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Entering sample_call()..."));

/* Check whether client is authorized or not...                     */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Calling dce_acl_is_client_authorized()..."));
dce_acl_is_client_authorized(
binding,          /* Client's binding handle.               */
&sample_acl_mgr_uuid,    /* ACL manager type UUID.          */
&sample_acl_uuid,        /* The ACL UUID.                   */
NULL,                   /*         Pointer to owner's UUID. */
NULL,                   /* Pointer to owner's group's UUID. */
sec_acl_perm_read,      /* The desired privileges.          */
&authorized,            /* Will be TRUE or FALSE on return. */
remote_status);

if (*remote_status != error_status_ok)
{
print_manager_error("dce_acl_is_client_authorized()",
*remote_status);
return;
}


if (authorized)
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Call authorized"));

/* HERE'S WHERE WE SHOULD ACTUALLY DO SOMETHING!            */

*status = error_status_ok;
}
else
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Call not authorized"));

/* Return no_permissions status to client...                */
*status = no_permissions;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Successfully exiting sample_call()"));

}



/******
 *
 * sample_get_text -- Extracts and returns an object's text information.
 *
 *
```

```
 *    Called from the client using its "object-bound" handle.
 *
 ******/
void sample_get_text(
rpc_binding_handle_t h,         /* Client binding handle passed into the   */
/*  server stub. sec_acl_bind() is used to  */
/*  create this handle.                     */
uuid_t object_uuid,             /* Desired object's UUID.
   */
idl_char text[TEXT_SIZE],       /* To return extracted text information.   */
idl_long_int *status,
error_status_t *remote_st       /* To return status.                       */
)
{

/* Our backing store data type. For a full explanation, see the     */
/*  body of sample_resolve_by_name(), in sample_server.c.           */
sample_data_t data;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Entering sample_get_text()..."));

*remote_st = rpc_s_ok;

/* Get the object's data header...                                  */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Calling dce_db_fetch_by_uuid()..."));
dce_db_fetch_by_uuid(db_object, &object_uuid, (void *)&data,
remote_st);
if (*remote_st != error_status_ok)
{
dce_svc_printf(OBJECT_NOT_FOUND_MSG);
  return;
}

/* Copy the text, if any, into the return parameter...              */
if (data.s_data.message)
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Text exists"));
strcpy(text, data.s_data.message);
}
else
{
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"No text"));
strcpy(text, "-No text-");
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Recovered text == %s", data.s_data.message));
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Message == %s", text));

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Successfully exiting sample_get_text()"));

}




/******
 *
 * sample_put_text -- Puts some text information "into" an object.
 *
 *
```

```
 *    Not called from anywhere.
 *
 ******/

void sample_put_text(
rpc_binding_handle_t h,         /* Client binding handle passed into the   */
/*  server stub. sec_acl_bind() is used to  */
/*  create this handle.                    */
uuid_t object_uuid,             /* Desired object's UUID.                  */
idl_char text[TEXT_SIZE], /* Text information to put.                */
idl_long_int *status,
error_status_t *remote_st       /* To return status.                       */
)
{

/* Our backing store data type. For a full explanation, see the     */
/*  body of sample_resolve_by_name(), in sample_server.c. See also  */
/*  the contents of sample_db.idl.                                  */
sample_data_t data;

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Entering sample_put_text()..."));

*remote_st = rpc_s_ok;

/* Get the object's data header...                                  */
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Calling dce_db_fetch_by_uuid()..."));
dce_db_fetch_by_uuid(db_object, &object_uuid, (void *)&data,
remote_st);
if (*remote_st != error_status_ok)
{
dce_svc_printf(OBJECT_NOT_FOUND_MSG);
return;
}

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug8,
"Storing text in object database"));

/* Now insert the text and stick it back in the backing store...    */
strcpy(data.s_data.message, text);
DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Calling dce_db_store_by_uuid()..."));
dce_db_store_by_uuid(db_object, &object_uuid, (void *)&data,
remote_st);

DCE_SVC_DEBUG((smp_svc_handle, smp_s_manager, svc_c_debug6,
"Successfully exiting sample_put_text()"));

}



/******
 *
 * print_manager_error-- Manager version. Prints text associated with bad
 *                       status code.
 *
 *
 ******/
void
print_manager_error(
char *caller,  /* String identifying the routine that received the error.   */
error_status_t status) /* the status we want to print the message for.     */
{
dce_error_string_t error_string;
int print_status;
```

```
dce_error_inq_text(status, error_string, &print_status);
dce_svc_printf(MANAGER_ERROR_MSG, caller, error_string);

}


/******************************************************************************/
/* [27.VI.94]                                                                 */
/*                                                                            */
/* sample_client.c -- Client of "sample" interface.                           */
/*                                                                            */
/*                                                                            */
/*                                                                            */
/*                               -77 cols-                                    */
/******************************************************************************/



#include <stdio.h>
#include <string.h>
#include <dce/dce_error.h>
#include <dce/nbase.h>
#include <dce/rpc.h>
#include <dce/dce_msg.h>
#include <dce/dbif.h>
#include <dce/dce.h>
#include <dce/dce_cf.h>
#include <dce/dcesvcmsg.h>
#include <dce/svcremote.h>
#include <dce/sec_login.h>
#include <dce/pgo.h>
#include <dce/secidmap.h>

#include "sample.h"
#include "sample_bind.h"


/* Data structure for holding object entry names...                           */
typedef struct {
unsigned32 count;
unsigned_char_t *name[1];
} objectname_vector_t;


/* ANSI-C style prototypes for functions private to this module...            */

int do_client_command_line(int,
char **,
unsigned32 *,
objectname_vector_t *);

boolean32 is_valid_principal(unsigned_char_t *_1,
unsigned_char_t *_2,
unsigned32 *_3);

void bind_to_object(unsigned_char_t *,
rpc_if_handle_t,
uuid_t *,
handle_t *,
uuid_t *,
uuid_t *,
unsigned_char_t **,
unsigned_char_t **,
error_status_t *);

void print_error(char *_1,
```

```
error_status_t _2);

extern unsigned_char_t *malloc();

#define SGROUP "sample_servers"


/******
*
*
* main --
*
*
******/

int
main(
int argc,
char *argv[]
)
{
rpc_ns_handle_t import_context;  /* Context for importing bindings. */
rpc_binding_handle_t binding_h;  /* Our binding handle.             */
error_status_t status;           /* For DCE library error returns.  */
error_status_t rpc_remote_status; /* For remote error returns.      */
idl_long_int rpc_status;    /* Application-returned status from re- */
/*  mote calls.                 */
unsigned_char_t *string_binding; /* For string binding conversions. */
unsigned_char_t *server_princ_name;
handle_t object_handle;     /* For binding we get through junction. */
uuid_t object_uuid, mgr_uuid;   /* Various UUIDs.                   */
unsigned_char_t *u_string;      /* For converting UUIDs to strings. */
objectname_vector_t objectname_vector; /* For entry names read from */
/*  command line.               */
unsigned32 kill_server = FALSE; /* TRUE => invoker requested "kill */
/*  server" option.             */
idl_char server_message[TEXT_SIZE]; /* For message string returned  */
/*  from the server to us.      */
unsigned_char_t *entry_name = NULL; /* Object entry name from       */
/*  bind_to_object().           */
unsigned_char_t *object_name = NULL; /* Residual object name from   */
/*  bind_to_object().           */


/* Process the command line...                                      */
do_client_command_line(argc, argv, &kill_server,
&objectname_vector);

/* Start importing servers. Note that the contents of the environ-  */
/*  ment variable RPC_DEFAULT_ENTRY are used to determine the entry */
/*  to import from...                                               */
fprintf(stdout,
"sample_client: Calling rpc_ns_binding_import_begin()...\n");
rpc_ns_binding_import_begin(
rpc_c_ns_syntax_default,
NULL,                        /* Use the RPC_DEFAULT_ENTRY. */
sample_v1_0_c_ifspec,
NULL,
&import_context,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_ns_binding_import_begin()", status);
exit(1);
}

/* Import the first server (we could iterate here, but we'll just   */
```

```
/*  take the first one)...                                            */
fprintf(stdout,
"sample_client: Calling rpc_ns_binding_import_next()...\n");
rpc_ns_binding_import_next(import_context, &binding_h,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_ns_binding_import_next()", status);
exit(1);
}

/* Free the import context...                                         */
fprintf(stdout,
"sample_client: Calling rpc_ns_binding_import_done()...\n");
rpc_ns_binding_import_done(&import_context, &status);
if (status != rpc_s_ok)
{
print_error("rpc_ns_binding_import_done()", status);
exit(1);
}

/* Resolve the partial binding...                                     */
fprintf(stdout,
"sample_client: Calling rpc_ep_resolve_binding()...\n");
rpc_ep_resolve_binding(binding_h,
sample_v1_0_c_ifspec,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_ep_resolve_binding()", status);
exit(1);
}

/* Convert the binding to a readable string...                        */
fprintf(stdout,
"sample_client: Calling rpc_binding_to_string_binding()...\n");
rpc_binding_to_string_binding(binding_h, &string_binding,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_binding_to_string_binding()", status);
exit(1);
}

/* Print it...                                                        */
fprintf(stdout,
"sample_client: Imported resolved binding == %s\n",
string_binding);

/* Free the string binding space...                                   */
fprintf(stdout, "sample_client: Calling rpc_string_free()...\n");
rpc_string_free(&string_binding, &status);
if (status != rpc_s_ok)
{
print_error("rpc_string_free()", status);
exit(1);
}

/* Find out what the server's principal name is...                    */
fprintf(stdout,
"sample_client: Calling rpc_mgmt_inq_server_princ_name()...\n");
rpc_mgmt_inq_server_princ_name(binding_h,
rpc_c_authn_dce_secret,
&server_princ_name,
&status);
if (status != rpc_s_ok)
{
```

```
print_error("rpc_mgmt_inq_server_princ_name()", status);
}
fprintf(stdout, "sample_client: Principal name returned == %s\n",
server_princ_name);

/* And now find out if it's a valid member of our sample_servers    */
/*  group...                                                         */
fprintf(stdout, "sample_client: Calling
is_valid_principal()...\n");
if (is_valid_principal(server_princ_name, (unsigned_char_t *)SGROUP,
&status))
{
fprintf(stdout,
"sample_client: Calling rpc_binding_set_auth_info()...\n");
rpc_binding_set_auth_info(binding_h,
server_princ_name,
rpc_c_protect_level_pkt_integ,
rpc_c_authn_dce_secret,
NULL,
rpc_c_authz_dce,
&status);

}
if (status != rpc_s_ok)
{
print_error("rpc_binding_set_auth_info()", status);
exit(1);
}

/*******************************************************************/
/* Everything's okay, so do some remote stuff. There are currently  */
/*  two possibilities: Either stop the server via the remote mgmt   */
/*  interface, or actually bind to an object and call a couple of   */
/*  remote operations.                                              */

/* First alternative: Kill the server via the mgmt interface...     */
if (kill_server)
{
fprintf(stdout,
"sample_client: Kill server option enacted...\n");
fprintf(stdout,
"sample_client: Calling rpc_ep_resolve_binding()...\n");
rpc_ep_resolve_binding(binding_h,
sample_v1_0_c_ifspec,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_ep_resolve_binding()", status);
exit(1);
}

fprintf(stdout,
"sample_client: Calling rpc_mgmt_stop_server_listening()...\n");
rpc_mgmt_stop_server_listening(binding_h, &status);
if (status != rpc_s_ok)
{
print_error("rpc_mgmt_stop_server_listening()", status);
exit(1);
}

fprintf(stdout, "sample_client: Server successfully
killed.\n\n");

}
/* Second alternative: Do remote operations...                      */
else
{
```

```
/* This is a local call...                                 */
fprintf(stdout,
"sample_client: Remote call option enacted...\n");
fprintf(stdout,
"sample_client: Preparing to bind to object %s\n",
(char *)objectname_vector.name[0]);
fprintf(stdout,
"sample_client: Calling bind_to_object()...\n");
bind_to_object(
(unsigned_char_t *)objectname_vector.name[0],
/* ...Name of object to bind to.    */
NULL,             /* Interface spec "hint".        */
NULL,             /* Object UUID "hint".           */
&object_handle,   /* Here's where binding will be. */
&object_uuid,  /* Here's where object UUID will be. */
&mgr_uuid,     /* Type manager UUID will be here.   */
&entry_name,   /* Full entry name returned here.    */
&object_name,  /* "Unresolved", i.e. object name.   */
&status);
if (status != error_status_ok)
{
fprintf(stdout, "Can't bind to object %s\n",
objectname_vector.name[0]);
return;
}

/* Display the binding, just for fun...                    */
fprintf(stdout, "View object %s:\n",
(char *)objectname_vector.name[0]);
fprintf(stdout, "     Via junction: %s\n     Object name:%s\n",
entry_name, object_name);

/* Convert to string form...                               */
fprintf(stdout,
"sample_client: Calling rpc_binding_to_string_binding()...\n");
rpc_binding_to_string_binding(object_handle,
&string_binding,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_binding_to_string_binding()", status);
exit(1);
}

/* Show it...                                              */
fprintf(stdout, "     Binding: %s\n", string_binding);

/* Now convert the type manager UUID to a string...        */
fprintf(stdout, "sample_client: Calling uuid_to_string()...\n");
uuid_to_string(&mgr_uuid, &u_string, &status);
if (status != uuid_s_ok)
{
print_error("uuid_from_string()", status);
exit(1);
}

/* Show it...                                              */
fprintf(stdout, "     Manager Type UUID: %s\n", u_string);

/* Convert the object UUID to string form...               */
fprintf(stdout, "sample_client: Calling uuid_to_string()...\n");
uuid_to_string(&object_uuid, &u_string, &status);
if (status != uuid_s_ok)
{
print_error("uuid_to_string()", status);
exit(1);
}
```

```
/* And show it...                                          */
fprintf(stdout, "     Object UUID: %s\n", u_string);

/* Now free the space...                                   */
rpc_string_free(&string_binding, &status);
if (status != rpc_s_ok)
{
print_error("rpc_string_free()", status);
exit(1);
}
rpc_string_free(&u_string, &status);
if (status != rpc_s_ok)
{
print_error("rpc_string_free()", status);
exit(1);
}

/* Make call on returned handle to get object data...      */
fprintf(stdout,
"sample_client: Calling [remote] sample_get_text()...\n");
sample_get_text(object_handle,
object_uuid,
server_message,
&rpc_status,
&rpc_remote_status);
fprintf(stdout, "     Object Text: %s\n", server_message);

/* Call the sample_call() operation...                     */
/* First, get rid of the object UUID...                    */
rpc_binding_set_object(
binding_h,
NULL,
&status);
if (status != error_status_ok)
{
print_error("rpc_binding_set_object()", status);
return;
}

/* Then display the binding...                             */
/* Convert to string form...                               */
fprintf(stdout,
"sample_client: Calling rpc_binding_to_string_binding()...\n");
rpc_binding_to_string_binding(binding_h, &string_binding,
&status);
if (status != rpc_s_ok)
{
print_error("rpc_binding_to_string_binding()", status);
exit(1);
}

/* Show it...                                              */
fprintf(stdout,
"sample_client: Binding about to be used == %s\n",
string_binding);

/* Free it...                                              */
rpc_string_free(&string_binding, &status);
if (status != rpc_s_ok)
{
print_error("rpc_string_free()", status);
exit(1);
}

fprintf(stdout,
"sample_client: Calling [remote] sample_call()...\n");
```

```
sample_call(binding_h, &rpc_status, &rpc_remote_status);
if (rpc_remote_status != error_status_ok)
{
print_error("sample_call()", rpc_remote_status);
exit(1);
}

fprintf(stdout,
"sample_client: Remote call option successfully
completed.\n\n");

}

fprintf(stdout, "sample_client: Calling rpc_string_free()...\n");
rpc_string_free(&server_princ_name, &status);
if (status != rpc_s_ok)
{
print_error("rpc_string_free()", status);
exit(1);
}

}




/******
 *
 * do_client_command_line -- Get and interpret arguments and options from
 *                   the command line, and do other setup related to the
 *                   command line's contents.
 *
 *    Called from main().
 *
 ******/

int do_client_command_line(
int argc,
char *argv[],
unsigned32 *kill_server,
objectname_vector_t *objectname_vector
)
{
unsigned32 status;

/* Check the command line...                                        */

/* The "Usage" message requires some explanation. There are two     */
/*  modes for running the client: you can either specify that       */
/*  the server be killed, or you can specify a single object to     */
/*  bind to. It is possible that the object name is not a namespace */
/*  entry (for example, I suppose, if it's the management object,   */
/*  whatever its name is). That is when things get interesting, be- */
/*  cause the application in effect implements a junction located   */
/*  at its server entry in the namespace, and clients are supposed  */
/*  to be able to bind to objects under the junction. Essentially   */
/*  this is done as follows. The client tries to bind to the over-  */
/*  qualified CDS entry formed by concatenating the specified ob-   */
/*  ject name to the server entry name; it ends up getting a part-  */
/*  ial binding to (what else?) the server; and it then makes a     */
/*  call to the remote bind operation with that binding, ostensibly */
/*  to get the object UUID of the object whose name was specified   */
/*  (to bind to) when the client was invoked. These objects are     */
/*  held in a backing store database. The remote call makes its way */
/*  by the familiar procedure to the server; the name_to_object()   */
/*  routine then simply looks up the desired object UUID by access- */
/*  ing the name-indexed backing store. When the remote call com-   */
```

```
/*  pletes, the client finds itself with a full binding and the de- */
/*  sired object UUID. It is pointed out below that remote calls    */
/*  are not routed by object UUID, so this is actually useless in   */
/*  regard to further operations, and can be discarded. Whether     */
/*  this is the way things ought to be I'm not sure.                */
/*                                                                  */
/* If the object binding option is specified, the following things  */
/*  should happen:                                                  */
/*                                                                  */
/*    1. The message "View object <object_name_specified>" is dis-  */
/*       played.                                                    */
/*                                                                  */
/*    2. The message "Via junction: <server_entry>                 */
/*                    Object name: <object_name>"                  */
/*       is displayed.                                              */
/*                                                                  */
/*    3. The message "Binding: <full_binding_to_object>" is dis-   */
/*       played.                                                    */
/*                                                                  */
/*    4. The message "Manager Type ID: <manager_UUID_string>" is   */
/*       displayed.                                                 */
/*                                                                  */
/*    5. The message "Object ID: <object_UUID_string>" is dis-     */
/*       played.                                                    */
/*                                                                  */
/*    6. The message "Object Text: <text_string>" is displayed.    */
/*                                                                  */
/*    7. --And this should be followed by some serviceability in-   */
/*       formational (soon to be debug) messages, from the server.  */
/*                                                                  */
/*  ...This is all assuming, of course, that no errors occur.       */

fprintf(stdout, "sample_client: Entering
do_client_command_line()...\n");

if (argc < 2)
{
fprintf(stdout, "\n Usage:\n");
fprintf(stdout, "       %s {<object_name> |
kill}\n\n", argv[0]);
fprintf(stdout, "Note that the client imports via
RPC_DEFAULT_ENTRY!\n\n");
exit(1);
}

if ((argc == 2) && !strcmp("kill", argv[1]))
{
fprintf(stdout, "sample_client: Kill server option selected.\n");
*kill_server = TRUE;
}
else
{
fprintf(stdout, "sample_client: Remote call option selected.\n");

/* Get the list of object entry names from the command      */
/*  line...                                                 */
objectname_vector->count = 1;
objectname_vector->name[0] =
(unsigned_char_p_t)malloc(strlen(argv[1]));
strcpy((char *)objectname_vector->name[0], argv[1]);
fprintf(stdout,
"sample_client: objectname_vector->name == %s\n",
objectname_vector->name[0]);
}

return 0;
}
```

```
/******
 *
 *
 * is_valid_principal -- Find out whether the specified principal is a
 *                       member of the group he's supposed to be.
 *
 *
 ******/

boolean32 is_valid_principal(
unsigned_char_t *princ_name,          /* Full name of principal to test.   */
unsigned_char_t *group,               /* Group we want principal to be in.  */
unsigned32 *status)
{

unsigned_char_t *local_name;       /* For principal's local name.   */
char *cell_name;                   /* Local cell name.              */
sec_rgy_handle_t rhandle;          /* Local registry binding.       */
boolean32 is_valid;             /* To hold result of registry call. */

fprintf(stdout, "sample_client: Entering is_valid_principal()...\n");
fprintf(stdout, "sample_client: Initial principal name == %s\n",
princ_name);
fprintf(stdout, "sample_client: Initial group name     == %s\n",
group);

/* Find out the local cell name...                                  */
fprintf(stdout, "sample_client: Calling dce_cf_get_cell_name()...\n");
dce_cf_get_cell_name(&cell_name, status);
if (*status != dce_cf_st_ok)
{
print_error("dce_cf_get_cell_name()", *status);
return 0;
}

/* Now bind to the local cell registry...                           */
fprintf(stdout, "sample_client: Calling sec_rgy_site_open()...\n");
sec_rgy_site_open(cell_name, &rhandle, status);
if (*status != error_status_ok)
{
free(cell_name);
print_error("sec_rgy_site_open()", *status);
return 0;
}

/* Free the cellname string space...                                */
free(cell_name);
if (*status != rpc_s_ok)
{
print_error("free()", *status);
return 0;
}

/* Get the specified principal's local (cell-relative) name...      */
local_name = malloc(strlen((char *)princ_name));

fprintf(stdout, "sample_client: Calling
sec_id_parse_name()...\n");
sec_id_parse_name(rhandle,    /* Handle to the registry server.     */
princ_name, /* Global (full) name of the principal. */
NULL, /* Principal's home cell name returned here.  */
NULL, /* Pointer to UUID of above returned here.    */
local_name, /* Principal local name returned here.  */
NULL, /* Pointer to UUID of above returned here.    */
status);
```

```
if (*status != error_status_ok)
{
free(local_name);
print_error("sec_id_parse_name()", *status);
return 0;
}
else
{
fprintf(stdout,
"sample_client: Full principal name  == %s\n",
princ_name);
fprintf(stdout,
"sample_client: Local principal name == %s\n",
local_name);
}

/* And finally, find out from the registry whether that principal   */
/*  is a valid member of the specified group...                      */
fprintf(stdout, "sample_client: Calling
sec_rgy_pgo_is_member()...\n");
is_valid = sec_rgy_pgo_is_member(rhandle,
sec_rgy_domain_group,
group,
local_name,
status);
if (*status != error_status_ok)
{
free(local_name);
print_error("sec_rgy_pgo_is_member()", *status);
return 0;
}

/* Free the principal name string area...                            */
free(local_name);
return(is_valid);

}


/******
 *
 * bind_to_object -- Local client call to get UUID from object name. *
 *   (The real sub-title of this saga seems to be "How to Implement
 *    a Junction".)
 *
 *    Called from main().
 *
 ******/

void bind_to_object(
unsigned_char_t *object_name,   /* The name of the object we're to bind to. */
rpc_if_handle_t if_hint,      /* Interface specification; NULL from main(). */
uuid_t *id_hint,         /* Presumably the object's UUID; NULL from main(). */
rpc_binding_handle_t *binding_h,     /* Binding will be returned here.     */
uuid_t *object_uuid,     /* Object's object UUID will be returned here.    */
uuid_t *mgr_type_uuid, /* Object's type manager UUID will be returned here. */
unsigned_char_t **entry_name, /* Full entry name (?) will be returned here. */
unsigned_char_t **residual,   /* Unresolved (?) name part returned here.   */
error_status_t *status)
{

unsigned_char_p_t resolved_name = NULL; /* To hold resolved part of */
/* object name.                     */
rpc_ns_handle_t import_context;    /* For NSI import operations.    */
unsigned_char_t *uuid_string;      /* Not used.                     */
unsigned_char_t *string_binding;   /* Not used.                     */
```

```
fprintf(stdout, "sample_client: Entering bind_to_object()...\n");

/* Attempt to resolve the entry (i.e., object) name we were        */
/*  given. The idea is that we are feeding this routine an over-    */
/*  qualified name, which it will be able to resolve only to a cer- */
/*  tain depth. What's left should be only a simple name, i.e. of   */
/*  the object we want to bind to...                                */
fprintf(stdout, "sample_client: Object name == %s\n",
object_name);
fprintf(stdout, "sample_client: Calling
rpc_ns_entry_inq_resolution()...\n");
rpc_ns_entry_inq_resolution(
rpc_c_ns_syntax_dce, /* Syntax for object_name.            */
object_name,        /* Entry name to be resolved.         */
&resolved_name,  /* Pointer to resolved name returned here. */
residual, /* Pointer to unresolved name part returned here. */
status);
if (*status != rpc_s_ok)
{
print_error("rpc_ns_entry_inq_resolution()", *status);
}

/* Object name only, try default search path... [original note]     */
/*  (Apparently the assumption is that if we gave an incomplete     */
/*  name, that must mean that we were passed only a simple object   */
/*  name, which means that we must try to reconstruct the path to   */
/*  the "junction"...                                               */
if (*status == rpc_s_incomplete_name)
{
fprintf(stdout,"sample_client: Object name only given, trying default search path...\n");

/* Make the object name the "residual"...                   */
*residual = (unsigned_char_t *)malloc(strlen((char *)object_name));
strcpy((char *)*residual, (char *)object_name);

/* Try importing from the RPC_DEFAULT_ENTRY, with interface */
/*  and object UUID specified, if any were given to us      */
/*  (which they weren't in the original call made from      */
/*  main())...                                              */
fprintf(stdout, "sample_client: Calling rpc_ns_binding_import_begin()...\n");
rpc_ns_binding_import_begin(
rpc_c_ns_syntax_default,
NULL,
if_hint,
id_hint,
&import_context,
status);

/* If that didn't succeed, we're at a loss...               */
if (*status != rpc_s_ok)
{
print_error("rpc_ns_binding_import_begin()", *status);
return;
}
fprintf(stdout, "sample_client: Found object.\n");
}

/* We either resolved the name completely, or we resolved every-    */
/*  thing but the simple object name part. But if the latter is     */
/*  the case, that's the same thing for us as having a full entry   */
/*  name to import from, since the whole point of this exercise is  */
/*  that the object part of the name isn't in the namespace in the  */
/*  the first place. So...                                          */

/* Import a binding...                                              */
else if (*status == rpc_s_partial_results || *status == error_status_ok)
{
```

```
fprintf(stdout, "sample_client: Binding to resolved name...\n");
fprintf(stdout, "sample_client: Calling rpc_ns_binding_import_begin()...\n");
rpc_ns_binding_import_begin(
rpc_c_ns_syntax_default,
resolved_name, /* This should be a namespace leaf.  */
if_hint,  /* Interface we were originally given.    */
id_hint,  /* Object UUID we were originally given.  */
&import_context,
status);

/* If this has failed, one possible reason is that we sup-  */
/*  plied an id_hint and this wasn't in the junction. We    */
/*  could try to import with the nil-UUID at this point and */
/*  then put the id_hint into the returned binding. That    */
/*  way, we would succeed if the correct UUID was in the    */
/*  endpoint map. For now, though, we'll just fail.         */
/*  [--original note]                                       */
/*                                                          */
/* Well, it's hard to see what sense the above makes when   */
/*  the only way this function is ever called is with a     */
/*  null interface...                                       */
if (*status != error_status_ok)
{
print_error("rpc_ns_binding_import_begin()", *status);
return;
}

}

fprintf(stdout, "sample_client: Calling
rpc_ns_binding_import_next()...\n");
rpc_ns_binding_import_next(
import_context,
(rpc_binding_handle_t*)binding_h,
status);
if (*status != error_status_ok)
{
print_error("rpc_ns_binding_import_next()", *status);
return;
}

fprintf(stdout, "sample_client: Calling
rpc_ns_binding_import_done()...\n");
rpc_ns_binding_import_done(&import_context, status);
if (*status != error_status_ok)
{
print_error("rpc_ns_binding_import_done()", *status);
return;
}

/* We succeeded in importing a (partial) binding. Presumably the    */
/*  name we successfully used is in resolved_name, but for some     */
/*  reason he wants to make the following call to get that name.    */
/*  Note that this is only for the purpose of returning the name    */
/*  to the caller; we have no further use for it here...            */
fprintf(stdout, "sample_client: Imported partial binding.\n");
fprintf(stdout, "sample_client: Calling rpc_ns_binding_inq_entry_name()...\n");
rpc_ns_binding_inq_entry_name(
(rpc_binding_handle_t)*binding_h,
rpc_c_ns_syntax_default,
entry_name,
status);
if (*status != error_status_ok)
{
print_error("rpc_ns_binding_inq_entry_name()", *status);
return;
}
```

```
/* Note that at this point, we can only assume that the server      */
/*  has put at least one object UUID in the endpoint map and        */
/*  the name space. If id_hint was null, we got one of the object   */
/*  UUIDs from the namespace at random. If id_hint was supplied,    */
/*  we either got that UUID or failed. If no UUIDs were exported,   */
/*  then the binding contains none, so when we make the call        */
/*  we are only guaranteed to get to some server that supports      */
/*  the sample_bind interface on the bound-to host. It may          */
/*  well be the wrong one, in which case we will now fail...        */

/* This is the "remote binding interface" call. What we are hoping  */
/*  to get from it is the object UUID of the object whose name is   */
/*  pretending (via a junction) to be in the namespace. These       */
/*  things, not being in the namespace, are held in a backing store */
/*  database maintained by the server...                            */
fprintf(stdout, "sample_client: Calling [remote]
rs_bind_to_object()...\n");
rs_bind_to_object(
*binding_h,      /* The partial binding we just got.        */
*residual,    /* The backing store "key", i.e. object name. */
object_uuid,     /* To return the object UUID.              */
mgr_type_uuid,   /* To return the type manager UUID.        */
status);
if (*status != error_status_ok)
{
print_error("rs_bind_to_object()", *status);
return;
}

/* The binding handle is now fully bound. Our request for the ob-   */
/*  ject UUID was really only a pretext for doing the namespace     */
/*  lookup and getting the binding handle completed with a server   */
/*  endpoint. The object UUID is not used for routing within the    */
/*  server. So we can now clear out any UUID set by id_hint and re- */
/*  place it with any type manager UUID returned from the server... */
fprintf(stdout, "sample_client: Fully bound to object.\n");
/* fprintf(stdout, "sample_client: Calling
rpc_binding_set_object()...\n");
rpc_binding_set_object(
(rpc_binding_handle_t)*binding_h,
mgr_type_uuid,
status);
if (*status != error_status_ok)
{
print_error("rpc_binding_set_object()", *status);
return;
}
*/
fprintf(stdout, "sample_client: Calling rpc_string_free()...\n");
rpc_string_free(&resolved_name, status);
if (*status != error_status_ok)
{
print_error("rpc_string_free()", *status);
return;
}

}



/******
 *
 * print_error-- Client version. Prints text associated with bad status code.
 *
 *
 *
```

```
******/

void
print_error(char *caller, /* Routine that received the error.            */
error_status_t status)    /* The status we want to print the message for.  */
{
dce_error_string_t error_string;
int print_status;

dce_error_inq_text(status, error_string, &print_status);
fprintf(stderr," Client: %s: %s\n", caller, error_string);
}
```

Message (sams) File

The sample application's **sams** file, which contains definitions for various messages output by the serviceability interface routines, is as follows:

```
###############################################################################
# [28.VI.94]                                                                  #
#                                                                             #
# smp.sams -- sams input file for generic sample program.                     #
#                                                                             #
#                                                                             #
#                                                                             #
#                                                                             #
#                                -77 cols-                                    #
###############################################################################

# Part I
component       smp
table           smp__table
technology      dce

###############################################################################
# Part II
serviceability  table smp_svc_table handle smp_svc_handle
start
    sub-component smp_s_server   "server"       smp_i_svc_server
    sub-component smp_s_manager  "manager"      smp_i_svc_manager
    sub-component smp_s_binder   "binder"       smp_i_svc_binder
end

###############################################################################
# Part III
# Note that defining the "sub-component" and "attributes" fields
#  will result in a convenience macro's being generated for the
#  message in question...

start
code sign_on
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Starting up"
explanation ""
action "None required."
end

start
code cleanup
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Cleaning up"
explanation "Starting server cleanup"
action "None required."
```

```
end

start
code server_exit
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Exiting"
explanation ""
action "None required."
end

start
code signal_catcher
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Spawning signal handler thread"
explanation ""
action "None required."
end

start
code no_signal_catcher
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Spawn signal handler failed"
explanation "RPC runtime error. pthread_create() failed."
action ""
end

start
code bad_entryname_count
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Bad entryname count"
explanation "Count of entrynames doesn't match count of object uuids"
action ""
end

start
code cannot_resolve_name
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Can't resolve name"
explanation "ACL manager resolver failed to resolve name"
action "The ACL databases may be corrupt and need to be regenerated."
end

start
code cannot_manage_keys
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Can't spawn key management thread."
explanation "RPC runtime error."
action ""
end

start
code no_acl_dbs
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "ACL databases not found, creating them from scratch"
explanation ""
action "None required."
end

start
code exporting_to
```

```
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Exporting to %s"
explanation "Exporting to CDS entry"
action "None required."
end

start
code unexporting_from
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Unexporting from %s"
explanation "Unexporting from CDS entry"
action "None required."
end

start
code importing_from
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Importing from %s"
explanation "Importing from CDS entry"
action "None required."
end

start
code auth_set_client
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Beginning client authentication setup"
explanation ""
action "None required."
end

start
code bindings_received
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Nr of %s bindings received == %d"
explanation "Server diagnostic message."
action "None required."
end

start
code full_binding
sub-component smp_s_server
attributes "svc_c_sev_notice"
text "Full %s binding in string form == %s"
explanation "Server diagnostic message."
action "None required."
end

start
code server_error
sub-component smp_s_server
attributes "svc_c_sev_fatal"
text "%s: %s"
explanation "general error message"
action "?"
end

start
code no_permissions
sub-component smp_s_manager
attributes "svc_c_sev_notice"
text "No permissions"
explanation "Client does not have permissions for operation"
```

```
action "None required."
end

start
code object_not_found
sub-component smp_s_manager
attributes "svc_c_sev_error"
text "Object not found"
explanation "object was not found in UUID-indexed database"
action "None required."
end

start
code manager_error
sub-component smp_s_manager
attributes "svc_c_sev_fatal"
text "%s: %s"
explanation "general error message"
action "?"
end

start
code binder_error
sub-component smp_s_binder
attributes "svc_c_sev_fatal"
text "%s: %s"
explanation "general error message"
action "?"
end

###############################################################################
# Part IIIa
# Messages for serviceability table
#
#  Note that there has to be one of these for each of
#   the sub-components declared in the second part of
#   the file (above)...


start !intable undocumented
code smp_i_svc_server
text "Sample server"
end

start !intable undocumented
code smp_i_svc_binder
text "Sample object binder"
end

start !intable undocumented
code smp_i_svc_manager
text "Sample manager"
end
```

Makefile

A generic Makefile suitable for building the sample code is as follows:

```
###############################################################################
# [22.VI.94]                                                                  #
#                                                                             #
# Makefile: A generic makefile suitable for building the sample               #
#           application.                                                      #
#                                                                             #
#                                -77 cols-                                    #
###############################################################################
```

```
# Library and include paths:
DCEROOT = /opt/dcelocal

# SAMS reference:
SAMS = sams

# IDL compiler:
IDLC = /bin/idl
# IDLC = /project/dce/build/nb_ux/tools/hp800/bin/idl

# Library directories:
LIBDIRS = -L$(DCEROOT)/usr/lib

# Libraries:
LIBS = -ldce
LIBALL = $(LIBDIRS) $(LIBS)

# Include directories (some compilers need -I. to pick up local header
files):
INC = -I. -I$(DCEROOT)/share/include -I$(DCEROOT)/usr/include

#
# NOTE: The following five lines are needed on HPUX...
#
# LDFLAGS =-Wl,-Bimmediate,-Bnonfatal
# IDLCC=-cc_cmd 'c89 -c'
# IDLCOPT=-cc_opt -D_HPUX_SOURCE
# CDEFS = -D_HPUX_SOURCE
# CC = c89
#

# CC flags:
CFLAGS = -g $(CDEFS) $(INC)

# IDL compiler flags. There are two versions of this line because, for the
#  sample_bind interface, we explicitly declare and initialize the entry-
#  point vector ourselves, so we specify that no epv structure be generated
#  by IDL for it; but for the sample interface itself, we want to use the
#  default epv structure, so when processing its .idl file we let IDL go
#  ahead and generate the structure. There actually is no particular reason
#  for explicitly declaring the vector for sample_bind (that I can see),
#  but it's instructive to see the two ways this can be done.
#
# The "-keep all" option is specified in order to avoid having IDL continu-
#  ally create and delete stub files. Doing it this way makes the build
#  much shorter...
NO_EPV_IFLAGS = -v -no_mepv $(IDLCC) $(IDLCOPT) $(INC) -keep all
IFLAGS = -v $(IDLCC) $(IDLCOPT) $(INC) -keep all
# -cc_cmd "$(CC) $(CFLAGS) -c" -keep all

# Interface name:
IF = sample

###############################################################################
# TARGETS:
# Executables...
CLIENT = $(IF)_client
SERVER = $(IF)_server

# Objects:
CLIENTO = $(IF)_client.o
SERVERO = $(IF)_server.o
CLIENTSO = $(IF)_cstub.o
SERVERSO = $(IF)_sstub.o
DBSO = $(IF)_db_cstub.o
MGRO = $(IF)_manager.o
```

```
SVCMSGO = dcesmpmsg.o
SVCSVCO = dcesmpsvc.o

# Remote bind interface:
BIND_REMOTE = sample_bind
BIND_REMOTEC = $(BIND_REMOTE).c
BIND_REMOTEO = $(BIND_REMOTE).o

# Sams generated:
SVCH = dcesmpmsg.h dcesmpsvc.h dcesmpmac.h
SVCMSGC = dcesmpmsg.c
SVCSVCC = dcesmpsvc.c
FROMSAMS = $(SVCH) $(SVCC)

# IDL generated:
HDR = $(IF).h
DBHDR = $(IF)_db.h
SHDR = $(SERVER).h
CLIENTSC = $(IF)_cstub.c
SERVERSC = $(IF)_sstub.c
DBSC = $(IF)_db_cstub.c
FROMIDL = $(HDR) $(CLIENTSC) $(SERVERSC)
FROMDBIDL = $(DBHDR) $(DBSC)
FROMBINDIDL = sample_bind.h sample_bind_sstub.c sample_bind_cstub.c

###############################################################################
# DEPENDENCIES:

all: $(CLIENT) $(SERVER)

###############################################################################
# Executables (.o dependencies):

$(SERVER): $(SERVERSO) $(BIND_REMOTEO) $(DBSO) $(SERVERO) $(MGRO) \
$(SVCMSGO) $(SVCSVCO) sample_bind.h sample_bind_sstub.c
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $(SERVERSO) $(BIND_REMOTEO) \
$(DBSO) $(SERVERO) $(MGRO) $(SVCMSGO) $(SVCSVCO) \
sample_bind_sstub.o $(LIBALL)

$(CLIENT): $(CLIENTSO) $(CLIENTO) $(SVCMSGO) $(SVCSVCO) sample_bind.h
\
sample_bind_cstub.c
$(CC) $(CFLAGS) $(LDFLAGS) -o $@ $(CLIENTSO) $(CLIENTO) \
sample_bind_cstub.o $(SVCMSGO) $(SVCSVCO) $(LIBALL)

###############################################################################
# Object files (.c and .h dependencies):

$(SERVERSO): $(SERVERSC) $(HDR) $(SHDR) sample_bind_sstub.c

$(CLIENTSO): $(CLIENTSC) $(HDR) sample_bind_cstub.c

$(DBSO): $(DBSC) $(DBHDR)

$(SERVERO): $(IF)_server.c $(HDR) $(SHDR) $(DBHDR) $(SVCH)

$(CLIENTO): $(IF)_client.c $(HDR) $(SVCH) $(FROMBINDIDL)

$(BIND_REMOTEO): sample_bind.c sample_bind.h $(SHDR) $(SVCH)

$(MGRO): $(IF)_manager.c $(HDR) $(SHDR) $(DBHDR) $(SVCH)

$(SVCMSGO): $(SVCMSGC)

$(SVCSVCO): $(SVCSVCC)

###############################################################################
```

```
# IDL generated files (.idl and .acf dependencies):

$(FROMIDL): $(IF).idl $(IF).acf
$(IDLC) $(IF).idl $(IFLAGS)

$(FROMDBIDL): $(IF)_db.idl $(IF)_db.acf
$(IDLC) $(IF)_db.idl $(IFLAGS)

$(FROMBINDIDL): sample_bind.idl sample_bind.acf
$(IDLC) sample_bind.idl $(NO_EPV_IFLAGS)

##############################################################################
# Sams generated files (.sams dependencies):

$(FROMSAMS): smp.sams
$(SAMS) smp.sams
##############################################################################
clean:
rm -f $(FROMIDL) $(SERVERSO) $(SERVERO) $(DBSO) $(MGRO) $(UTILO) \
$(CLIENTSO) $(CLIENTO) $(UTILO) $(FROMSAMS) \(null\).idl
\
dcesmp.cat dcesmp.msg dcesmpmsg.c dcesmpmsg.idx \
dcesmpmsg.man dcesmpmsg.sgm dcesmpsvc.c sample_bind.h \
sample_db.h dcesmpmsg.o dcesmpsvc.o sample_bind.o \
dcesmpmsg.sml sample_bind_cstub.o sample_bind_sstub.o .idl \
sample_bind_cstub.c sample_bind_sstub.c sample_db_cstub.c

rmtarget:
rm -f $(CLIENT) $(SERVER) core

clobber: clean rmtarget
```

Index

Special Characters

_cstub.c file 10
-no_cxxmgr IDL compiler option 168
_sstub.c file 10

A

abstract class 167
accept() 43
access algorithm
  as implemented in DCE ACL library 68
  standard DCE 28
access control (authorization)
  definition of 51
access control list (ACL) 16
  defined 67
  retrieving an object's 27
  use of in authorization 67
access criteria
  tests of 65
ACF attribute cxx_static 168
ACL 7
  binding routine 70
  database 71
  design guidelines 28
  entry types 69
  handles 27
  manager 16, 69, 70, 71, 72, 73, 74, 75, 76
  operations implemented by library 70
  standard DCE entry types 28
  types 28
  UUID 70
ACL-based authorization 53
acl_edit 28, 74
adding an interface 168
allocating
  new memory 139
  parameter memory 133
anonymous user UUID 67
any_other ACL entry type 68
arrays 145
asynchronous
  cancelability of threads 42, 46
at-least-once semantics 131
atfork() 40, 49
attribute configuration file 3, 6, 8, 10
attribute configuration language 4
authenticated RPC 16
authentication
  and authorization 25, 52
  available services 65
  client's decision to use or not 24
  default model for server authentication 59
  definition of 51
  definition of identity 56
  determining client's information 26
  effect of on RPC calls 52
  how often performed 52
authentication *(continued)*
  model of in DCE 25
  necessity of for server access to namespace 58
  not requested by client 51
  requested by client 51
  service registered by servers 52
  service specified in client's binding handle 52
  services 24
  setting up for 24
  steps involved in using 53
  three options for server 56
  three stages in 54
authorization 16, 65
  absence of service acceptable to server 66
  available services 52
  checking service acceptability to server 66
  decision made by server 52
  determining client's information 26
  how identity expressed 52
  how often performed 52
  list of available services 66
  making decision 28
  recommended scheme of 52
  service based on client's credentials 66
  service based on client's principal name 66
  service based on group and organization membership 66
  service specified per binding 52
  steps involved in using 53
  supported only with authentication 52
automatic binding 98

B

backing store
  API 13
  database items 75
  headers 74
  items (records) 74
  standard header 75
base class for DCE 167
basic data passing styles 131
benefit of C++ 165
bind() function 169
binding
  acquiring a handle 22
  annotating handle for security 23, 24, 57
  automatic method 10, 22, 26
  choices of 22
  compatibility 23
  discarding import context 26
  explicit method 10, 22
  exporting 7, 19
  failure 25
  freeing memory used by handle 29
  handle 10, 19
  implicit method 10, 22
  importing compatible 23

binding (continued)
management methods 22
methods 10, 22
partial 14, 15, 18, 19
registration of 14, 18
string 22
binding to objects 169
broadcast semantic attribute 132
buffer
incoming RPC call request 20, 29, 37

C

C++ language 165
benefits in using 165
CDS object entries 171
changing an interface 168
character sets 138
classes
DCE base class rpc_object_reference 167
defining in IDL 167
interface 167
manager 168
proxy 168
RPC hierarchy 167
client proxy class 168
clients
portability of 32
comm_status attribute 10
condition variables 40
and threads cancellation 44, 45
when they should be used 44
connect() 43
context handles 145
conversation keys 54, 55
credentials
acquisition 59
anonymous 67
client's 26
definition of 56
inheriting 56
null 67
revalidation 59
customized binding handle 100
cxx_static ACF attribute 168

D

daemon
DCE host 6
data protection
definition of 51
steps involved in using 53
supported levels of 52
data types
handle_t 100
rpc_binding_handle_t() 100
dce_acl_is_client_authorized() 66, 72, 74
dce_acl_register_object_type() 76
DCE Audit Service 51
dce_cf_get_cell_name() 63
dce/database.idl 75
dce_db_open() 75, 76
dce_db_std_header_init() 75
dce_db_store_by_name() 13, 14
dce_db_store_by_uuid() 13, 14
dce/dceacl.idl 76
dce_error_inq_text() 12
DCE host daemon 6
DCE login 58
dce_msg_define_msg_table() 12
DCE programming style
recommended 29
dce_server_disable_service() 18, 21
dce_server_enable_service() 18
dce_server_inq_attr() 11
dce_server_inq_server() 173
dce_server_inq_uuids() 11
dce_server_register() 17, 19, 174
dce_server_register_data_t structure 17
dce_server_sec_begin() 15, 22, 57, 174
dce_server_sec_done() 22
dce_server_unregister() 21
DCE_SVC_DEBUG() 13
dce_svc_debug_routing() 12
dce_svc_printf() 12, 19
dce_svc_register() 12
dce_svc_routing() 12
DCE threads
user-space implementation 36
dcecp 6, 7
rpcentry show 8
rpcgroup show 8
rpcprofile show 8
scripts 8
server create 11
server ping 8
server show -executing 8
server start 8, 11
server stop 20
dced 6, 173
configuration data for 11
endpoint mapper service 14
host data management 7
key table management 7
security validation 7
server control 7
dced_server_start() 11
dced_server_stop() 20
defining C++ classes 167
distributed applications
administrative demands of 32
distributed object 165
distributed objects and RPCs 169
dynamic objects 166

E

endpoint
dynamic 14, 18
map 6, 7
mapper service 18
mapping mechanism 15
registering 18

endpoint *(continued)*
RPC 14
stale data in map 21
well-known 18
ENDTRY macro 45
entry point vector
addresses in 16
manager 16
multiple 16
registering 17
entry types
in ACL, recommended checking order 68
EPV (entry point vector) 15
exception handlers 21
exception-handling macros and C++ 165
exceptions
and rpc_server_listen() 29
handling interface 21
signals and 21
terminating 21
exec() 49
execution semantics 5
explicit binding 98

F

filenames
representation of in DCE 103
FINALLY macro 45
foreign cells
user and group ACL entries for 68
foreign_group ACL entry type 68
foreign_user ACL entry type 68
fork() 40, 49
framework 165
free() 63
full pointer parameters 134

G

general cancelability of threads 42
global lock 39
and threads cancellation 44
group ACL 68
group membership
reason for checking 24
group_obj ACL entry type 68
ignored by DCE ACL library 69
group UUID
client's 27

H

handle
binding 5
context 5
customized binding 100
hierarchy of RPC classes 167

I

idempotent semantics 131
IDL 1
IDL 3 *(continued)*
compiler 3, 6, 10
file 9
operations 9
unions 150
IDL compiler option
-no_cxxmgr 168
IDL static keyword 168
implicit binding 98
incoming RPCs
specifying maximum concurrency of 37
initialization tasks, server 174
instance, server 7
interface 2
ensuring compatibility of between server and client 25
generic 18
handle 6
multiple definitions 9
specification in stubs 25
UUID 3, 8, 14, 18
version number 9, 10, 16
interface changes 168
interface class 167
interface definition language (IDL) 4
interface specification 6
is_valid_principal() 63
ISO_LATIN_1 character set 138
ISO_MULTI_LINGUAL character set 138
ISO_UCS character set 138

K

Kerberos shared secret key protocol 53
keys
local copies of user keys 56
maintenance of by server 56
managing server key 16, 58, 60
secret key 16
should not be kept in program memory 57
where server obtains its key 59
keytab file 56
how created 58
permissions needed to access 58
kinds of objects 165

L

local login 58
login context 16, 60
client 24
default 57
definition of 56
handle 23
inheriting 55, 56, 57
represented by sec_login_handle_t 57
longjmp() 46

M

malloc() 38, 63

manager
  entry point vectors 15, 16
  type 15
  type UUID 71
manager class 168
mappings
  IDL to C 135
marshalling pointers 139, 146
mask entries
  in ACL, defined 67
mask_obj ACL entry type 69
maybe semantic attribute 132
memory
  allocating new 139
memory management, parameter 133
message catalogs 12, 156
message generation 157
multithreadedness 3
  advantages and disadvantages of 36
  and blocking 36
  and multiprocessors 36
  and single processors 36
  and unsafe libraries 40
  complexity cost of 37
  in clients 36
  in servers 19, 29, 36
  questions to consider when adding to applications 37
mutexes 39
  and locking sequences 40
  and shared objects 40
  and threads cancellation 44
  when they should be used 44

N

named objects 166
names
  difference of from objects 103
namespace
  defined 103
  entry name in 23
  exporting binding information into 22
  importing binding handles from 22
  unexporting information from 21
naming objects 170
nonpointer type parameters 134
NSI routines 7

O

object 15
  ACL for 72
  and ACL managers 13
  as an abstraction 72
  defined 71
  difference of from names 103
  server objects 13
  UUIDs 13, 14, 23
object entries in CDS 171
object naming 170
object-oriented applications 165
object-oriented binding model 27
object reference 169
objects
  binding to rather than servers 169
  keeping track of clients 166
  kinds of 165
  reference counting 166
on-demand server 173
operations
  declaration of 9
  IDL 9
orphaned node 134
other_obj ACL entry type 68

P

parameter semantics 133
parameters
  full pointers 134
  in 5
  nonpointer types 134
  out 5
  reference pointers 134
permissions semantics 68
permissions set
  in ACL 67
persistent objects 166
persistent server 173
pipe mechanism 32
pipes 152
pointers
  embedded 140
  marshalling 139, 146
  reference 139
  RPC 138
  unique 139
polymorphism 167
primitive binding handle 100
primitive data types 135
principal identity
  role of in authentication 54
principal name
  client's 27
  how obtained by clients 63
  maintaining a client list of 63
  server 23, 58
principal UUID
  client's 27
privilege attribute certificate 27
privilege attributes
  defined 67
privileges
  standard DCE 28
protection level
  checking acceptability of by server 65
  client's specification altered by runtime 65
  minimum level aceptable to server 65
  server policy with respect to 65
protocol sequences
  and binding compatibility 23
  RPC 18
  RPC runtime and 23

proxy class 168
PTGT (privilege ticket granting ticket) 56
pthread_cancel() 41, 42, 44, 46
pthread_cleanup_pop() 45
pthread_cleanup_push() 45
pthread_cond_signal() 48
pthread_cond_timedwait() 43, 46
pthread_cond_wait() 40, 43, 46
pthread_create() 19, 35, 60
pthread_delay_np() 43
pthread_detach() 41
pthread_equal() 41
pthread_exit() 46
pthread_getspecific() 41
pthread_join() 41, 43, 44
pthread_keycreate() 41
pthread_lock_global_np() 39, 40
pthread_mutex_lock() 39, 44, 45, 46
pthread_mutex_unlock() 45
pthread_set_prio() 40
pthread_set_scheduler() 40
pthread_setasynccancel() 42, 43
pthread_setcancel() 42, 44
pthread_setprio() 35
pthread_setspecific() 41
pthread_signal_to_cancel_np() 42
pthread_testcancel() 43, 44

R

rdacl interface
  use of by DCE applications 27
read() 43
readv() 43
recv() 43
recvfrom() 43
recvmsg() 43
reentrancy 38
  non-reentrant library calls 39
  reentrant libraries 40
reference counting 166
reference pointers 134, 139
register_data structure 17
remote ACL (rdacl) interface
  use of by DCE applications 27
remote call
  server's return from 29
resolver routine
  sample application's name resolution routine 74
rgy_edit
  ktadd 58
RPC
  binding mechanism 13
  data types 100, 135
  incoming calls 20
  interface 5, 6, 9
  parameter data 131
  pointers 138
  server instances 7
  threads 49
rpc_binding_free() 29
rpc_binding_from_string_binding() 22
rpc_binding_inq_auth_caller() 26, 52
rpc_binding_inq_auth_client() 66
  used in authorization 65
rpc_binding_set_auth_info() 23, 24, 57, 58, 63
  parameters required by 57
rpc_binding_to_string_binding() 23
RPC class hierarchy 167
rpc_ep_resolve_binding() 63
rpc_ep_unregister() 21
rpc_mgmt_inq_if_ids() 73
rpc_mgmt_inq_princ_name() 24
rpc_mgmt_inq_server_princ_name() 58, 63
rpc_mgmt_set_authorization_fn() 20
rpc_mgmt_set_authorizaton_fn() 174
rpc_mgmt_stop_server_listening() 20, 29
rpc_network_inq_protseqs() 23
rpc_network_is_protseq_valid() 23
rpc_ns_binding_export() 19
rpc_ns_binding_import_begin() 22, 29
rpc_ns_binding_import_done() 22, 29
rpc_ns_binding_import_next() 22
rpc_ns_binding_unexport() 21
rpc_ns_mgmt_handle_set_exp_age() 26
rpc_object_reference class 167
rpc_object_set_type() 15, 165
rpc_server_listen() 20, 29, 37, 174
  and *max_calls_exec* parameter 20
  and max_calls_exec parameter 29
rpc_server_register_auth_info() 52, 59, 62
rpc_sm_allocate() 134
rpc_sm_client_free() 134
rpc_string_binding_parse() 23
rpcentry export 7
RPCs and distributed objects 169

S

sams utility 12, 157
sec_acl_bind() 27, 70
sec_acl_bind_to_addr() 70
sec_acl_calc_mask() 70
sec_acl_get_access() 70
sec_acl_get_error_info() 70
sec_acl_get_manager_types() 70
sec_acl_get_mgr_types_semantics() 70
sec_acl_get_printstring() 70
sec_acl_lookup() 70
sec_acl_mgr_is_authorized() 66
sec_acl_replace() 70
sec_acl_test_access() 70
sec_acl_test_access_on_behalf() 70
sec_id_parse_name() 63
sec_key_mgmt_get_key() 57, 60, 61
sec_key_mgmt_manage_key() 59, 60
sec_login_get_current_context() 24
sec_login_get_expiration() 61
sec_login_refresh_identity() 61
sec_login_set_context() 57, 60
sec_login_setup_identity() 57, 60
sec_login_validate_identity() 57, 60, 61
sec_rgy_pgo_is_member() 24, 63
sec_rgy_site_open() 63

secret key
  authentication 52, 54
security
  basic model for DCE applications 51
select() 43
semantics
  parameter 133
send() 43
sendmsg() 43
sendto() 43
server
  binding information recommendations 32
  configuration information 8
  design guidelines 32
  initialization of 11, 174
  local identity needed by 58
  management 173
  multiple servers on one host 15
  on-demand 173
  persistent 173
  portability of 32
  principal identity 16
  recommended design of interfaces 32
  registering 17
server instances
  distinguishing 18
server manager class 168
serviceability 3, 12, 157
severity level attribute 158
sigaction() 47
  should not be used for asynchronous signals 47
siglongjmp() 46
signals
  and rpc_server_listen() 29
  asynchronous 47
  blocking of 46
  handling of in a multithreaded environment 46
  masking of 47
  rules for handling in multithreaded programs 48
  semantics of 47
  synchronous 47
sigprocmask() 47
sigwait() 43, 47, 48
static functions 168
static IDL keyword 168
structures 149
stubs 2
  files 10

T

TGT (ticket granting ticket) 56
threads
  attributes object 35, 40
  avoiding priority inversions 40
  behavior of when blocked 38
  cancelability 40
  cancelability state 42
  canceling 42
  cancellation and condition variables 44, 46
  cancellation and exception-handling block 44, 45
threads *(continued)*
  cancellation and mutexes 35
  cancellation cleanup 45
  cancellation points 43
  default scheduling policy of 38, 40
  disabling cancellation state 44
  handles 41
  manager 20
  RPC 19, 29
  rules for cancellation 46
  scheduling policies 36
  side effects of cancellation 44
  specific global storage 41
  specifying the number of in server 37
  termination 42
  timeslice interruption of 43
  wrappers 38
  wrappers and pthread.h 40
  wrappers as cancellation points 43
threadsafe routines provided by DCE 38
threadsafeness 20
  criteria for routines 39
  of library routines 38
tickets 54
  Kerberos use of 55
  privilege ticket granting ticket (PTGT) 56
  ticket granting ticket (TGT) 56
TRY macro 45
type
  manager 15
  managers 28
  UUID 15

U

unauthenticated access 51
unexporting server namespace information 21
unions 149
unique pointers 139
unregistered applications 156
unregistering server address information 21
user ACL 68
user_obj ACL entry type 68
  ignored by DCE ACL library 69
uu_convert() 75
UUID 14
  generator 4
  interface 8, 14, 17
  object UUIDs 18, 19
  type 17
uuid_create() 13, 14
uuidgen 4, 8, 9, 14

V

virtual functions 167

W

wait() 40
well-known objects 166

write()   43
writev()   43