HEWLETT PACKARD

HP 9000
Series 700/800
Computers

NCS 1.5.1 to DCE RPC Transition Guide

NCS 1.5.1 to DCE RPC Transition Guide

Workstation Systems Division
Order No. B3193–90002
Manufacturing No. B3193–90002

First Printing: January 1993

NOTICE

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

HEWLETT-PACKARD 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. Hewlett-Packard shall not be liable for errors contained herein or for incidental or consequential damages in connection with the furnishing, performance or use of this material.

Hewlett-Packard assumes no responsibility for the use or reliability of its software on equipment that is not furnished by Hewlett-Packard.

RESTRICTED RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in sub-paragraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013.

Hewlett-Packard Company
3000 Hanover Street
Palo Alto, CA 94304 U.S.A.

Rights for non-DOD U.S. Government Departments and Agencies are as set forth in FAR 52.227-19(c)(1,2).

10 9 8 7 6 5 4 3 2 1

Preface

The OSF Distributed Computing Environment provides several compatibility features that enable Network Computing System applications to run in a DCE Remote Procedure Call (RPC) environment. NCS applications can also be migrated to DCE RPC to take advantage of services not available in an NCS environment. We recommend that you migrate to DCE RPC any NCS applications that you expect to continue using under future releases of DCE, and that you use DCE RPC for new development efforts.

The *NCS 1.5.1 to DCE RPC Transition Guide* is designed to help you migrate your NCS applications to DCE RPC. It describes the differences between NCS and DCE RPC and shows you how to change your NCS applications so that they can run as DCE RPC applications. In this manual we focus specifically on applications written for NCS 1.5.1. If your applications were written for NCS 2.0, you will need to make few, if any, changes, because NCS 2.0 is a subset of DCE RPC.

The manual is organized as follows:

Chapter 1	Contains a brief overview of DCE and the DCE Remote Procedure Call Application Programming Interface; Chapter 1 also includes an overview of this manual.
Chapter 2	Describes the major concepts in the DCE RPC API and compares the DCE RPC API with the NCS 1.5.1 API.
Chapter 3	Describes DCE IDL and compares it with NCS NIDL.

Chapter 4	Describes how to write DCE interface definitions and how to use the **nidl_to_idl** tool to convert NCS 1.5.1 interface definitions.
Chapter 5	Describes how to convert clients and servers and how to use the DCE Threads exception–returning package.
Chapter 6	Describes how to handle multiple managers in DCE.
Chapter 7	Contrasts NCS and DCE location services and summarizes the steps clients and servers must take to use DCE location services.
Appendix A	Summarizes the DCE RPC API.
Appendix B	Contains source code listings for the NCS 1.5.1 programs that are converted in Chapters 5 and 6.
Appendix C	Summarizes DCE IDL and ACF Attributes.

Related Manuals and Books

This manual is not intended to provide a complete guide to writing DCE RPC applications. For more information on DCE programming, refer to the following manuals:

- *Introduction to OSF DCE*
- *OSF DCE Application Development Guide*
- *OSF DCE Application Development Reference*

For information about using DCE threads, refer to the following manual:

- *Programmer's Notes on HP DCE Threads*

For information about the DCE RPC Daemon (**rpcd**), refer to the following manual:

- *OSF DCE Administration Guide*

To order Hewlett-Packard manuals, call 800–227–8164. Outside the USA, please contact your local sales office.

The following books, which are not available through Hewlett-Packard at this time, may also prove helpful:

- Shirley, John, *Guide to Writing DCE Applications*, O'Reilly and Associates, Inc., Sebastopol CA, June 1992, ISBN 1–56592–004–X
- Rosenberry, Ward, David Kenney, and Gerry Fisher, *Understanding DCE*, O'Reilly and Associates, Inc., Sebastopol CA, September 1992, ISBN 1–56592–005–8

Your local bookseller can order these books for you if they are not in stock.

Does This Manual Support Your Software?

This manual was released with the HP DCE Developers' Environment. If you are using a later version of HP DCE, this manual may not be applicable. Refer to the release document to ascertain the correct titles and versions of manuals for the release you are using.

Problems, Questions, and Suggestions

If you have any questions or problems with our hardware, software, or documentation, please contact either your HP Response Center or your local HP representative.

You may also use the Reader's Response Form at the back of this manual to submit comments about our documentation.

Documentation Conventions

Unless otherwise noted in the text, this manual uses the following symbolic conventions.

literal values	Bold words or characters in formats and command descriptions represent commands or keywords that you must use literally. Pathnames are also in bold. Bold words in text indicate the first use of a new term.
user–supplied values	Italic words or characters in formats and command descriptions represent values that you must supply.
sample user input	In interactive examples, information that the user enters appears in bold.
`output/source code`	Information that the system displays appears in this `typeface`. Examples of source code also appear in this `typeface`.
. . .	Vertical ellipsis points mean that irrelevant parts of a figure or example have been omitted.
——— ◊◊◊ ———	This symbol indicates the end of a chapter or part of a manual.

——— ◊◊◊ ———

Contents

Chapter 1 Overview of DCE

1.1 DCE Architecture 1–1
1.2 DCE Components 1–2
1.2.1 DCE Components Present in the HP DCE Developers' Environment 1–2
1.2.2 The RPC Component of the DCE 1–4
1.3 Differences between DCE RPC and NCS 1.5.1 1–4
1.3.1 Differences in the APIs 1–4
1.3.2 Differences in the Binding Mechanism 1–5
1.3.3 Differences in the Interface Definition Language 1–5
1.3.3.1 The DCE Attribute Configuration File 1–5
1.3.4 Differences in Global Location Services 1–6
1.3.5 Differences in Fault and Signal Handling 1–6
1.3.6 Support for Multiple Threads 1–6
1.3.7 Support for Concurrent Binding Handles 1–7
1.3.8 Differences in UUID Format 1–7
1.4 Converting NCS 1.5.1 Programs to DCE 1–8
1.4.1 DCE RPC API Concepts 1–8
1.4.2 DCE IDL Concepts 1–8

1.4.3 How to Write DCE Interface Definitions 1–8
1.4.4 Converting Clients and Servers 1–9
1.4.5 Handling Multiple Managers 1–9
1.4.6 Using DCE Location Services 1–9
1.4.7 Comparing DCE RPC and NCS 1.5.1 Terminology 1–9

Chapter 2 Understanding DCE RPC API Concepts

2.1 RPC Communications 2–1
2.1.1 Socket Addresses 2–1
2.1.2 DCE RPC String Bindings 2–2
2.1.3 DCE RPC Protocol Sequences 2–3
2.2 Bindings and Handles 2–4
2.2.1 Binding Information 2–4
2.2.2 Binding States: Unbound, Bound, and Partially Bound 2–5
2.2.3 Partially Bound Binding Handles 2–6
2.3 Overview of the DCE RPC API 2–6
2.3.1 Communication Services 2–7
2.3.2 Endpoint Map Services 2–8
2.3.3 Error Service 2–8
2.3.4 Management Services 2–8
2.3.5 String Services 2–8
2.3.6 UUID Services 2–8
2.3.7 Authentication Services 2–9
2.3.8 Stub Support Services 2–9
2.4 Changing NCS 1.5.1 Routines to DCE RPC Routines 2–9
2.4.1 Mapping of NCS 1.5.1 **rpc_$** Calls to DCE RPC Routines 2–9
2.4.1.1 Client Calls 2–10
2.4.1.2 Server Calls 2–11
2.4.1.3 Calls for Clients or Servers 2–12
2.4.2 Mapping of NCS 1.5.1 **rrpc_$** Calls to DCE RPC Routines 2–13
2.4.3 Mapping of NCS 1.5.1 **socket_$** Calls to DCE RPC Routines 2–13
2.4.4 Mapping of NCS 1.5.1 **lb_$** Calls to DCE RPC Routines 2–15
2.4.5 Mapping of NCS 1.5.1 **uuid_$** Calls to DCE RPC Routines 2–17

2.4.6 Mapping of NCS 1.5.1 **error_$** Calls to DCE RPC Routines 2–17
2.4.7 Replacement of the **pfm_$** Calls with the DCE Exception-Returning Package 2–18
2.5 Mapping of NCS 1.5.1 Data Types to DCE RPC Data Types 2–18
2.5.1 **error_$** and **status_$t** Data Types 2–18
2.5.2 **lb_$** Data Types 2–18
2.5.3 **pfm_$** Data Types 2–19
2.5.4 **rpc_$** Data Types 2–19
2.5.5 **rrpc_$** Data Types 2–20
2.5.6 **socket_$** Data Types 2–20
2.5.7 **uuid_$** Data Types 2–20
2.6 How to Migrate NCS 1.5.1 Applications to DCE RPC 2–21
2.6.1 Complete Conversion 2–21
2.6.2 Partial Conversion 2–21

Chapter 3 Overview of the DCE Interface Definition Language

3.1 An Overview of DCE Interface Definitions 3–1
3.2 DCE IDL Data Types and Attributes 3–2
3.2.1 Base (Simple) Types 3–2
3.2.2 Pipes and Pipe Attributes 3–3
3.2.3 Pointers 3–3
3.2.4 Arrays and Array Attributes 3–4
3.2.5 Strings 3–5
3.2.6 Customized Handles 3–6
3.2.7 Context Handles 3–6
3.2.8 Enumerations 3–7
3.2.9 Unions 3–7
3.2.10 Structures 3–7
3.2.11 Attributes for Compatibility with NCS 1.5.1 3–7
3.2.12 Sets 3–7
3.3 Structure of the DCE IDL Interface Definition 3–7
3.4 Interface Names 3–8
3.5 Interface Definition Attributes 3–9

3.6 Import Declarations 3–11
3.7 Constant Declarations 3–11
3.8 Type Declarations 3–12
3.8.1 Type Attributes 3–12
3.8.2 Type Specifiers 3–12
3.8.3 Type Declarators 3–13
3.9 Operation Declarations 3–14
3.10 Parameter Declarations 3–15
3.11 The Attribute Configuration File 3–15
3.11.1 Binding Attributes: **auto_handle**, **explicit_handle**, **implicit_handle** 3–17
3.11.2 Handling Errors with **comm_status** and **fault_status** 3–18
3.11.3 Controlling Client Stub Generation with **code** and **nocode** 3–18
3.11.4 Controlling the Marshalling of Code with **in_line** and **out_of_line** 3–19
3.11.5 Controlling Data Representation with **represent_as** 3–19
3.11.6 Initializing Memory Management Routines with **enable_allocate** 3–19
3.11.7 Allocating Objects from the Heap 3–19
3.12 Example of an Interface Definition and Attribute Configuration File 3–20
3.13 DCE IDL Output Files 3–21

Chapter 4 Writing DCE Interface Definitions

4.1 Using the **nidl_to_idl** Tool to Convert NCS 1.5.1 Interface Definitions 4–1
4.1.1 The **nidl_to_idl** Tool for Translating Interface Definitions 4–1
4.1.2 Invoking the **nidl_to_idl** Tool 4–2
4.1.3 IDL Attributes for Compatibility with NCS 1.5.1 4–4
4.1.4 Creating an Attribute Configuration File 4–6
4.1.5 Converting from the NIDL Pascal Syntax 4–7
4.2 Writing New DCE Interface Definitions 4–9
4.2.1 Generating Interface UUIDs 4–9
4.2.2 Writing the Interface Definition 4–10
4.2.2.1 Naming the Interface 4–10
4.2.2.2 Specifying Interface Definition Attributes 4–10
4.2.2.3 Specifying Import Declarations 4–10
4.2.2.4 Specifying Constant Declarations 4–10

4.2.2.5 Specifying Type Declarations 4–11
4.2.2.6 Specifying Operation Declarations 4–11
4.2.3 Writing an Attribute Configuration File 4–11
4.3 Running the IDL Compiler 4–12

Chapter 5 Converting Distributed Applications to DCE RPC

5.1 Converting the Client Code 5–1
5.2 Converting the Server Code 5–6
5.2.1 Initializing a DCE Server 5–6
5.2.2 The **server.c** Module 5–7
5.2.3 The **manager.c** Module 5–11
5.3 Converting the **util.c** Module 5–11
5.4 Building DCE RPC Applications 5–12
5.5 Running the **binopfw** Program 5–13
5.5.1 Starting the Server Program 5–13
5.5.2 Starting the Client Program 5–13
5.6 Improving the **binopfw server.c** Program 5–14
5.7 Handling Signals in DCE RPC 5–16
5.7.1 Using Exceptions with the DCE Exception-Returning Package 5–17
5.7.2 Handling Asynchronous Signals 5–18
5.8 No Replacement for the **rpc_$set_fault_mode** 5–20
5.9 Using NCS 1.5.1 and DCE RPC UUIDs 5–20

Chapter 6 Writing Servers with Multiple Managers

6.1 The **stacks** Interface Definition 6–1
6.2 Generating the Object UUIDs in the stackdf.h File 6–2
6.3 The **stacks** Client Module 6–4
6.4 The **stacks** Server Module 6–9
6.5 The **stacks util.c** Module 6–17

Chapter 7 Using DCE Location Services

7.1 The Endpoint Map Service 7–2
7.1.1 Mapping NCS 1.5.1 **lb_$** Calls to DCE RPC **rpc_ep** Routines 7–2
7.1.2 Using **rpc_ep** Routines in a Client 7–3
7.1.3 Using **rpc_ep** Routines in a Server 7–3
7.1.4 Using **rpc_mgmt_ep** Routines in a Manager 7–5
7.2 The Name Service 7–5
7.2.1 Mapping of **lb_$** Calls to **rpc_ns** Routines 7–6
7.3 Client Name Service Routines 7–7
7.3.1 Importing Binding Handles 7–8
7.3.2 Looking up a Set of Binding Handles 7–12
7.4 Server Name Service Routines 7–13
7.5 The **lookup** Sample Application 7–18

Appendix A DCE RPC Routines

A.1 Authentication Services A–2
A.2 Communication Services A–3
A.2.1 Binding Routines A–4
A.2.1.1 Create a Binding Handle from a String Binding A–4
A.2.1.2 Release a Binding Handle A–4
A.2.1.3 Copy a Binding Handle A–4
A.2.1.4 Change a Server Binding A–4
A.2.1.5 Convert a Binding Handle A–5
A.2.1.6 Get Binding Information A–6
A.2.1.7 Set Binding Information A–6
A.2.1.8 Convert a Client Binding Handle A–6
A.2.2 Interface Routines A–7
A.2.3 Network Routines A–7
A.2.4 Object UUID Routines A–8
A.2.5 Server Routines A–8
A.2.5.1 Register Protocol Sequences A–8

A.2.5.2 Other Server Initialization A–9
A.3 Endpoint Map Services A–11
A.4 Error Services A–13
A.5 Management Services A–14
A.5.1 Local Management Routines A–14
A.5.2 Local/Remote Management Routines A–16
A.5.2.1 Routines Used by All Applications A–16
A.5.2.2 Routines Used by Management Applications A–17
A.6 Name Service A–18
A.6.1 Export a Server to the Name Service A–19
A.6.2 Search a Name Service Database for Binding Information A–20
A.6.2.1 Automatic Binding A–20
A.6.2.2 Import Routines A–20
A.6.2.3 Lookup Routines A–21
A.6.3 Manage Name Service Entries A–21
A.6.3.1 Find Entries A–21
A.6.3.2 Create and Delete Entries A–22
A.6.3.3 View Objects of an Entry A–22
A.6.3.4 Get Information from Entries A–22
A.6.4 Managing Name Service Groups A–23
A.6.4.1 Delete a Group A–23
A.6.4.2 Add and Remove Group Members A–23
A.6.4.3 View Members of a Group A–24
A.6.5 Managing Name Service Expirations A–24
A.6.6 Managing Name Service Profiles A–24
A.6.6.1 Delete a Profile Attribute A–25
A.6.6.2 Add and Remove Profile Elements A–25
A.6.6.3 Obtain Profile Elements A–25
A.7 String Services A–26
A.8 Stub Support Services A–26
A.8.1 Using the Stub Memory Management Scheme A–26
A.8.1.1 Allocate and Free Memory on a Server A–26
A.8.1.2 Enable and Disable Allocation by **rpc_ss_allocate** A–27
A.8.1.3 Establish Routines that Free and Allocate Memory A–27

A.8.1.4 Change the Current Allocation and Freeing Mechanism A–27
A.8.2 Using Thread Handles in Memory Management A–28
A.8.3 Other Memory Management Routines A–28
A.9 UUID Services A–29
A.10 DCE RPC Runtime Routine Summary A–30
A.10.1 DCE RPC Client Runtime Routines A–30
A.10.1.1 Binding Routines A–30
A.10.1.2 Interface Routines A–30
A.10.1.3 Network Routines A–30
A.10.1.4 Endpoint Map Services A–31
A.10.1.5 Error Services A–31
A.10.1.6 Inquire of Protocol Sequences A–31
A.10.1.7 Local Management Services A–31
A.10.1.8 Local/Remote Management Services A–31
A.10.1.9 String Services A–31
A.10.1.10 Name Service Routines A–31
A.10.1.11 UUID Services A–33
A.10.1.12 Stub Support Routines A–33
A.10.2 DCE RPC Server Runtime Routines A–33
A.10.2.1 Binding Routines A–33
A.10.2.2 Interface Routines A–34
A.10.2.3 Network Routines A–34
A.10.2.4 Object UUID Routines A–34
A.10.2.5 Server Routines A–34
A.10.2.6 Endpoint Map Services A–35
A.10.2.7 Managing Binding Handles A–35
A.10.2.8 Error Services A–35
A.10.2.9 Local Management Services A–35
A.10.2.10 Local/Remote Management Services A–35
A.10.2.11 String Services A–35
A.10.2.12 Managing the Server A–36
A.10.2.13 Name Service Routines A–36
A.10.2.14 Stub Support Routines A–37
A.10.2.15 UUID Services A–37
A.10.3 DCE Management Application Runtime Routines A–38

A.10.3.1 Binding Routines A–38
A.10.3.2 Interface Routines A–38
A.10.3.3 Error Services A–38
A.10.3.4 Endpoint Map Services A–38
A.10.3.5 Local Management Services A–38
A.10.3.6 Local/Remote Management Services A–38
A.10.3.7 String Services A–39
A.10.3.8 Name Service Routines A–39
A.10.3.9 UUID Services A–40

Appendix B NCS 1.5.1 Client and Server Programs

B.1 NCS 1.5.1 **binopfw** Program B–1
B.1.1 Client Code B–1
B.1.2 NCS 1.5.1 **util.c** B–4
B.1.3 NCS 1.5.1 **server.c** B–4
B.1.4 NCS 1.5.1 **manager.c** B–8
B.2 NCS 1.5.1 **stacks** Program B–8
B.2.1 The **stacks** Interface Definition B–8
B.2.2 The **stacksdf.h** Header File B–9
B.2.3 The **stacks** Client Module B–10
B.2.4 The **stacks** Server Module B–13

Appendix C IDL and ACF Attribute Summary

C.1 DCE IDL Attribute Summary C–1
C.1.1 IDL Attributes in Interface Definition Headers C–1
C.1.2 IDL Attributes for Operations C–3
C.1.3 IDL Attributes for Parameters C–3
C.1.4 IDL Attributes for Structures C–3
C.1.5 IDL Attributes for Unions C–3
C.1.6 IDL Attributes for Arrays C–4

C.1.6.1 Conformant Array Attributes C–4
C.1.6.2 Varying Array Attributes C–4
C.1.7 IDL Attributes for Pointers C–4
C.1.8 IDL Attributes for Customized Handles C–4
C.1.9 IDL Attributes for Context Handles C–5
C.1.10 IDL Attributes for Type Declarations C–5
C.1.11 IDL Attributes for Compatibility with NCS 1.5.1 ..... C–5
C.2 DCE ACF Attribute Summary C–6

Glossary

Figures

Figure 1–1. Where DCE Resides in a Software Architecture 1–2
Figure 2–1. Comparison of a Socket Address and a DCE String Binding 2–3
Figure 3–1. DCE Interface Definition 3–20
Figure 3–2. DCE Attribute Configuration File 3–20
Figure 4–1. NCS 1.5.1 Interface Definition 4–3
Figure 4–2. DCE Interface Definition Created with **nidl_to_idl** 4–3
Figure 4–3. NCS 1.5.1 Interface Definition, **string.idl** 4–5
Figure 4–4. Conversion of **string.idl** to **string_v2.idl** 4–5
Figure 4–5. NCS 1.5.1 Interface Definition, **params.idl** 4–6
Figure 4–6. **nidl_to_idl** Warning 4–6
Figure 4–7. Conversion of **params.idl** to **params_v2.idl** 4–7
Figure 4–8. The **params_v2.acf** Attribute Configuration File 4–7
Figure 4–9. NCS 1.5.1 Interface Definition in NIDL Pascal Syntax 4–8
Figure 4–10. Conversion of the Pascal Interface Definition to DCE IDL 4–8
Figure 5–1. DCE RPC Version of **binopfw/client.c** 5–3
Figure 5–2. DCE RPC Version of **binopfw/server.c** 5–8
Figure 5–3. DCE RPC Version of **binopfw/manager.c** 5–11
Figure 5–4. DCE RPC Version of **binopfw/util.c** 5–11
Figure 5–5. Server Using **rpc_server_use_all_protseqs** 5–15
Figure 5–6. Handling Asynchronous Signals 5–19
Figure 5–7. Comparison of **uuid_$t** and **uuid_t** 5–21
Figure 5–8. Changing the NCS 1.5.1 UUID C Initialization 5–21
Figure 6–1. The DCE RPC **stacks/stacks.idl** Interface Definition 6–2
Figure 6–2. The **stacks/stacksdf.h** Header File 6–3
Figure 6–3. The **stacks/client.c** Module 6–5
Figure 6–4. The **stacks/server.c** Module 6–10
Figure 6–5. The **stacks/amanager.c** Module 6–15
Figure 6–6. The **stacks/lmanager.c** Module 6–16
Figure 6–7. The **stacks/util.c** Module 6–17
Figure 7–1. Registering and Unregistering with the Local Endpoint Map Database 7–4

Figure 7–2. Name Service Routines in the **string_conv/client.c** Program 7–9
Figure 7–3. Name Service Routines in the **string_conv/server.c** Program 7–14
Figure B–1. NCS 1.5.1 Version of **binopfw/client.c** B–2
Figure B–2. NCS 1.5.1 Version of **binopfw/util.c** B–4
Figure B–3. NCS 1.5.1 Version of **binopfw/server.c** B–5
Figure B–4. NCS 1.5.1 Version of **binopfw/manager.c** B–8
Figure B–5. The NCS 1.5.1 **stacks.idl** Interface Definition B–9
Figure B–6. The NCS 1.5.1 **stacksdf.h** Header File B–9
Figure B–7. The NCS 1.5.1 **stacks/client.c** Module B–10
Figure B–8. The NCS 1.5.1 **stacks/server.c** Module B–13
Figure B–9. The NCS 1.5.1 **stacks/lmanager.c** Module B–17
Figure B–10. The NCS 1.5.1 **stacks/amanager.c** Module B–18

Tables

Table 3–1. Comparison of IDL and NIDL Interface Attributes 3–9
Table 3–2. DCE Type Specifiers 3–13
Table 3–3. Comparison of IDL and NIDL Operation Attributes 3–14
Table 3–4. NIDL and IDL Output Files 3–21
Table 7–1. RPC Endpoint Map Equivalents to Location Broker Calls 7–3
Table 7–2. RPC Name Service Equivalents to Location Broker Calls 7–6
Table C–1. DCE IDL Attributes C–2
Table C–2. DCE ACF Attributes C–6

Chapter 1

Overview of DCE

This chapter provides an overview of the OSF Distributed Computing Environment (DCE) with an emphasis on the DCE Remote Procedure Call (RPC) facility. This chapter also compares the DCE RPC Application Programming Interface (API) and the NCS 1.5.1 API. Subsequent chapters provide more details on DCE RPC programming features and how they compare with the corresponding NCS 1.5.1 features.

1.1 DCE Architecture

The OSF Distributed Computing Environment is a set of services that supports the creation, use, and maintenance of distributed applications in a heterogeneous environment. It provides the services that allow a distributed application to interact with a collection of computers, operating systems, and networks that may be heterogeneous as if they were a single system. For a more detailed overview of DCE, see the *Introduction to OSF DCE.*

From an architectural perspective, DCE is a layer of software that resides above the operating system and network services. It resides below the distributed applications that rely on the DCE layer. Figure 1–1 shows where DCE resides in a software architecture.

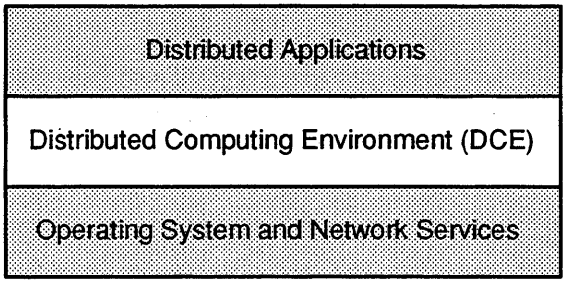


Figure 1–1. Where DCE Resides in a Software Architecture

The network on which DCE resides can be a local or wide area network or a combination of both. DCE can run over different types of network protocol families (such as Internet or OSI) as long as the hosts share a common set of protocols.

While DCE can be pictured as a single layer of software, it actually consists of several related components as described below.

1.2 DCE Components

This section contains a brief description of the DCE components that are supported by the HP DCE Developers' Environment followed by a more detailed description of the DCE Remote Procedure Call (RPC) component. For more information about DCE, see the *Introduction to OSF DCE.*

1.2.1 DCE Components Present in the HP DCE Developers' Environment

The following DCE components are present in the HP DCE Developers' Environment Release:

Threads	Provides support for creating, managing, and synchronizing multiple threads of control within a single process. This package is compliant with draft 4 of *Threads Extensions for Portable Operating Systems* proposed by the POSIX 1003.4a standards committee. RPC clients and servers may be threaded programs. For more information

about using threads, see *Programmer's Notes on HP DCE Threads.*

Remote Procedure Call	Provides an application programming interface (API) that allows programmers to develop distributed applications using the client/server model. It defines the interface definition language (IDL) and provides the IDL compiler, which generates code that transforms procedure calls into network messages. It also includes a runtime service which implements the network protocols by which the client and server sides of an application communicate.
Security Service	Provides secure communications and controlled access to resources in the distributed system.
Directory Service	Serves as the central source for much of the necessary information in the distributed system. It keeps track of users, machines, and RPC-based services such as the Time and Security Services. The Directory Service actually consists of three components: the Cell Directory Service (CDS), the Global Directory Service (GDS), and the Global Directory Agent (GDA). The CDS keeps track of a group (called a CDS cell) of machines that are typically located near each other. The GDS keeps track of a worldwide database of resources by implementing the ISO X.500 Directory Service specifications. The GDA manages transactions between cell and global directory services. HP's DCE product does not support the GDS and, instead, uses the Domain Name Service (DNS) as the global directory service. Therefore, in HP's DCE product, the GDA manages the interaction between the CDS and the DNS.
Time Service	Provides synchronized time on the computers in a distributed environment using the UTC, a universal time standard.
Management	Provides tools for managing the various components in a distributed network. Each DCE service above provides ways to manage the component over the network.

1.2.2 The RPC Component of the DCE

RPC is the communication mechanism used by all other DCE components. In addition, RPC provides an application programming interface (API) to other components.

The RPC component provides access to other DCE components. For example, RPC provides access to the DCE Directory Services through the Name Service Interface (NSI); NSI routines begin with the **rpc_ns** prefix. Similarly, the RPC component provides access to the DCE Security Service with the RPC routines (such as **rpc_binding_inq_auth_info**) that have **auth** in the routine name.

Applications written with DCE RPC can support connection-oriented transport services (such as TCP) and connection-oriented network services (such as X.25), as well as datagram–oriented services (such as UDP).

For detailed information about programming in a DCE environment, see the *OSF DCE Application Development Guide* and the *OSF DCE Application Development Reference.*

1.3 Differences between DCE RPC and NCS 1.5.1

In the process of creating a standard, portable RPC implementation for DCE, several changes were made to NCS 1.5.1. Some changes represent enhancements to NCS, such as support for pointers and pipes. Other changes improve the portability of the software; for example, the API routine names no longer contain the dollar sign ($). Some other changes only affect names, while the underlying concepts remain the same.

This section highlights some major differences between the NCS 1.5.1 and DCE RPC APIs.

NCS 1.5.1 programs can communicate with DCE RPC programs; for example, an NCS 1.5.1 client can call a DCE server. Within a single client or server, however, you can use routines from only one API; you cannot call both NCS 1.5.1 and DCE RPC routines from within the same program.

1.3.1 Differences in the APIs

The DCE RPC API replaces the application programming interface that is available with NCS 1.5.1. The DCE RPC API is designed to better meet the needs of today's RPC systems and the communication requirements of the DCE environment.

1.3.2 Differences in the Binding Mechanism

The DCE RPC API does not provide any routines that directly manipulate socket addresses. Details of addressing are internal and not accessible at the API level. Instead, binding handles are represented in either string or binary format.

1.3.3 Differences in the Interface Definition Language

The DCE Interface Definition Language (IDL) and compiler are enhanced versions of the NCS 1.5.1 Network Interface Definition Language (NIDL) and compiler. Features include

- Enhanced support for pointers. DCE IDL supports both reference and full pointers, and has lifted many of the NCS 1.5.1 restrictions on pointers. A pointer can now have a NULL value or be an alias. Pointers no longer need to be at the "top level."
- Enhanced support for constants. DCE IDL supports integer, boolean, character, string, and null pointer constants. It also supports constant expressions.
- Support for pipes, which provide a mechanism for transferring large quantities of typed data.
- Enhanced support for open (now called conformant) arrays, which are arrays whose size is determined at run time.
- Support for context handles, which provide the ability to maintain state information across remote procedure calls.

DCE provides a translating tool, **nidl_to_idl**, that allows you to automatically convert an NCS 1.5.1 interface definition into a format appropriate for the DCE IDL compiler. Chapter 4 describes how to use this tool.

The IDL compiler supports only a C-like syntax; it does not support a Pascal-like syntax as in NCS 1.5.1. However, you can use the **nidl_to_idl** translator to automatically generate a DCE interface definition from an NCS 1.5.1 NIDL file that was created in the Pascal-like syntax.

Another difference between IDL and NIDL is that some information that was previously defined in the interface definition now belongs in the **Attribute Configuration File** (**ACF** or **.acf** file).

1.3.3.1 The DCE Attribute Configuration File

In the DCE programming environment, the attribute configuration file (ACF) allows you to define attributes that modify the interaction between the application and stubs without

changing the IDL file. Rather than having all attributes defined in a single file, information is separated into two files as follows:

Interface definition file	Contains any information pertinent to the client/server contract (that is, the interaction between the client and server stubs).
Attribute configuration file	Contains information pertinent to the local behavior (that is, the interaction between application code and stub code).

The attribute configuration file (ACF) allows application programmers to customize their applications more easily without affecting interoperability ("over the wire") of existing DCE interfaces. Application programs built from the same interface definition file are guaranteed to interoperate; differences between ACF files are guaranteed to not affect interoperability. The client and server sides of an application must be built using the same interface definition, but they can be built using different attribute configuration files.

For more details on how DCE IDL differs from NCS 1.5.1 NIDL, see Chapters 3 and 4.

1.3.4 Differences in Global Location Services

To enable clients to locate possible servers in a network dynamically, the DCE RPC API uses a name service mechanism rather than the NCS 1.5.1 Global Location Broker service. Clients can locate servers using either the name service interface (NSI) or a mechanism such as passing in a host name from the command line.

1.3.5 Differences in Fault and Signal Handling

The DCE RPC API does not use the Process Fault Management (**pfm_$**) routines for fault and signal handling. Instead, DCE uses the DCE exception-returning package (sometimes referred to by the macros it contains, TRY/CATCH). This package allows programmers to map signals to exceptions and then catch the exceptions when performing cleanup operations. This package is made available by the underlying operating system for multithreaded programs. Chapter 5 provides a programming example illustrating how to use the macros defined in this exception-returning package. Refer to the *OSF DCE Application Development Guide* for more information about the package.

1.3.6 Support for Multiple Threads

The NCS RPC runtime does not create multiple threads unless a server specifically requests that they be created. An NCS server can request multiple threads of execution by calling

rpc_$listen with a number greater than one for the *max_calls* parameter, which specifies how many concurrent requests are allowed. This feature is only available on platforms on which Concurrent Programming Support (CPS) is implemented.

The DCE RPC runtime automatically creates multiple threads, so that all clients and servers are multithreaded. There are several aspects of multithreaded programming using the DCE Threads package of which you must be aware. We recommend that you refer to *Programmer's Notes on HP DCE Threads* for more information.

1.3.7 Support for Concurrent Binding Handles

The DCE RPC runtime permits DCE RPC-based applications to use a single binding handle for making concurrent remote procedure calls. NCS 1.5.1 applications cannot use a single binding handle for making concurrent remote procedure calls; each thread must create its own binding handle (via **rpc_$bind**) and then maintain it.

Still, for both NCS 1.5.1-based and DCE RPC-based multithreaded programs, you must take the same precautions to avoid concurrency conflicts when multiple threads try to operate on the same object simultaneously. For details on writing multithreaded programs, refer to the *OSF DCE Application Development Guide* and *Programmer's Notes on HP DCE Threads.*

1.3.8 Differences in UUID Format

The DCE RPC UUID format is different from that of NCS 1.5.1. The DCE RPC **uuid_t** is the same size as the NCS 1.5.1 **uuid_$t**, but it has a different internal structure. (See Section 5.9 for a comparison of UUID internal structures.) Because the internal structures are different, generating UUIDs with C initializations (using **uuidgen** for a DCE RPC-style UUID and **uuid_gen** for a NCS 1.5.1-style UUID) produces different output.

In programs that attempt to use both NCS 1.5.1 and DCE RPC UUID routines, a **uuid_t** can be passed where a **uuid_$t** is expected and vice versa, but conversion to and from **uuid_$string_t** can lose information.

The **uuid_$nil** global variable has no replacement in the DCE RPC API. The NCS 1.5.1 API provides a nil UUID as a global variable. With the DCE RPC API, you use the **uuid_create_nil** routine to create a nil UUID.

1.4 Converting NCS 1.5.1 Programs to DCE

You may wish to convert NCS 1.5.1 applications to DCE in order to take advantage of DCE features not supported by NCS 1.5.1, or you may wish to convert a working NCS application to DCE as a learning experience or to evaluate the features and benefits of DCE that are supported by the HP DCE Developers' Environment.

If your NCS 1.5.1 application relies on dynamic binding to locate server hosts, you must revise it to use either the DCE name service interface or to use string bindings to identify server hosts. As described in Section 1.3.4, applications that use the DCE RPC API cannot access the Global Location Broker.

The rest of this book describes how to convert existing NCS 1.5.1 applications to the DCE RPC API. The following sections summarize the topics covered in the rest of this book.

1.4.1 DCE RPC API Concepts

Chapter 2 introduces some major concepts that underly the DCE RPC API and explains how it differs from the NCS 1.5.1 API. It also describes how the DCE RPC API is structured and how its routines and data types differ from those of the NCS 1.5.1 API.

1.4.2 DCE IDL Concepts

Chapter 3 provides an overview of how the DCE Interface Definition Language differs from the Network Interface Definition Language (NIDL) in NCS 1.5.1. It introduces the new syntax features available with this release.

1.4.3 How to Write DCE Interface Definitions

The first step in developing a distributed application is to define its interface (or interfaces) using the DCE Interface Definition Language (IDL). Chapter 4 provides details on the process of writing interface definitions with some examples.

Also, DCE provides the **nidl_to_idl** tool to automatically convert NCS 1.5.1 NIDL files into IDL files. Note that **nidl_to_idl** takes care of most, but not all of the translation automatically. See Chapter 4 for a description of how to use this tool and how to invoke the IDL compiler to create stubs and header files for DCE RPC-based applications.

1.4.4 Converting Clients and Servers

Chapter 5 provides a programming example that shows how to convert an existing NCS 1.5.1 application (a client and a server) to the DCE RPC API and how to compile, link, and execute the resulting programs. Chapter 5 also provides an example of using the TRY/CATCH macros for fault handling.

1.4.5 Handling Multiple Managers

Chapter 6 describes how to handle multiple managers in a DCE RPC-based application by presenting a DCE RPC version of the **stacks** application that was provided with NCS 1.5.1. This sample application is available online as part of the DCE Programmers' Environment.

1.4.6 Using DCE Location Services

Chapter 7 contrasts the NCS location broker service with the DCE endpoint map and name services, and summarizes the steps that clients and servers must take to access the DCE services.

1.4.7 Comparing DCE RPC and NCS 1.5.1 Terminology

A glossary highlights the differences in terminology between NCS 1.5.1 and DCE RPC. You may be able to get a high-level understanding of the differences between these releases by reading this glossary.

Chapter 2

Understanding DCE RPC API Concepts

This chapter provides an overview of the DCE RPC application programming interface (API). It also describes how the DCE RPC API and its underlying concepts differ from those of NCS 1.5.1. Subsequent chapters describe the differences between the NCS Network Interface Definition Language (NIDL) and the DCE RPC Interface Definition Language (IDL) and then provide programming examples demonstrating these differences.

This chapter discusses the conversion of NCS 1.5.1 calls only. If your program uses any other non-HP-UX calls (such as Domain/OS calls), you must convert those calls. If your application is not written in C, you should convert it to C, preferably ANSI C.

2.1 RPC Communications

In DCE RPC, as in NCS, the RPC runtime library is independent of any underlying communications protocol. The destination address of a message automatically determines the protocol used; both the sending and receiving host must support the protocol.

2.1.1 Socket Addresses

The NCS 1.5.1 API contains routines requiring socket addresses. These socket addresses are based on the Berkeley socket abstraction, though the NCS socket address is more generalized to provide a transport-independent API for interprocess communication.

In the NCS 1.5.1 API, addressing information is represented by strings; these strings are converted to socket addresses (the **socket_$addr_t** data type), and the socket addresses are then converted to binding handles. Some NCS 1.5.1 routines require socket addresses as input parameters.

The socket abstraction is not sufficient to completely represent the addressing needs of the DCE RPC system. The DCE RPC API enhances the string representation to contain additional required information. None of the DCE RPC API routines require socket addresses as input parameters; DCE routines require address information in string format only. These strings are then converted to binding handles.

2.1.2 DCE RPC String Bindings

The DCE RPC string representation, or **string binding**, contains the character representation of a network destination. A string binding contains character strings that represent the following:

- An object UUID, which specifies the UUID of the object to be operated on by the remote procedure that is called using this string binding. If this optional field is not provided, the RPC runtime assumes a nil UUID.
- An RPC protocol sequence identifier, which identifies the RPC, network, and transport protocols to use for making remote procedure calls. (The DCE RPC protocol sequence ID replaces the NCS 1.5.1 protocol family ID.)
- A network address of the host on the network that receives remote procedure calls made with this string binding. The format and content depend on the protocols defined in the RPC protocol sequence identifier field. (The formats of IP network addresses are the same in DCE RPC and NCS 1.5.1.)
- An **endpoint**, which is the address of the specific server instance on a host that receives remote procedure calls made with this string binding. (The DCE RPC endpoint replaces the NCS 1.5.1 port.)

A string binding is not required to contain all of this information; only the protocol sequence is always required:

- The UUID is optional
- The protocol sequence is required
- If the server is on the client's system, the network address is optional
- The endpoint is optional

For a detailed description of string bindings, see the **intro_rpc(3)** manual page.

Figure 2–1 compares the NCS 1.5.1 and DCE RPC string binding representations.

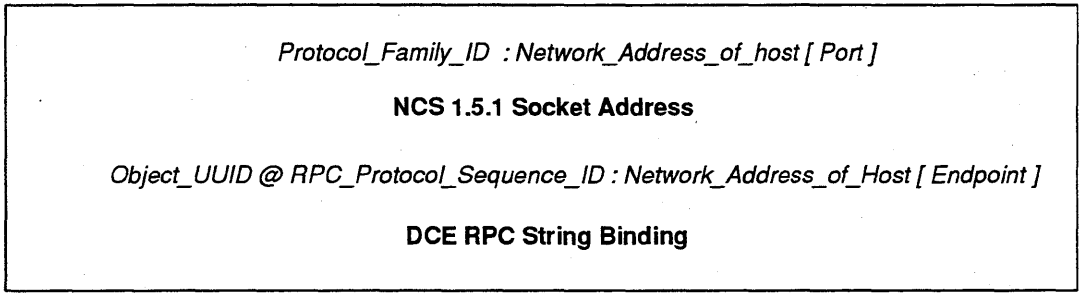


Figure 2–1. Comparison of a Socket Address and a DCE String Binding

A DCE RPC string binding (without an object UUID) looks like the following. See Section 2.1.3 for an explanation of the protocol sequences.

For **ncacn_ip_tcp:** `ncadg_ip_tcp:15.22.145.3[1035]`
For **ncadg_ip_udp:** `ncadg_ip_udp:15.22.145.3[3925]`

2.1.3 DCE RPC Protocol Sequences

This section describes the DCE protocol sequences supported by the HP DCE Developers' Environment. In the DCE RPC API, a **protocol sequence** replaces the NCS 1.5.1 **address family** for identifying the communications protocols used to establish a relationship between a client and a server. In NCS 1.5.1, while the address family specifies the network protocol layer of the communications protocol stack, the transport and RPC layers are defined implicitly. In DCE RPC, the protocol sequence typically specifies a network protocol, a transport protocol, and an RPC protocol that works with them, that is, all the layers of the communications stack.

The following predefined strings represent protocol sequences supported by the HP DCE Developers' Environment:

Protocol Sequence	Description
ncacn_ip_tcp	Network Computing Architecture (NCA) Connection for the RPC protocol layer over Internet Protocol; Transmission Control Protocol for the network and transport layers.
ncadg_ip_udp	NCA Datagram service over the Internet Protocol; User Datagram Protocol for the RPC, network, and transport protocol layers respectively.

2.2 Bindings and Handles

For both NCS and DCE, the act of creating a relationship between a client and server occurs at binding. In remote procedure calls, a binding is a relationship between the calling client and a server. Before a client can initiate remote procedure calls, it must first establish a binding with a server that can respond to its requests.

2.2.1 Binding Information

Before a client can establish a binding, it must first identify a server that offers the requested RPC interface and object and that supports the same communications protocols as the client. A server that meets these criteria is called a **compatible server.** (NCS 1.5.1 did not formally define a compatible server, but the requirement existed nonetheless.)

In the DCE RPC API, a client gets binding information for a compatible server from one of the following

- A string representation of the binding information, which can come from many possible choices, such as an application-specific variable, a file, or the command line.
- A local database on a server's host, if the server's host name is specified. This database contains a list of servers running on the specific host which have previously registered an interface. A local daemon (**rpcd** in DCE RPC, **llbd** in NCS 1.5.1) forwards remote procedure calls to the server or returns server endpoint information to the client. The database is called the **endpoint map** in DCE RPC and the **Local Location Broker (LLB) database** in NCS 1.5.1.

- A name service database. This database contains binding information for servers that have previously exported their RPC interfaces and bindings as part of their initialization process.

In NCS 1.5.1, a client receives binding information from either a string, the Local Location Broker (LLB), or the Global Location Broker (GLB). The GLB is similar to a name service; however, it contains a database of UUIDs rather than names. The GLB is part of NCS and is accessed through **lb_$** routines.

A DCE client receives binding information from either an application-specific source or a name service. If the client receives a binding in string format, it must use the **rpc_binding_from_string_binding** routine to obtain a binding handle. A client can use the name service database to obtain a binding either by using the automatic binding method or by searching the database for a compatible server. In the automatic method of binding, the client stub transparently manages the binding. Alternatively, the client can use either the import routines (**rpc_ns_binding_import_***) or the lookup routines (**rpc_ns_binding_lookup_***) to search for a compatible server and obtain a binding handle.

A binding handle serves as a contract between the client and server and exists for the lifetime of the application, unless the application explicitly breaks the relationship (by resetting the binding handle). This is a logical connection that remains active regardless of the underlying physical network activity.

2.2.2 Binding States: Fully Bound, Partially Bound and Unbound

Another distinction between NCS 1.5.1 and DCE RPC is how well a client needs to identify a server when it creates a binding handle.

In NCS 1.5.1, a client could create the binding handle in one of three different states depending on how completely the server location is specified:

fully bound — Contains a complete binding, that is, it identifies a specific host (network address), and a specific server instance on that host (an endpoint). Also called **bound-to-server** in NCS 1.5.1.

partially bound — Specifies the address of a specific host, but no particular server process running on that host (that is, it lacks a port number or

endpoint). This is also called **bound-to-host** in NCS 1.5.1. A partially bound handle becomes fully bound after the RPC runtime forwards the handle to the compatible server.

unbound Does not specify a specific host.

In the DCE RPC API, a client can create only a fully bound or partially bound binding handle. The DCE RPC API does *not* allow a client to create an unbound handle. In NCS 1.5.1, if a client uses an unbound handle to make a remote procedure call, the RPC runtime broadcasts the request to all hosts on the local network. The only way you can broadcast remote procedure calls in DCE RPC is by explicitly labeling an operation with the **broadcast** attribute in the interface definition.

2.2.3 Partially Bound Binding Handles

In both NCS 1.5.1 and DCE RPC, a client often makes its first remote procedure call with a partially bound handle, and the RPC runtime supplies the specific service instance (endpoint) dynamically. This endpoint is sometimes called a **dynamic** endpoint because neither the client nor the server needs to know the actual value of the endpoint; the RPC runtime takes care of it.

Dynamic endpoints are registered in a local database on the server's system. In NCS 1.5.1, the local database is called the Local Location Broker (LLB) database, and is maintained by the Local Location Broker Daemon (**llbd**). In the DCE RPC API, the local database is called the local endpoint map, and the RPC daemon (**rpcd**) maintains this database.

Entries are added to the local endpoint map database each time a server starts up. That is, when a server initializes, it registers its endpoint with the local endpoint map. If the server does not use a well-known endpoint, endpoints are assigned dynamically by the RPC runtime and can change each time a server starts up.

After the client makes the initial remote procedure call, the returned binding handle is fully bound, and the client does not need to use any forwarding mechanism for subsequent calls to that server. This is the same for DCE RPC and NCS 1.5.1.

2.3 Overview of the DCE RPC API

This section describes how the DCE RPC API is organized. For complete details on the routines in the API, see the *OSF DCE Application Development Reference.*

The DCE RPC API is composed of the following services:

- Communication Services
- Endpoint Map Services
- Error Service
- Management Services
- String Services
- UUID Services
- Authentication Services
- Stub Support Services

For a listing of the specific routines in each service, see Appendix A.

2.3.1 Communication Services

Communication Services contain the API routines related to establishing a client/server relationship (binding) and inquiring about the binding. The routines are categorized as follows: binding, interface, network, object, protocol sequence, and server.

Binding	Routines that client and server applications can use to manipulate binding handles.
Interface	Routines that client, server, and management applications can use to get and release memory for the interface identification structures.
Network	Routines that client and server applications can use to get the protocol sequences supported by the RPC runtime and operating system, and to check that a protocol sequence is valid.
Object	Routines that a server application can use to get the type of an object, register an object inquiry function, and assign a type to an object.
Protocol Sequence	A routine that client and server applications can use to free protocol sequences contained in a vector.
Server	Routines that a server application can use to initialize a server application and then listen for remote procedure calls.

2.3.2 Endpoint Map Services

The Endpoint Map Services provide routines for manipulating the local endpoint map. The local endpoint map is a database that resides on each DCE host; it contains a listing of the endpoints on which servers are listening. As part of the server initialization, the server application registers with the RPC daemon (**rpcd**), which maintains the local endpoint map.

The Endpoint Map Services provide routines that server applications can use to add, modify, or remove information from the local endpoint map database. They also provide a routine for client and management applications to get a fully bound handle from a partially bound handle.

2.3.3 Error Service

The Error Service provides an error routine to handle status information from DCE RPC-based programs. The error routine is used by client, server, and management applications to get the message text for a DCE RPC status code.

2.3.4 Management Services

The Management Services comprise the following two services:

- The local management service provides routines that are called only by an application to manage itself. For example, applications can get information related to the value of the binding communications timeout in a binding handle.
- The local/remote management service provides routines that are called either by an application managing itself, or by a remote application (the caller) wanting to manage the application (the callee). For example, the applications can get information from and update the local endpoint map database, and get RPC runtime statistics.

2.3.5 String Services

The String Services provide a string routine that client, server, and management applications use to free the allocated storage for a character string that was previously allocated by the RPC runtime.

2.3.6 UUID Services

The UUID Services provide UUID routines that allow client, server, and management applications to manipulate UUIDs. Routines include: creating new UUIDs, comparing two UUIDs, creating a hash value for a UUID, and converting UUIDs to strings and vice versa.

2.3.7 Authentication Services

Authentication Services provide routines that support authenticated communications between clients and servers. Authenticated RPC uses the authentication and authorization services supplied by the DCE Security Services.

2.3.8 Stub Support Services

Stub Support Services provide routines that permit applications to use the Stub Memory Management Scheme. Since a DCE full pointer can change its value across a call, DCE applications that use full pointers may need to allocate memory for pointed-to nodes, and stubs must be able to manage this memory.

2.4 Changing NCS 1.5.1 Routines to DCE RPC Routines

This section maps the existing NCS 1.5.1 routines to the routines that you would use in the DCE RPC API, if any equivalent exists. It presents the mapping for each NCS 1.5.1 API:

- **rpc_$** routines
- **rrpc_$** routines
- **socket_$** routines
- **lb_$** routines
- **uuid_$** routines
- **error_$** routines
- **pfm_$** routines

For a listing of the routines in the DCE RPC API, see Appendix A.

2.4.1 Mapping of NCS 1.5.1 rpc_$ Calls to DCE RPC Routines

In NCS 1.5.1, the **rpc_$** calls constitute the API to the RPC runtime library. Some of these calls are used only by clients, some only by servers, and some by either clients or servers.

This section lists the routines supported by NCS 1.5.1 and provides the equivalent routines available in the DCE RPC API where they exist.

2.4.1.1 Client Calls

In NCS 1.5.1, most of the **rpc_$** client routines allowed you to either create a handle or manage its binding state.

rpc_$alloc_handle — No replacement. NCS 1.5.1 allowed you to use this routine to get an unbound handle, and then use **rpc_$set_binding** to create a fully bound handle from this unbound handle. No DCE RPC API routines return an unbound handle, they return only fully bound or partially bound handles.

rpc_$set_binding — No replacement. See **rpc_$alloc_handle**.

rpc_$bind — Replaced by **rpc_binding_from_string_binding**, which returns a binding handle from a string representation of a binding handle, and **rpc_string_binding_compose**, which combines the components of a string binding (such as a protocol sequence and host ID) into a string binding handle.

rpc_$clear_server_binding — Replaced by **rpc_binding_reset**, which resets a binding handle to a partially bound handle, so that it identifies a specific host but no longer identifies a specific endpoint on that host.

rpc_$clear_binding — No replacement. The DCE RPC API does not support unbound handles; all remote procedure calls must have at least a partially bound handle.

rpc_$dup_handle — No analogous call in the DCE RPC API. That is, DCE RPC does not provide a routine to create a new *reference* to a single binding handle. It provides a routine, **rpc_binding_copy**, which creates a new *instance* of the binding handle.

In NCS 1.5.1 applications, multiple threads manipulating the *same* binding handle used **rpc_$dup_handle** to create a new reference to the binding handle and then **rpc_$free** to free each reference; **rpc_$free** did not free the binding handle until all references are freed. Since this call created a reference to the same handle, operations on the handle affected any copies.

If, in a DCE RPC-based multi-threaded application, a thread wants to maintain its own private binding handle, the

application can use **rpc_binding_copy** to create a new instance of the handle. Operations on the two handles work independently from each other. Applications use **rpc_binding_copy**, for example, so that a thread references a separate instance of the binding handle, thereby avoiding other concurrency issues.

rpc_$free_handle — Replaced by **rpc_binding_free**, which releases binding handle resources.

rpc_$set_async_ack — No replacement. This call is available for those NCS 1.5.1 platforms that could not support multi-tasking. There is no DCE RPC analog, since DCE RPC requires a platform that supports threads.

rpc_$set_short_timeout — Replaced by **rpc_mgmt_set_com_timeout**, which changes the communications timeout value in a server binding handle.

2.4.1.2 Server Calls

Most of the NCS 1.5.1 **rpc_$** server calls are used to initialize the server so that it has a socket (now an endpoint) on which to listen and is registered with the RPC runtime library on its host.

rpc_$use_family — Replaced by **rpc_server_use_protseq** (or by **rpc_server_use_all_protseqs**) which registers a specified protocol sequence (or sequences) with the RPC runtime. The RPC runtime creates a binding handle with a dynamically generated endpoint. In NCS 1.5.1 you call **rpc_$use_family** iteratively to listen on each available family. In DCE RPC, you can register all supported protocol sequences at once with the **rpc_use_all_protseqs** routine.

rpc_$use_family_wk — Replaced by **rpc_server_use_protseq_if** (or by **rpc_server_use_all_protseqs_if**) which tells the RPC runtime to use the specified protocol sequence combined with the specified endpoint information in the interface specification for receiving remote procedure calls.

rpc_$register	Replaced by **rpc_server_register_if**, which registers an interface and managers with the RPC runtime.
rpc_$register_mgr	Replaced by **rpc_server_register_if**, which registers an interface and managers with the RPC runtime.
rpc_$register_object	Replaced by **rpc_object_set_type**, which assigns the type of an object.
rpc_$unregister	Replaced by **rpc_server_unregister_if**, which unregisters an interface from the RPC runtime.
rpc_$listen	Replaced by **rpc_server_listen**, which tells the RPC runtime to listen for remote procedure calls.
rpc_$shutdown	Replaced by **rpc_mgmt_stop_server_listening**, which tells a server to stop listening for remote procedure calls.
rpc_$allow_remote_shutdown	Replaced by **rpc_mgmt_set_authorization_fn**, which allows servers to establish an authorization routine to control remote access to *all* the server's management routines.
rpc_$set_fault_mode	No replacement. NCS 1.5.1 allowed you to set this mode to help debug server applications. The DCE RPC API does not provide such a mechanism. You must run your application with a debugger to attempt to catch faults generated in server applications. For more details, see Section 5.8.

2.4.1.3 Calls for Clients or Servers

The following NCS 1.5.1 **rpc_$** calls are used by either clients or servers.

rpc_$inq_binding	Replaced by **rpc_binding_to_string_binding**, which returns the string representation of a binding handle.
rpc_$inq_object	Replaced by **rpc_binding_inq_object**, which returns the object UUID from a binding handle.
rpc_$name_to_sockaddr	Replaced by **rpc_binding_from_string_binding**, which returns a binding handle from a string representation of a binding handle.

rpc_$sockaddr_to_name Replaced by **rpc_binding_to_string_binding**, which returns the string representation of a binding handle.

2.4.2 Mapping of NCS 1.5.1 rrpc_$ Calls to DCE RPC Routines

In NCS 1.5.1, the **rrpc_$** calls enabled a client to request information about a server or to shut down a server. In DCE RPC, the **rpc_mgmt** routines allow client, server, and management applications to perform these tasks.

rrpc_$are_you_there Replaced by **rpc_mgmt_is_server_listening**, which checks whether a server is listening for remote procedure calls.

rrpc_$inq_stats Replaced by **rpc_mgmt_inq_stats**, which applications use to get statistics about a specified server.. Most of the server statistics reported by DCE RPC for the DCE RPC API are the same as in NCS 1.5.1. However, **rpc_mgmt_inq_stats** does not provide statistics about fragments as the NCS 1.5.1 **rrpc_$inq_stats** did.

rrpc_$inq_interfaces Replaced by **rpc_mgmt_inq_if_ids**, which applications use to get a listing of the interface IDs that a server has registered.

rrpc_$shutdown Replaced by **rpc_mgmt_stop_server_listening**, which directs a server to stop listening for remote procedure calls, if the server allows clients to issue this call.

2.4.3 Mapping of NCS 1.5.1 socket_$ Calls to DCE RPC Routines

Many NCS 1.5.1 **socket_$** routines are provided for manipulating socket addresses. The DCE RPC API uses string bindings to represent binding information, so no address manipulation is necessary. You can replace many NCS 1.5.1 **socket_$** routines with the DCE RPC routines that manipulate strings. (For more information on the difference between NCS 1.5.1 socket addresses and the DCE RPC string bindings, see Section 2.1.)

The following DCE routines manipulate strings:

rpc_binding_to_string_binding
Converts a client or server binding handle to its string representation.

rpc_binding_from_string_binding
Creates a binding handle from a string representation of a binding handle.

rpc_string_binding_parse Parses the binding handle for specific binding information.

rpc_string_binding_compose
Composes a binding handle with the specified string components.

For the remaining **socket_$** calls, the equivalent DCE RPC routines are as follows:

socket_$equal No replacement.This call allows you to compare separate components of the socket address. There is no analogous routine in the DCE RPC API. If you need to compare individual components of a DCE RPC string binding, use whatever library routines are available in the underlying supported network services. For example, use socket calls to get IP network address information.

socket_$to_name No replacement.

socket_$family_to_name No replacement. The DCE RPC API uses protocol sequences; they have no integer representation.

socket_$family_from_name No replacement. The protocol sequences supplied in the DCE RPC routines have no integer representation.

socket_$valid_family Replaced by **rpc_network_is_protseq_valid**, which determines if a specified protocol sequence can be used for making remote procedure calls. This routine actually provides more functionality than **socket_$valid_family**, which only checks whether the host's *operating system* supports the specified family.
rpc_network_is_protseq_valid tries to determine if the specific *host* supports the specified protocol sequence.

socket_$valid_families Replaced by **rpc_network_inq_protseq**, which lists the protocol sequences supported by both the RPC runtime and the operating system.

socket_$set_wk_port	No replacement. This call is used by the NCS 1.5.1 runtime only.
socket_$inq_my_netaddr	No replacement. See the explanation under **socket_$equal**.
socket_$inq_broad_addrs	No replacement. The DCE RPC API does not support broadcasts.
socket_$max_pkt_size	No replacement. This call is used by the NCS 1.5.1 runtime only.
socket_$to_local_rep	No replacement. This call is not used on typical NCS 1.5.1 platforms.
socket_$from_local_rep	No replacement. This call is not used on typical NCS 1.5.1 platforms.

2.4.4 Mapping of NCS 1.5.1 lb_$ Calls to DCE RPC Routines

The NCS 1.5.1 API provides the **lb_$** calls as the interface to the Location Broker services. These calls allow clients and servers to look up, register, or unregister entries in Local or Global Location Broker databases. The **lb_$** calls handle registration of both local (local-host-only) and global (network-wide) entries.

The DCE RPC API provides an interface to the endpoint map database, which resides on each host. DCE RPC also includes the DCE Name Service database, which allows applications to do global lookup operations.

Note that while NCS 1.5.1 Local Location Broker and DCE RPC daemons perform similar functions, the APIs for the corresponding routines (**lb_$** and **rpc_ep**) are not equivalent. The **lb_$** calls support both local and global location services, while **rpc_ep** calls support only the *local* endpoint map database on a local host. To handle *global* location services, the DCE RPC defines a separate Name Services API, the **rpc_ns** routines.

lb_$lookup_object	Replaced by **rpc_ns_entry_object_inq_next** routine. Before using this routine, the application must set up the search context with **rpc_ns_entry_object_inq_begin**, and after viewing, the application must remove the context with **rpc_ns_entry_object_inq_done**.

lb_$lookup_type	No replacement.
lb_$lookup_interface	For lookups in the local endpoint map, replaced by **rpc_ep_resolve_binding**. For lookups in the global database, replaced by the **rpc_ns_binding_lookup_next** routine. Before using **rpc_ns_binding_lookup_next**, the application must set up the search context with **rpc_ns_binding_lookup_begin**, and after viewing, it must remove the context with **rpc_ns_binding_lookup_done**.
lb_$lookup_object_local	Replaced by the **rpc_mgmt_ep_elt_inq_next** routine, which returns an element from the local or remote endpoint map database. Before using this routine, the application must set up the search context with **rpc_mgmt_ep_elt_inq_begin** and after viewing, it must remove the context with **rpc_mgmt_ep_elt_inq_done**. The **rpc_ep_resolve_binding** routine can also provide this functionality on the local host.
lb_$lookup_range	No replacement.
lb_$register	Servers can publicly offer an interface (and object UUIDs of resources it offers) with the name-service database for use by any client application by exporting the interface and objects with **rpc_ns_binding_export**. These interfaces are also registered in the local endpoint map database. If a server created a binding with **rpc_server_use_all_protseqs** or **rpc_server_use_protseq**, it must register with the local endpoint map database using either **rpc_ep_register** or **rpc_ep_register_no_replace**.
lb_$unregister	Servers use **rpc_ns_binding_unexport** to remove the binding handles for an interface and/or object from a name-service database entry. Servers use **rpc_ep_unregister** to unregister their own entries from the local host's endpoint map database. Management routines call **rpc_mgmt_ep_unregister** to remove an interface ID, if a server is no longer available, or to remove object UUIDs if a server no longer supports the objects.

2.4.5 Mapping of NCS 1.5.1 uuid_$ Calls to DCE RPC Routines

The **uuid_$** calls generate and manipulate Universal Unique Identifiers.

uuid_$gen	Replaced by **uuid_create**, which creates a new UUID.
uuid_$decode	Replaced by **uuid_from_string**, which converts a string UUID (as generated by the **uuidgen** program) to a binary UUID.
uuid_$encode	Replaced by **uuid_to_string**, which converts a binary UUID to its character-string representation.
uuid_$from_uid	No replacement. This routine is useful for Apollo systems only.
uuid_$to_uid	No replacement. This routine is useful for Apollo systems only.
uuid_$equal	Replaced by **uuid_equal**, **uuid_compare**, and **uuid_is_nil**. Use **uuid_equal** to determine if two UUIDs are equal. Use **uuid_compare** to determine the lexical order of two UUIDs, and use **uuid_is_nil** to determine if a UUID has a nil value. (NCS 1.5.1 combined all this functionality in one call, **uuid_$equal**.)
uuid_$hash	This routine (which was undocumented) is replaced by the routine **uuid_hash**, which generates a hash value for a specified UUID.
uuid_$nil	This global variable is replaced by the routine **uuid_create_nil**, which creates a nil-valued UUID.

2.4.6 Mapping of NCS 1.5.1 error_$ Calls to DCE RPC Routines

Most of the NCS and DCE RPC calls indicate their completion status via status codes. In NCS 1.5.1, the **error_$** calls converted these status codes into textual error messages. In the DCE RPC API, the **dce_error_inq_text** call performs this task. Note that the **error_$** calls returned more information for Domain/OS systems. The **dce_error_inq_text** routine returns information specific to RPC applications only.

error_$c_get_text	Replaced by **dce_error_inq_text.**
error_$c_text	Replaced by **dce_error_inq_text.**

2.4.7 Replacement of the pfm_$ Calls with the DCE Exception-Returning Package

The DCE RPC API does not use **pfm_$** routines for handling UNIX signals. Instead, it uses the DCE exception-handling package, which accompanies the threads implementation available with the underlying operating system.

For more information on the DCE exception-handling package, see Chapter 5.

2.5 Mapping of NCS 1.5.1 Data Types to DCE RPC Data Types

This section supplies equivalent DCE RPC data types for NCS 1.5.1 data types, if they exist. For details on the DCE RPC data types, see the **intro_rpc**(3) manual page of the *OSF DCE Application Development Reference.*

2.5.1 error_$ and status_$t Data Types

Each DCE RPC routine returns a status code indicating whether the routine completed successfully. A return value of **rpc_s_ok** indicates success. The status code argument has a data type of **unsigned32.** To translate a status code to a text message, call the **dce_error_inq_text** routine.

The DCE RPC API does not support the **error_$t** and **status_$t** data types.

2.5.2 lb_$ Data Types

NCS 1.5.1 **lb_$** data types are used for both global location and local location. DCE RPC supplies two separate interfaces, the Name Service Interface for global naming services and the Endpoint Map Services for local forwarding services.

See the *OSF DCE Application Development Reference* for information on the name service handle that contains information that the RPC runtime uses to return data from the name service database. The routines requiring a name service handle show an argument data type of **rpc_ns_handle_t.**

The endpoint map database contains binding handles. Routines requiring a binding handle as an argument show a data type of **rpc_binding_handle_t**.

The DCE RPC API does not support the **lb_$** data types **lb_$entry_t**, **lb_$lookup_handle_t**, and **lb_$server_flag_t**.

2.5.3 pfm_$ Data Types

The DCE RPC API does not support the **pfm_$cleanup_rec** routine for handling faults and exceptions. It supports the DCE exception-handling package described in Chapter 5 .

2.5.4 rpc_$ Data Types

The following lists the DCE equivalents of the NCS 1.5.1 **rpc_$** data types.

handle_t	Routines requiring a binding handle show a data type of **rpc_binding_handle_t**.
rpc_$epv_t	The NCS RPC entry point vector is replaced by the interface handle, which is of type **rpc_if_handle_t**. The IDL compiler automatically creates an interface specification data structure from each IDL file and creates a global variable of type **rpc_if_handle_t** for the interface specification.
rpc_$generic_evt_t	This is the DCE RPC data type **rpc_if_handle_t**.
rpc_$if_spec_t	This is the DCE RPC data type **rpc_if_handle_t**.
rpc_$mgr_epv_t	This is the DCE RPC data type **rpc_mgr_epv_t**.
rpc_$shut_check_fn_t	This is the DCE RPC data type **rpc_mgmt_authorization_fn_t**. (See the **rpc_mgmt_set_authorization_fn**(3) manual page for more information.)

2.5.5 rrpc_$ Data Types

The following lists the DCE equivalents of the NCS 1.5.1 **rrpc_$** data types.

rrpc_$interface_vec_t	This is the DCE RPC interface identification vector of type **rpc_if_id_vector_t**.
rrpc_$stat_vec_t	This is the DCE RPC statistics vector of type **rpc_stats_vector_t**.

2.5.6 socket_$ Data Types

The DCE RPC API does not allow direct manipulation of the socket address so there are no equivalent data types for the NCS 1.5.1 **socket_$** types. The binary representation for all addressing information is contained in the RPC binding handle of type **rpc_binding_handle_t**.

2.5.7 uuid_$ Data Types

The NCS 1.5.1 **uuid_$t** data type is replaced by the DCE **uuid_t** data type, which is the same size but has a different internal structure. Among other differences, one field that was reserved (all zeros) in NCS 1.5.1 UUIDs is not reserved in DCE UUIDs. Although they differ internally, **uuid_$t** and **uuid_t** have the same NDR representation, which means that the network representations are exactly the same. (This is referred to as "wire interoperability.")

However, since the internal structure is different, the C initializations generated by the **uuidgen** and **uuid_gen** tools are slightly different. For more details, see Section 5.9.

The global variable **uuid_$nil** is replaced by a DCE RPC routine, **uuid_create_nil**, which creates a nil-valued UUID.

The NCS 1.5.1 **uuid_$string_t** data type has no replacement. In the DCE RPC API, use the **unsigned_char_t** type to initialize a string or pass an item of type **unsigned_char_t *** to an RPC runtime routine that allocates a string.

The UUID string representation has a new format in the DCE RPC API. The NCS 1.5.1 UUID string has 28 digits and 8 periods; the reserved field (which is all zeros) is not represented:

```
54c2c718f000.0d.00.01.e1.e9.00.00.00
```

The DCE RPC UUID string format has 32 digits and 4 dashes; all fields are represented:

```
847ec9f0-9203-11ca-8e74-08001e01e1e9
```

The DCE RPC runtime routines that accept UUID strings accept either format. However, a DCE RPC UUID cannot be correctly represented by an NCS 1.5.1 UUID string, so conversion of a DCE RPC UUID to and from an NCS 1.5.1 UUID string will lose information.

2.6 How to Migrate NCS 1.5.1 Applications to DCE RPC

The easiest way to convert an application to DCE RPC is to convert both the client and server programs. However, it is also possible to convert only the server or client portion of an application, as long as you write the code required to maintain interoperability between the programs.

2.6.1 Complete Conversion

You can choose to convert an entire application (that is, both the client and server portions) to DCE RPC and then bring up all the clients and servers on all hosts at the same time.

The first step in the conversion process is to define the interfaces and create the IDL stub files. To do this, use the **nidl_to_idl** translator to convert your NIDL files to IDL syntax, and then edit the resulting files, as described in Chapter 4. Note that when you use **nidl_to_idl**, the resulting interface definition contains attributes provided for compatibility (**v1_array**, **v1_string**, **v1_struct**, and **v1_enum**). However, if you are converting an entire application to DCE RPC (that is, all clients and all servers) and do not require interoperability with NCS 1.5.1-based programs, you should create DCE RPC interface definitions that do *not* contain these attributes. These attributes may not be supported in future releases of DCE.

After you have written the IDL files, convert the client and server programs to use the DCE RPC API, as described in Chapter 5.

2.6.2 Partial Conversion

You can choose to convert only the client or server portion of an application to DCE RPC, so that one program uses the NCS 1.5.1 API and another program uses the DCE RPC API. For example, you can write a DCE RPC server that can support both old NCS 1.5.1 and new DCE

RPC clients, as long as you do not mix routines from the NCS 1.5.1 and DCE RPC API in the same program. This type of conversion is more complicated than converting an entire application because you need to consider how the NCS 1.5.1 and DCE RPC programs interoperate. If part of your application must run on platforms that do not support DCE (for example, if your clients must run on Domain/OS), partial conversion may be your only alternative.

Note that if an application uses the NCS 1.5.1 Global Location Broker (GLB), you cannot convert only part of the application to DCE RPC. For example, you cannot convert a server to DCE RPC and expect the server to provide service to NCS 1.5.1-based clients that continue to use the GLB.

If you convert only one part of an application to DCE RPC, remember that interoperability between NCS programs is determined by the interface definition (**.idl** file). As long as the network representation of the **.idl** file is the same in NCS 1.5.1 and DCE RPC, the programs will interoperate. However, if you are writing a new DCE RPC interface that must service NCS 1.5.1 servers or clients, you must *not* use DCE RPC features that are not supported by NCS such as full pointers, pipes, or context handles, because NCS 1.5.1 programs will not be able to recognize such features. Chapter 3 provides more details on these DCE RPC IDL features.

In addition, for any data type whose network data representation has changed between NCS 1.5.1 and DCE RPC, your interface definition must declare that you want the NCS 1.5.1 network data representation of this type. The attributes that specify NCS 1.5.1 network data representation are **v1_array**, **v1_string**, **v1_struct**, and **v1_enum**.

You can designate that you want the NCS 1.5.1 network data representation by adding the **v1_** attribute to the type declarations. However, the simplest way to do this is to use the tool **nidl_to_idl**, which makes the necessary changes automatically. Chapter 4 describes when and how to use the **nidl_to_idl** translation tool.

Chapter 3

Overview of the DCE Interface Definition Language

This chapter provides an overview of the differences between the DCE Interface Definition Language (IDL) and the NCS 1.5.1 Network Interface Definition Language (NIDL) and introduces the new features available in IDL. Chapter 4 provides examples and details on the process of writing interface definitions.

This chapter is intended to highlight the differences between NCS 1.5.1 NIDL and DCE IDL. For a complete description of DCE IDL syntax, see the *OSF DCE Application Development Guide.*

3.1 An Overview of DCE Interface Definitions

In RPC applications, a network interface is a set of related procedures that can be called across a network. An interface definition written in the IDL language defines the signatures for each operation in the interface that can be called via the RPC facility.

An IDL declaration resembles a C header file, except that it contains extra information needed by the remote procedure call mechanism.

NOTE: Some versions of NCS 1.5.1 support a Pascal-like syntax for NIDL in addition to the C-like syntax. DCE IDL supports a C-like syntax only.

Some of the interface definition information that is defined in an NCS 1.5.1 interface definition is defined in a DCE attribute configuration file (ACF or **.acf** file). An attribute configuration file is a secondary (and optional) file for declaring attributes that apply to the relationship between the stubs and the application code. In contrast, the interface definition file is for declaring any information that applies to the contract between the client and server.

The following sections provide more information on the changes in DCE IDL:

- DCE IDL data types, constructed types and attributes
- Structure of the DCE interface definition
- Interface names
- Interface attributes
- Import declarations
- Constant declarations
- Type declarations
- Operation declarations
- Parameter declarations
- The attribute configuration file
- DCE IDL output files

3.2 DCE IDL Data Types and Attributes

DCE IDL supports more data types and attributes than NCS 1.5.1 NIDL. You use these types in type declarations and as parameters in operation declarations (which are described later in this chapter). The following subsections provide more detail about these data types. Section 3.2.1 describes the simple data types; the remaining sections describe the constructed types.

Note that if you use the new DCE data types and attributes in an interface definition file, the resulting **.idl** file will not be interoperable with an NCS 1.5.1 version of the interface definition. For details on this, see Section 2.6.

3.2.1 Base (Simple) Types

The following lists the IDL base types and compares them to the NIDL simple types.

Integer types	IDL support is same as NIDL.
Floating-point types	IDL support is same as NIDL.
The character (**char**) type	IDL support is same as NIDL.
The boolean type	IDL support is same as NIDL.
The byte type	IDL support is same as NIDL.
The void type	Different IDL support. In both NCS 1.5.1 and DCE RPC, the **void** type is used to specify the type of an operation that does not return a value. In DCE RPC, **void** is also used to specify the type of a context handle parameter or null pointer constant, both of which must be declared **void ***.
The handle type	This **handle_t** is the same as the NIDL **handle_t**.
The **error_status_t** type	This is a new predefined data type to hold RPC communications status information. It is analogous to the NCS 1.5.1 type **status_$t**.

3.2.2 Pipes and Pipe Attributes

IDL supports pipes as a mechanism for transferring large quantities of typed data; an IDL **pipe** is an open-ended sequence of elements of one type. IDL recognizes three kinds of pipes: an **in** pipe for transferring data from a client to a server, an **out** pipe for transferring data from a server to a client, and an **in,out** pipe for two-way data transfer between a client and server. When using pipes, you need to write a set of push and pull routines that are called by the client stub.

3.2.3 Pointers

IDL provides enhanced support for pointers. It supports both reference and "full" pointers.

A **reference pointer**, designated by the **ref** attribute, provides the simplest form of pointer semantics and incurs the least amount of overhead during remote procedure calls. A reference pointer value never changes during a remote procedure call, it never has a NULL value, and it cannot be an alias. That is, a reference pointer cannot point to a storage area that is pointed to by any other pointer used as a parameter of the same operation.

A **full pointer**, designated by the **ptr** attribute, provides "full" pointer semantics but incurs the most overhead during remote procedure calls. A full pointer value can change across a remote procedure call, its value can be NULL, and it can be an alias. That is, a full pointer can point to a storage area that is pointed to by any other full pointer used in a parameter of the same operation. If two pointers are aliases of one another, they must both point to the same range of storage. They cannot point to differing parts of a structure or to overlapping storage areas.

IDL allows you to apply array attributes to pointer parameters and fields. It supports the C idiom:

T *p;

where pointer **p** points to the first element of an array of **T**'s.

The **ignore** attribute can also be applied to pointers to prevent the pointer from being transmitted in a remote procedure call. Using the **ignore** attribute prevents marshalling code from dereferencing a pointer; the remote instance has an undefined value. The **ignore** attribute can be used to handle problems with aliases.

3.2.4 Arrays and Array Attributes

IDL supports three types of arrays: fixed, conformant, and varying.

A **fixed** array is an array whose size is defined in the interface definition and whose storage is allocated at compile time.

A **conformant** array (called an **open** array in NCS 1.5.1) is an array whose size is determined at run time. The size is specified by the value of the parameter referenced in a **size_is** attribute (an element count). The **size_is** attribute is analogous to the NCS 1.5.1 **max_is** attribute, but is an element count, rather than an index. Although DCE supports the **max_is** attribute for compatibility reasons, it is usually easier to use the **size_is** attribute.

The array's type definition determines whether it is fixed or conformant:

```
typedef long fixed_array[13];
typedef long conf_array[]; /*Empty brackets imply a conformant array */
```

The lower bound of an array must be 0.

A **varying** array is an array *instance* whose "interesting part" (the part that must be marshalled) is determined at run time. The size of the marshalled part is determined by values named in one or more of the following data limit attributes: **length_is** (an element count), **last_is** (the maximum index value), and **first_is** (the minimum index value).

Note that a varying array applies to a particular array instance, not the array type; so a varying array can be either a fixed or conformant array.

A varying array is determined by the attributes applied to an array in an operation declaration:

```
op (..., [in, length_is (len)] conf_array c, ...[in] long (len), ...);
```

A DCE varying array differs slightly from an NCS 1.5.1 varying array. In NCS 1.5.1, you can declare a varying conformant array with only a **last_is** attribute and the NIDL compiler assumes that the conformant array size is the same size as the amount you want marshalled (as specified in the **last_is** attribute). In DCE RPC, when you use a varying conformant array, you must specify both its size (with **size_is** or **max_is**) and the amount you want marshalled (with **length_is** or **last_is**).

3.2.5 Strings

IDL supports strings differently from NIDL. NIDL supports the **string0** data type. IDL supports a **string** attribute that you apply to an array.

While both the NIDL **string0** type and the IDL **string** attribute designate a null-terminated character array, they differ in how they are specified as the following examples illustrate.

In C syntax, a character string "s" is specified as follows:

```
char s[8]
```

In NIDL syntax, a character string "s" is specified as follows:

```
string0[8] s
```

In IDL syntax, a character string "s" is specified as follows:

```
[string] char s[8]
```

NIDL and IDL strings also differ in that the NIDL **string0** is always a fixed array of characters, but the IDL **string** attribute can be applied to a broader range of types. An IDL string can be a varying array of characters, bytes, or structures whose members are bytes.

3.2.6 Customized Handles

The **handle** attribute is used to specify a customized handle, that is, a user-defined binding handle of a type that is other than the primitive binding handle type. This enables application developers to manage additional binding information that cannot be handled by the primitive binding handle.

The **handle** attribute exists in NCS 1.5.1 as well and is used to create a generic handle. The DCE customized handle is analogous to the NCS 1.5.1 **generic handle**.

To use a customized handle, you must declare the handle in the interface definition and add application code that the client stub calls to get a primitive handle from a customized handle and to release any resources used for the customized handle.

3.2.7 Context Handles

A context handle, designated by the **context_handle** attribute, is an IDL feature that is useful for distributed applications in which manager code maintains state information for a client across several remote procedure calls. The client passes the context handle, which points to the state information, as an input or input/output parameter in each call.

Applications that use context handles may have to supply a rundown procedure in the server to perform clean-up operations when the client context is no longer needed. A type declaration for the rundown procedure is declared in the interface definition; this ensures that the stub knows about the procedure in the server. Without performing such clean-up operations, the server could eventually consume all the available disk space maintaining information about non-existent clients. The use of context handles and clean-up operations helps prevent such memory leaks.

A context handle is specified with the **context_handle** attribute on a parameter of type **void ***, or on a type that is defined as **void ***.

3.2.8 Enumerations

IDL supports one **enum** type for providing names for integers. An enumeration can have up to 32767 identifiers. In contrast, NIDL supports a 32-bit **enum** and 16-bit **short enum**.

3.2.9 Unions

IDL and NIDL both support the **union** type. A **union** may hold (at different times) data of different types and sizes, analogous to a Pascal variant record. The IDL syntax for declaring a **union** is similar to the NIDL syntax. However, IDL allows the specification of names that the IDL compiler uses to construct identifiers in the generated C code.

3.2.10 Structures

NIDL and IDL both support the **struct** type, although the IDL **struct** type has some additional features. An IDL **struct** can contain pointer members, which allows DCE RPC applications to pass parameters such as linked lists and trees. In contrast, a NIDL **struct** cannot contain a pointer unless you apply the **transmit_as** type attribute and supply routines to convert the structure to a transmissible type.

In addition, an IDL **struct** has an **ignore** attribute, which allows you to specify that the data pointed to by a pointer member should not be transmitted in a remote procedure call. The **ignore** attribute can increase efficiency at times when a **struct** is being passed as an argument, but the remote procedure call does not need to access the data indicated by a pointer member.

3.2.11 Attributes for Compatibility with NCS 1.5.1

These attributes (**v1_array**, **v1_enum**, **v1_string**, and **v1_struct**) are provided for transitional purposes only, to provide compatibility between NCS 1.5.1 and DCE programs. The **nidl_to_idl** tool adds the appropriate attribute to the declaration of any type constructor whose DCE network data representation differs from the NCS 1.5.1 representation. These attributes should not be used in new DCE interface definitions and they may not be supported in future DCE releases. For more details on these attributes, see Section 4.1.1. See also Sections 2.6 and C.1.11.

3.2.12 Sets

NIDL supports sets (**bitset enum** and **short bitset enum**) which allow you to define names for bits in a single integer. DCE provides no analogous data types.

3.3 Structure of the DCE IDL Interface Definition

The structure of the IDL interface definition is similar to that of the NIDL interface definition except that IDL supports only a C-like syntax, so it neither requires nor accepts the NCS 1.5.1 syntax identifier (**%c** or **%pascal**) in the heading of an interface definition.

The IDL interface definition has the following components:

Interface Heading	Consists of a list of *interface_attributes* enclosed in brackets, followed by the keyword **interface**, followed by the name of the interface.
Import Declarations	Specifies the names of other IDL interfaces that define types and constants used by this interface. Import declarations must precede all other declarations.
Constant Declarations	Specifies the constants that the interface exports.
Type Declarations	Specifies the types that the interface exports.
Operation Declarations	Specifies the operations that the interface exports.

The list of declarations is enclosed in braces; each declaration is terminated with a semicolon. Constant, type, and operation declarations can appear in any order (provided that any constant or type is defined before it is used).

Subsequent sections of this chapter provide more details on each component.

3.4 Interface Names

By convention, the name of an interface definition file is the same as the name of the interface it defines, along with the suffix **.idl**. For example, the definition for a **bank** interface would reside in a **bank.idl** interface definition file. Note that the interface name is not necessarily the base name of the file that contains the interface definition. Thus, the interface name is not limited by the filename restrictions of any operating system.

The IDL compiler incorporates the interface name in identifiers it constructs for various data structures and data types, so the length of an interface name can be at most 17 characters. Most

IDL identifiers have a maximum length of 31 characters, and the IDL compiler issues a warning if an identifier exceeds that length, but does not truncate it. (The NIDL compiler did not issue a warning.)

3.5 Interface Definition Attributes

Interface attributes appear within brackets in the header of the interface definition, as in NCS 1.5.1. IDL supports more interface attributes than NIDL. Table 3–1 lists the IDL interface definition attributes and compares them to the attributes available in NCS 1.5.1 NIDL.

Table 3–1. Comparison of IDL and NIDL Interface Attributes

IDL Attribute	NIDL Attribute	Where Used
uuid	**uuid**	Interface Definition Headers
version	**version**	
endpoint	**port**	
local	**local**	
pointer_default	Not Supported	
ACF Attribute in DCE	**implicit_handle**	

The IDL interface attributes are as follows:

- The **uuid** attribute specifies the Universal Unique Identifier (UUID) assigned to the interface. It serves the same purpose as in NCS 1.5.1 and is automatically generated by the **uuidgen** utility. The format of the DCE UUID string, however, is different.

- The **version** attribute supports major and minor version numbers (NIDL supports only a single integer version number).

  Version numbers are used to control interoperability between clients and servers. Clients and servers must use an interface with the same major version number, and the minor version number of the client must be less than or equal to the minor version number of the server. You must increase the major version number when making any incompatible changes to an interface definition.

  Interface versions are considered compatible if they satisfy these conditions:

  – You add new operations to an interface only *after* all existing operation declarations.

- You add new type and constant declarations that are used only by operations added at the same time or later.

You cannot change the signatures of existing operation declarations.

- The DCE **endpoint** attribute replaces the NCS 1.5.1 **port** attribute as the way to specify a well-known endpoint.

  The following example, excerpted from the **ep.idl** interface definition, specifies the well-known endpoints on which **rpcd** listens:

```
endpoint ("ncadg_ip_udp:[135]", "ncadg_dds:[12]",
    "ncacn_ip_tcp:[135]", "ncacn_dnet_nsp:[#69]")
```

  IDL recognizes and accepts more protocol sequences than are typically supported by any single implementation of the DCE runtime software, so that one IDL definition can be used on several DCE platforms supporting different sets of protocol sequences.

  The runtime software in the HP DCE Developers' Environment supports two protocol sequences, **ncadg_ip_udp** and **ncacn_ip_tcp**.

- The **local** attribute indicates that the interface definition does not declare any remote operations and the IDL compiler should generate only header files, not stubs. This is the same as the NCS 1.5.1 **local** attribute.

- The **pointer_default** attribute is available because DCE RPC handles both reference and full pointers. This attribute allows you to specify a default behavior of the pointers in the interface definition that are not specifically labeled with the **ptr** or **ref** attribute. This default behavior is overridden when you apply either attribute to a specific pointer declaration.

  The **pointer_default** attribute affects pointers that appear in the declaration of a structure or union member or that are at any other level besides the "top level" of an operation parameter declared with more than one pointer operator. Note that **pointer_default** does *not* apply to a pointer in the return value of an operation, since this return value is always a full pointer.

- The **implicit_handle** attribute is not part of the DCE interface definition language syntax. It is part of the DCE attribute configuration language syntax.

3.6 Import Declarations

You specify import declarations in DCE RPC in the same way as you do in NCS 1.5.1. The IDL **import** declaration specifies another interface definition whose types and constants are used by the importing interface.

DCE **import** declarations work in the same way as in NCS 1.5.1; the resulting **#include** directives (in the C code generated by IDL) enclose file names in double quotation marks (" ").

By default, the IDL compiler resolves relative pathnames of imported files by looking first in the current working directory and then in the system IDL directory. Use the **–I** IDL compiler option (which is the same as the **–idir** option in NCS 1.5.1) to specify a directory from which the compiler will resolve the pathnames of imported files. Using **–I** lets you avoid putting absolute pathnames in your interface definitions.

3.7 Constant Declarations

The DCE IDL constant declaration, which provides a way for getting **#define** statements into derived **.h** files, supports more types than NCS 1.5.1 does. In NCS 1.5.1, the constant declaration, **const**, supports only integer and character constants. DCE supports integer, boolean, character, string, and null pointer constants, and constant expressions. In addition, DCE permits declaring string constants as **char ***.

The following are examples of DCE IDL **const** declarations:

```
const unsigned short int SCORE = 20;
const unsigned short int FOUR_SCORE_AND_SEVEN = 4*SCORE+7;
const boolean VRAI = TRUE;
const char H = 'h';
const char *JSB = "Johann Sebastian Bach"
const void *NULLSVILLE = NULL;
```

3.8 Type Declarations

The IDL syntax for the type declaration, **typedef**, is the same as the NIDL type declaration.

3.8.1 Type Attributes

NIDL supports only two type attributes: **handle** and **transmit_as**. IDL supports the following type attributes:

handle	The type being declared is a user-defined, customized handle type. (See Section 3.2.6.)
context_handle	The type being declared is a context handle type. (See Section 3.2.7.)
transmit_as	The type being declared is a "presented type" which, when passed in remote procedure calls, is converted to a specified "transmitted type." IDL, like NIDL, supports the **transmit_as** attribute to pass complex data types for which the IDL compiler cannot generate marshalling and unmarshalling code, and to pass data more efficiently.
ref	The type being declared is a reference pointer; it cannot be null and cannot be an alias.
ptr	The type being declared is a full pointer; it can be null and can be an alias.
string	The array type being declared is a string type.

IDL also supports the type declarations **v1_array**, **v1_string**, **v1_struct**, and **v1_enum** for compatibility with NCS 1.5.1 applications. These types are not intended to be used in new applications. For details, see Section 2.6 and for an example, see Section 4.1.3.

3.8.2 Type Specifiers

IDL supports different type specifiers from NIDL, as shown in Table 3–2. The IDL base specifiers are similar to NIDL's simple types: integers, floating-point numbers, characters, booleans, a **byte** type, a **void** type, and a primitive handle type, **handle_t**.

IDL does not support the NIDL constructed types **bitset**, **short bitset**, **short enum**, and **string0**. IDL additionally supports **pipe**.

The IDL compiler imports **nbase.idl**, which predefines some types as shown in Table 3–2. The type macros emitted by the IDL compiler are the same as the NIDL macros, except that **ndr_$** is replaced by **idl_** (for example, **ndr_$char** is now **idl_char**).

Table 3–2. DCE Type Specifiers

IDL Type Specifier			Size	Type Macro
sign	size	type		emitted by IDL
	small	int	8 bits	idl_small_int
	short	int	16 bits	idl_short_int
	long	int	32 bits	idl_long_int
	hyper	int	64 bits	idl_hyper_int
unsigned	small	int	8 bits	idl_usmall_int
unsigned	short	int	16 bits	idl_ushort_int
unsigned	long	int	32 bits	idl_ulong_int
unsigned	hyper	int	64 bits	idl_uhyper_int
		float	32 bits	idl_short_float
		double	64 bits	idl_long_float
		char	8 bits	idl_char
		boolean	8 bits	idl_boolean
		byte	8 bits	idl_byte
		void	—	idl_void_p_t
		handle_t	—	—

3.8.3 Type Declarators

An IDL type declarator can be either a simple declarator or a complex declarator. A **simple declarator** is just an identifier. A **complex declarator** is an identifier that specifies an array, a function pointer, or a pointer. (A function pointer can only be used in interface definitions with the **local** attribute; the IDL compiler generates an error if you try to use a function pointer in remote procedure calls.)

3.9 Operation Declarations

In DCE RPC, parameter attributes precede the type specifier rather than the declarator.

For example, an NCS 1.5.1 NIDL operation might be declared as follows:

```
void ms_show(
  ms_string_t [in] to_name,
  ms_string_t [in] from_name,
  ms_string_t [in] message
);
```

A similar DCE IDL operation is declared as follows:

```
void ms_show(
  [in] ms_string_t to_name,
  [in] ms_string_t from_name,
  [in] ms_string_t message
);
```

Table 3–3 lists the attributes associated with operation declarations.

Table 3–3. Comparison of IDL and NIDL Operation Attributes

IDL Attribute	NIDL Attribute	Where Used
broadcast	**broadcast**	Operations
maybe	**maybe**	
idempotent	**idempotent**	
ptr	Not Supported	
context_handle	Not Supported	
string	Not Supported	
ACF Attribute in DCE	**comm_status**	

IDL supports different operation attributes from NIDL. Both IDL and NIDL support **idempotent**, **broadcast**, and **maybe**. IDL does not support **comm_status**. (**comm_status** is declared in the attribute configuration file.)

Note that the **broadcast** operation attribute is the only way that a DCE-based program can implement broadcasting; broadcasting is not supported by the RPC API. We recommend that you broadcast operations sparingly.

3.10 Parameter Declarations

IDL syntax for declaring parameters of an operation is the same as in NIDL.

IDL supports different attributes from NIDL. While both support **in** and **out** parameters, IDL does not support **comm_status**. **comm_status** is declared in the DCE attribute configuration file (ACF). IDL supports the following attributes that NIDL does not support:

ref	The parameter is a reference pointer; it cannot be null and cannot be an alias.
ptr	The parameter is a full pointer; it can be null and can be an alias.
string	The parameter is a string.
context_handle	The parameter is a context handle.

IDL and NIDL both support array attributes that specify the characteristics of arrays. NIDL and IDL both support the **max_is** and **last_is** array attributes. In addition, IDL supports **size_is**, **first_is**, and **length_is** array attributes.

IDL also supports the parameter declarations **v1_array**, **v1_string**, **v1_struct**, and **v1_enum** for DCE migration. These declarations are not intended for use in new applications. For details, see Section 2.6.

3.11 The Attribute Configuration File

While all attributes relating to an interface are in one file in NCS 1.5.1, DCE RPC interface definitions are designed so that the attributes are in two files. The interface definition contains any attributes that affect the interoperability between a client and server. The attribute configuration file (ACF) is a secondary (and optional) file that contains attributes that apply to the relationship between the generated stub code and the local application code. The ACF

allows application programmers to customize their applications more easily without affecting the interoperability of existing interfaces. Application programs built from the same interface definition are guaranteed to interoperate; differences between ACFs are guaranteed to not affect interoperability.

Client and server sides of an interface *must* be built using the same interface definition (**.idl**) file, but they can be built using different **.acf** files. Clients may use **.acf** files, for example, to omit unnecessary operations from the client stub (thereby reducing the size of the client stub), to represent a network data type as a local data type, to define how a client establishes a binding, or to specify how arguments of nonscalar data types are marshalled and unmarshalled.

When you invoke the IDL compiler, specifying an interface definition, the compiler automatically checks for a related attribute configuration file. It looks for a file with the same name as the interface definition, with the suffix **.acf** instead of **.idl**. The compiler searches the current directory and then any directories specified via **–I** options on the **idl** command line. If an ACF exists, it's compiled with the IDL file.

The attributes that you can declare in an ACF are as follows. Subsequent sections describe these attributes in greater detail:

- **auto_handle, explicit_handle, implicit_handle**
- **comm_status, fault_status**
- **code, nocode**
- **in_line, out_of_line**
- **represent_as**
- **enable_allocate**
- **heap**

The attribute configuration file also supports the **include** statement. The **include** statement specifies any additional header files that you want included in the generated stub code. Use **include** whenever you use the **represent_as** or **implicit_handle** attributes and the specified type is not defined or imported in the IDL files.

3.11.1 Binding Attributes: auto_handle, explicit_handle, implicit_handle

The **auto_handle**, **explicit_handle**, and **implicit_handle** attributes relate to the methods a DCE application uses to manage bindings before making remote procedure calls. These attributes allow the application to define at compile time which method of binding to use for the entire interface.

The **auto_handle** attribute causes the client stub and the RPC runtime to manage the binding to the server by using a name service. Any operation in the interface that has no parameter containing binding information is bound automatically by client stub code to a server so that the client application code does not have to specify the server on which an operation executes.

The **explicit_handle** attribute allows the application program to manage the binding to the server. This attribute indicates that a binding handle is passed to the runtime as an operation parameter.

The **implicit_handle** attribute allows the application program to manage the binding to the server. You specify the data type and name of a handle variable as part of the **implicit_handle** attribute. This attribute informs the compiler of the name and type of the global variable through which the binding handle is implicitly passed to the client stub. The client stub code declares a variable of the type and name specified in the **implicit_handle** attribute; the application must initialize the variable before making a call to the interface.

If an ACF file defines the **auto_handle** binding attribute for an interface, an individual operation can still use an explicit binding by declaring a binding handle or a context handle parameter for the operation. The following is an example of an operation declaration with an explicit handle as the first parameter of the operation declaration:

```
void expl_op(
    [in] handle_t h,
    [in] long a,
    [out] long *b);
```

3.11.2 Handling Errors with comm_status and fault_status

The IDL **comm_status** and **fault_status** attributes cause the status code of any communication failure or server run-time failure that occurs in a remote procedure call to be stored in a parameter or returned as an operation result, instead of being raised to the client code as an exception.

The IDL **comm_status** attribute is similar to the corresponding NIDL attribute. Use the **comm_status** attribute to report communication errors that may occur when executing remote procedure calls.

DCE IDL also supports the **fault_status** attribute, which you use to report errors in the execution of an operation in the application.

Use the **comm_status** attribute if a call to an operation has a recovery action to perform when communications failures occur (for example, **rpc_s_comm_failure** or **rpc_s_no_more_bindings**).

Use the **fault_status** attribute when a calling application has a recovery action to perform depending on how server faults occur (for example, **rpc_s_fault_invalid_tag**, **rpc_s_fault_pipe_comm_error**, **rpc_s_fault_int_overflow**, or **rpc_s_fault_remote_no_memory**).

3.11.3 Controlling Client Stub Generation with code and nocode

The **code** and **nocode** attributes specify whether the IDL compiler will generate client stub code for the associated interface or operation. Specifying either **code** or **nocode** as an interface attribute establishes the default behavior for all operations in the interface. A **code** or **nocode** operation attribute on an individual operation can override the default behavior.

At least one operation in the interface must have the **code** attribute. If you do not specify any attribute, **code** is assumed, and the IDL compiler generates client stub code for all operations in the interface.

3.11.4 Controlling the Marshalling of Code with in_line and out_of_line

The **in_line** and **out_of_line** attributes control whether the marshalling and unmarshalling of data are performed by in-line code or by out-of-line code through a subroutine call. These can be either interface or operation attributes. The interface attribute establishes the default behavior of all operations in the interface. The operation attribute can override the default interface behavior; it defines the behavior for a single operation.

In-line code executes faster than out-of-line code, but the executable files are larger.

3.11.5 Controlling Data Representation with represent_as

The **represent_as** attribute is used to associate a local data type that your application uses with a data type defined in the IDL file. If you use the **represent_as** attribute, conversions between the local (language) representation and network (IDL) data representation occur during the marshalling and unmarshalling conversions. The DCE **represent_as** attribute is similar to the NCS 1.5.1 **transmit_as** attribute, except that the interface is slightly different. Both require you to supply data conversion routines.

3.11.6 Initializing Memory Management Routines with enable_allocate

The **enable_allocate** attribute on an operation forces the server stub to initialize the **rpc_ss_allocate** routine, which performs memory management. It has no effect if the operation uses full pointers (**ptr**) or a type with the **represent_as** attribute, either of which causes the server stub to enable **rpc_ss_allocate** automatically.

3.11.7 Allocating Objects from the Heap

The **heap** attribute specifies that the server stub's copy of a parameter (or all parameters) be allocated in heap memory rather than on the stack. **heap** can be a type attribute or a parameter attribute. When used with a parameter, **heap** affects only the associated parameter. As a type attribute, **heap** affects all parameters of that type. The **heap** attribute is useful to prevent large objects from exhausting available stack space.

3.12 Example of an Interface Definition and Attribute Configuration File

Figure 3–1 is an example of a DCE interface definition. Figure 3–2 is an example of an attribute configuration file that might accompany this interface definition.

```
[
uuid(810a7d99-956c-13ca-859f-08001e022936),
version (1.0)
]
interface my_interface
{
    typedef long T[10];
    void first_op(
          [in] handle_t h,
          [in] error_status_t st
    );
    void next_op(
          [in] handle_t h,
          [out] error_status_t *st
   );
}
```

Figure 3–1. DCE Interface Definition

```
[in_line, nocode] interface my_interface
{
    include "some_types";
    typedef [out_of_line, heap] T;
    [code] first_op();
    next_op([comm_status] st);
}
```

Figure 3–2. DCE Attribute Configuration File

3.13 DCE IDL Output Files

The IDL compiler uses the information in an interface definition to generate header files, client stub files, and server stub files. The IDL compiler produces header files in C and can produce stubs either as C source files or as object files. The NCS 1.5.1 NIDL compiler creates only source files.

When generating its client and server stub files, the DCE IDL compiler uses a naming convention similar to that used by the NIDL compiler. The names of interface definition files end with the suffix **.idl**. The **idl** compiler replaces the **.idl** with other extensions as shown in Table 3–4.

Table 3–4. NIDL and IDL Output Files

File	NCS 1.5.1 Output File	DCE Output File
Header files	**.h**	**.h**
Client stub files	**_cstub.c**	**_cstub.c**
Server stub files	**_sstub.c**	**_sstub.c**
Client switch files	**_cswtch.c**	N/A
Client auxiliary files	N/A	**_caux.c**
Server auxiliary files	N/A	**_saux.c**

Note that the IDL compiler does not generate client switch files.

The DCE IDL compiler generates additional client and server auxiliary files when necessary. An **auxiliary file** contains auxiliary routines that the IDL compiler generates when out-of-line marshalling is requested or when pipes or certain kinds of pointer-based structures are used.

Chapter 4

Writing DCE Interface Definitions

The first step in developing a distributed application is to define its interface or interfaces in the DCE Interface Definition Language (IDL). This chapter describes the processes of converting existing interface definitions using a translator tool and of creating a new DCE IDL interface definition.

After writing an interface definition, you must write the client and server programs and build a DCE application. Chapter 5 describes how.

Before reading this chapter, you may want to become familiar with the DCE IDL functionality and how it compares to NCS 1.5.1 NIDL. Chapter 3 covers this information.

4.1 Using the nidl_to_idl Tool to Convert NCS 1.5.1 Interface Definitions

When you are converting an application from NCS 1.5.1 to DCE RPC, you do *not* have to rewrite the NCS 1.5.1 interface definition. DCE supplies a conversion tool that generates an IDL file using the DCE syntax. This tool, **nidl_to_idl**, converts both the C and Pascal syntaxes of NCS 1.5.1 NIDL into the DCE IDL syntax.

4.1.1 The nidl_to_idl Tool for Translating Interface Definitions

The **nidl_to_idl** tool helps automate the process of converting an NCS 1.5.1 interface definition to DCE. DCE applications built from the resulting DCE interface definition can

interoperate with NCS programs built from the original NIDL interface definition because the tool guarantees that the network data representation for the interface definition is unchanged.

The **nidl_to_idl** tool accepts the following input:

- An interface definition file that compiled successfully under the NCS Version 1 NIDL compiler
- Arguments to indicate either special actions to be performed by the translator or special properties of the input or output files

The tool translates the NCS 1.5.1 interface definition into a DCE IDL interface definition. It also creates an attribute configuration file if one is required. Once the translation is complete, you must compile the translated output file using the DCE IDL compiler. See the **nidl_to_idl**(1rpc) man page for complete information on the **nidl_to_idl** tool.

The **nidl_to_idl** translator takes care of most of the translation automatically. In some cases, particularly if the original interface definition contained complex type definitions, **nidl_to_idl** may not recognize the intended construct. In this case, it generates a **v2.idl** file but it issues a warning. You must then edit the **v2.idl** file manually to correct the problem. Often, you need only copy the construct from your NCS 1.5.1 interface definition and make minor edits.

4.1.2 Invoking the nidl_to_idl Tool

You invoke the **nidl_to_idl** tool, specifying your NCS 1.5.1 interface definition, as follows:

% **nidl_to_idl** *name*.**idl**

If it succeeds, the tool creates a new file called *name*_**v2.idl**. If your NCS 1.5.1 interface definition contained information that now belongs in the attribute configuration file, it also generates a *name*_**v2.acf**.

Figure 4–1 contains an NCS 1.5.1 interface definition called **binopfw.idl**. Figure 4–2 shows the interface definition created by **nidl_to_idl**, called **binopfw_v2.idl**. This figure also shows two minor changes that were made by hand. The arrows (⇐ and ⇑) point to the changes that were made by hand, as described in the following paragraphs.

```
%c
[uuid(4448ee491000.0d.00.00.fe.da.00.00.00), version(1)]

interface binopfw
{
    [idempotent]
        void binopfw$add(handle_t [in] h,
                         long [in] a, long [in] b, long [out] *c);
}
```

Figure 4–1. NCS 1.5.1 Interface Definition

```
[
 /* V1 format UUID: 4448ee491000.0d.00.00.fe.da.00.00.00 */
 uuid(4448EE49-1000-0000-0D00-00FEDA000000),
 version(2)]  ⇐
interface binopfw
{

    [idempotent] void binopfw_add
    (                          ⇑
        [in] handle_t h,
        [in] long a,
        [in] long b,
        [out] long *c
    );
}
```

Figure 4–2. DCE Interface Definition Created with nidl_to_idl

In NCS 1.5.1 interface definitions, the version number specified by the version attribute is always an integer. In DCE RPC interface definitions, the version number can be either a single integer or a pair of integers called the major and minor version numbers. A client and server can communicate only if the interface imported by the client has the same major version number as the interface exported by the server, and the interface imported by the client has a minor version number less than or equal to the minor version number of the interface exported by the server.

The **nidl_to_idl** translator retains version 1 as the version of the interface definition. If you are making changes to the interface definition, you must change either the major or minor version number of the interface.

- If the changes are incompatible, you must increase the major version number.
- If the changes are compatible, you must increase either the major or the minor version number.
- If you need interoperability between new clients and existing servers, you should not make any changes in the interface definition, and you should not change the version number.

In Figure 4–2, we increased the version number from 1 to 2 because we have changed some of the operation names. See the *OSF DCE Application Development Guide* for more information about version numbers and compatible changes.

The **nidl_to_idl** tool preserves the operation names as they were originally declared. So, if you used the dollar sign ($) in operation names according to the NCS 1.5.1 convention, we recommend that you replace the $ with an underscore (_) to comply with ANSI C standards, unless you need to preserve interoperability between new clients and existing servers. You can continue to use the $ in operation names when you compile in ANSI C mode if you specify the **+e** C compiler flag. Figure 4–2 illustrates the ANSI-compliant interface definition.

4.1.3 IDL Attributes for Compatibility with NCS 1.5.1

To provide interoperability with NCS 1.5.1 programs, the **nidl_to_idl** tool directs the IDL compiler to generate the NCS 1.5.1 representation for those type constructors whose network data representation has changed at DCE. It does so by adding a special attribute to the declaration of any of the changed type constructors: arrays, structures, strings, or enumerations. These attributes, **v1_array**, **v1_struct**, **v1_string**, and **v1_enum**, tell the IDL compiler to generate code that marshalls and unmarshalls the NCS 1.5.1 network data representation for these type constructors.

Using the **nidl_to_idl** tool is a good way to begin migrating your applications to DCE because it offers interoperability with NCS 1.5.1 programs. Once you begin adding DCE functionality to your interfaces, you should create DCE interface definitions that do *not* contain the attributes provided for compatibility with NCS 1.5.1 because they may not be supported in later versions of DCE.

Figure 4–3 shows an NCS 1.5.1 interface definition which declares a string operation using the NIDL C-style null-terminated string, **string0**. Figure 4–4 shows the command line used to invoke the **nidl_to_idl** tool. It also shows the output file with the **_v2.idl** extension that **nidl_to_idl** generated. The tool inserts **v1_** attributes in the type declarations to handle the NCS 1.5.1 **string0** type, which is not supported in DCE IDL syntax.

```
%c
[
uuid(41c460b7a000.0d.00.00.c3.66.00.00.00),
version(1)
]
interface string0test {

typedef string0[100] string0test$t1;

void string0test$op1(
    handle_t [in]h,
    string0test$t1 [in] i,
    string0test$t1 [out] o
);

}
```

Figure 4–3. NCS 1.5.1 Interface Definition, string.idl

% nidl_to_idl string.idl
% cat string_v2.idl

```
 [
 /* V1 format UUID: 41c460b7a000.0d.00.00.c3.66.00.00.00 */
 uuid(41C460B7-A000-0000-0D00-00C366000000),
 version(1)]
interface string0test
{
    typedef [v1_array, v1_string] char string0test$t1[100];

    void string0test$op1
    (
      [in] handle_t h,
      [in] string0test$t1 i,
      [out] string0test$t1 o
    );
}
```

Figure 4–4. Conversion of string.idl to string_v2.idl

4.1.4 Creating an Attribute Configuration File

The **nidl_to_idl** tool creates a separate **.acf** file for those attributes that belong in this file. Given the NIDL interface definition in the file **params.idl** shown in Figure 4–5, the **nidl_to_idl** tool generates a warning, as shown in Figure 4–6, that it has created the **.acf** file **params_v2.acf**, in addition to the interface definition in file **params_v2.idl**.

```
%c
[
uuid(41248b269000.0d.00.00.c3.66.00.00.00),
version(1)
]
interface pattrtest {

void pattrtest$op1(
  handle_t [in] h,
  status_$t [in, out, comm_status] *st
  );

void pattrtest$op2(
  handle_t [in] h,
  status_$t [in] st_in,
  status_$t [out, comm_status] *st_out
  );
}
```

Figure 4–5. NCS 1.5.1 Interface Definition, params.idl

```
% nidl_to_idl params.idl
*** Warning: Parameter 'st' in operation 'pattrtest$op1':
  [comm_status] requires .acf file
*** Warning: Parameter 'st_out' in operation 'pattrtest$op2':
  [comm_status] requires .acf file
```

Figure 4–6. nidl_to_idl Warning

Figure 4–7 shows the interface definition, **params_v2.idl**, and Figure 4–8 shows the additional **.acf** file, **params_v2.acf**, created from the **params.idl** interface definition.

```
[
 /* V1 format UUID: 41248b269000.0d.00.00.c3.66.00.00.00 */
 uuid(41248B26-9000-0000-0D00-00C366000000),
 version(1)]
interface pattrtest
{

    void pattrtest$op1
    (
      [in] handle_t h,
      [in, out] error_status_t *st
    );

    void pattrtest$op2
    (
      [in] handle_t h,
      [in] error_status_t st_in,
      [out] error_status_t *st_out
    );
}
```

Figure 4–7. Conversion of params.idl to params_v2.idl

```
interface pattrtest
{
    pattrtest$op1 ([comm_status] st);
    pattrtest$op2 ([comm_status] st_out);
}
```

Figure 4–8. The params_v2.acf Attribute Configuration File

4.1.5 Converting from the NIDL Pascal Syntax

The **nidl_to_idl** tool creates a C-like IDL file when given a file written in NIDL's Pascal-like syntax. Figure 4–9 shows an NCS 1.5.1 interface definition written in Pascal syntax. Figure 4–10 shows what the resulting DCE IDL interface definition looks like.

```
%pascal
[
uuid(41248b185000.0d.00.00.c3.66.00.00.00),
version(1)
]
{* Another Interface Definition *}
interface tattrtest;

type tattrtest$t1 = [handle] integer32;
);

function tattrtest$op1 (
    in h: handle_t;
    out msg: string0[100]
): boolean;
end;
```

Figure 4–9. NCS 1.5.1 Interface Definition in NIDL Pascal Syntax

```
[
 /* V1 format UUID: 41248b185000.0d.00.00.c3.66.00.00.00 */
 uuid(41248B18-5000-0000-0D00-00C366000000),
 version(1)]
interface tattrtest
{
/* * Another Interface Definition * */
    typedef [handle] long tattrtest$t1;

    boolean tattrtest$op1
    (
      [in] handle_t h,
      [out, v1_array, v1_string] char msg[0..99]
    );
}
```

Figure 4–10. Conversion of the Pascal Interface Definition to DCE IDL

After converting your NCS 1.5.1 interface definition files to DCE versions, you must now compile the file or files with the DCE IDL compiler to generate the stub and header files as described in Section 4.3.

4.2 Writing New DCE Interface Definitions

The DCE process for writing interface definitions and generating stubs is similar to the same process for NCS. You must:

1. Generate an interface UUID by running **uuidgen**. (This is the DCE analog to the NCS 1.5.1 **uuid_gen** tool.)
2. Write an interface definition in IDL. (Refer to the IDL syntax description in the *OSF DCE Application Development Guide.*)
3. Write an attribute configuration file (ACF), if necessary, to define attributes that modify the interaction between the application code and stubs. (Refer to the ACF syntax description in the *OSF DCE Application Development Guide.*)
4. Compile the interface definition using the IDL compiler.

4.2.1 Generating Interface UUIDs

If you are creating a new DCE interface definition, run **uuidgen** with the **–i** option to generate a UUID and create an IDL file template that includes the generated UUID string in the template; for example:

% uuidgen –i > bino.idl
% cat bino.idl

```
[
uuid(202c81a8-5663-11ca-bbf1-08001e01b30d),
version(1.0)
]
interface INTERFACENAME
{

}
```

For more information about **uuidgen**, see the *OSF DCE Application Development Guide* and the **uuidgen**(1) manual page.

4.2.2 Writing the Interface Definition

After generating the interface definition template, you must edit it to supply the interface name, attributes, and operations.

4.2.2.1 Naming the Interface

To name the interface, follow the same conventions as for NCS 1.5.1. (The interface name must be at most seventeen characters long and a valid IDL identifier; see the *OSF DCE Application Development Guide.*)

After you have used **uuidgen –i** to generate a skeletal interface definition, replace the dummy string "INTERFACENAME" with the name of your interface. By convention, the name of an interface definition file is the same as the name of the interface it defines, along with the suffix **.idl**. For example, the definition for a **bank** interface would reside in a **bank.idl** interface definition file.

4.2.2.2 Specifying Interface Definition Attributes

You specify interface attributes within brackets in the header of the interface definition as in NCS 1.5.1. For a list of the possible interface attributes, see Section 3.5.

The following is an example of an interface definition using **uuid** and **version** attributes:

```
[uuid (df961f80-2d24-11c9-be74-008002b0ecef1), version (1.1)]
interface my_interface
```

4.2.2.3 Specifying Import Declarations

You specify import declarations in DCE IDL in the same way as you do in NCS 1.5.1. The IDL **import** declaration specifies another interface definition whose types and constants are used by the importing interface.

4.2.2.4 Specifying Constant Declarations

The DCE IDL constant declaration, which provides a way for getting **#define** statements into derived **.h** files, supports more data types that the NCS 1.5.1 constant declaration. In NCS 1.5.1, the constant declaration supports only integer and character constants. The DCE

IDL constant declaration supports integer, boolean, character, string, and null pointer constants, and constant expressions. In addition, string constants can be declared as **char ***. For examples, see Section 3.7.

4.2.2.5 Specifying Type Declarations

IDL, like NIDL, provides a variety of data types, including

- Simple types, such as integers, floating-point numbers, characters, booleans, and the primitive binding handle type (**handle_t**).
- Constructed types, such as pipes, pointers, arrays, strings, structures, and unions.

These type declarations and data types are described in detail in Section 3.2.

Note that if you use the new DCE data types and attributes in an interface definition file, the resulting **.idl** file will not be interoperable with an NCS 1.5.1 version of the interface definition.

4.2.2.6 Specifying Operation Declarations

Operation declarations specify the signature of each operation in the interface, including the operation name, the type of data returned (if any), and the types of all parameters passed in a call. They also specify various field, parameter, and operation attributes. For a list of possible operation declarations, see Section 3.9.

4.2.3 Writing an Attribute Configuration File

Write an attribute configuration file (ACF) if you want to define attributes that tailor how the RPC interface appears to the application code. These attributes affect the interface between the application code and stubs; they do not affect the interoperability of stubs with other stubs.

When you invoke the IDL compiler, it checks for an attribute configuration file related to the interface definition you specify. The compiler looks for a file with the same name as the interface definition, with the **.acf** instead of the **.idl** extension. The IDL compiler searches the current directory and then any directories specified via **–I** options on the **idl** command line. If an ACF exists, it is compiled with the IDL file.

For a list of possible attributes that you can declare in an ACF, see Section 3.11.

4.3 Running the IDL Compiler

After you have written an interface definition or converted an existing interface definition with **nidl_to_idl**, run the IDL compiler to generate stub and header files as follows:

% **idl** *filename* [*options*]

where *filename* is the pathname of the interface definition file and *options* specifies any number of possible compiler options. See the **idl(1rpc)** man page in the *OSF DCE Application Development Reference* for complete information on running the IDL compiler.

A typical **idl** command line for a newly converted NCS 1.5.1 interface generates the files in the same manner as for NIDL. For example, the following line compiles **binopfw_v2.idl**, which is shown in Figure 4–2.

% idl binopfw_v2.idl –I/dce/latest/usr/include –no_cpp –keep c_source

The **–I** option is similar to the NIDL **–idir** option; it indicates the directory that contains imported interface definitions.

The **–no_cpp** option suppresses preprocessing of **.idl** and **.acf** files by the C preprocessor. The **–keep c_source** option tells the compiler to save the C source modules when running. By default, the IDL compiler runs the C compiler on the generated stubs. The NCS 1.5.1 NIDL compiler does not automatically invoke the C compiler.

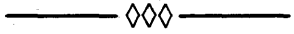
Note that you do not need to specify a **–m** or **–s** option to specify whether to support multiple managers or multiple interface versions. DCE IDL always provides this support.

You can also use the **–no_mepv** option if you are supplying manager code with operation names that differ from the operation names in the IDL file. Otherwise, the IDL compiler generates a manager EPV in the server stub using the names of the operations in the IDL file.

On UNIX systems, the IDL compiler generates the following **binopfw** files:

binopfw_v2.h
binopfw_v2_sstub.c
binopfw_v2_cstub.c

If the IDL compiler generates auxiliary files (with the **_caux.c** and **_saux.c** extensions), you must link them with the client and server programs, just as you do the **_sstub.c** and **_cstub.c** files.



Chapter 5

Converting Distributed Applications to DCE RPC

This chapter describes how to convert your NCS 1.5.1 client and server applications to use the DCE RPC API and then rebuild your distributed applications. This chapter also includes examples of using the DCE exception-returning package, which replaces the **pfm_$** exception-handling routines. Before converting your application, you must convert your interface definition as described in Chapter 4.

To illustrate the conversion from NCS 1.5.1 to DCE RPC, we converted the **binop** forwarding program, **binopfw**, that was shipped as a programming example in NCS 1.5.1. This chapter describes the DCE RPC version of **binopfw**. Appendix B lists the NCS 1.5.1 code so that you can compare the changes made here.

You may also wish to refer to the **stacks** application, which is described in Chapter 5 and is available online as part of the HP DCE Application Development Tools.

5.1 Converting the Client Code

This section lists some of the major things you need to do to convert an existing NCS 1.5.1 client module to a DCE RPC client module.

This example illustrates locating a server by passing the server's host name on the command line, one of the two ways for clients to locate servers. (Clients can also use the Name Service Interface (NSI) to locate servers.)

The **binopfw** client is built from the following source code modules:

- **client.c**
- **util.c**
- **binopfw_v2_cstub.c**

The stub module, **binopfw_v2_cstub.c**, is generated by the IDL compiler from the interface definition, **binopfw.idl**. Section 5.3 describes how to convert the **util.c** module, which is used by both the client and the server to generate error messages.

Figure 5–1 contains the C source code for the DCE RPC version of **client.c**. Comments in the code highlight the changes made from the NCS 1.5.1 version. The program takes two arguments: the host name (or network address) of the server that is running and the number of passes to execute. This program does *not* require the endpoint of the server; the **rpcd** process on the server's system forwards the call to the server.

Some of the changes made to the DCE RPC version of **client.c** include:

- Different include files are required. The **binopfw_v2.h** file tool contains a **#include** directive for **rpc.h**. The order in which the include files are listed is important, as some later include files rely on declarations contained in previous include files.
- Rather than calling **pfm_$init** to handle faults via the Process Fault Manager (PFM), this module uses the DCE exception-returning package to handle exceptions. (For details on the package, see Section 5.7 and the *OSF DCE Application Development Guide*.)
- The client code no longer references a **uuid_$nil** global variable. Calls that previously required **uuid_$nil** will typically accept NULL.
- To understand how to replace the NCS 1.5.1 routines, refer to the information on mapping NCS 1.5.1 to DCE RPC calls in Chapter 2.
- Since we do not specify an endpoint, the **rpc_binding_from_string_binding** call generates a partially bound handle. The first time the client calls the remote procedure **binopfw_add**, the call is sent to the **rpcd** process at the server's system, and then **rpcd** forwards the call to the server. On return, the handle is fully bound, so that all subsequent calls are sent directly to the server endpoint. After each pass, the client prints the real elapsed time per call and the fully bound handle. After the last pass, the program exits.

- In this example, there is no need to build a binding handle from the supplied arguments (with **rpc_string_binding_compose**) as the user supplies the string binding on the command line. The object UUID is nil because **binopfw** does not operate on any particular object.

See Appendix B for the NCS 1.5.1 version of **client.c**.

```
#include <stdio.h>
#include <signal.h>
#include <dce/pthread_exc.h>
#include <dce/dce_error.h>
#include <dce/rpcexc.h>
#include "binopfw_v2.h"
 #define CALLS_PER_PASS 100
ndr_char   nils[] = "";
extern char *error_text();     /* New error call in util.c */

int main(int argc, char *argv[])       /* ANSI C function */
{
    rpc_binding_handle_t   handle;     /* Was handle_t */
    sigset_t      sigset;
    pthread_t     thread;
    ndr_char      *string_binding;     /* New string binding */
    unsigned32    st;                  /* Was status_$t */
    int           k;
    int           passes;
    ndr_long_int  i, n;                /* Was ndr_$long_int */
    long         start_time, stop_time;


    if (argc != 3)
    {
        fprintf(stderr,
           "Usage: %s ProtocolSequence:HostID passes\n", argv[0]);
        exit(1);
    }
    passes = atoi(argv[2]);

/* Map SIGINT and SIGHUP to cancels to catch asynch signals */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);
    sigaddset(&sigset, SIGHUP);
    thread = pthread_self();
    pthread_signal_to_cancel_np(&sigset, &thread);
```

Figure 5–1. DCE RPC Version of binopfw/client.c

```
/* Create binding from string binding passed from command line */
    rpc_binding_from_string_binding(
        (unsigned_char_t *) argv[1], &handle, &st);
    if (st != rpc_s_ok)
    {
        fprintf(stderr,
           "Cannot convert name \"%s\" to binding - %s\n",
            argv[1], error_text(st));
        exit(1);
    }

/* Convert binding handle to string representation to print name */
    rpc_binding_to_string_binding(handle, &string_binding, &st);
    if (st != rpc_s_ok)
    {
        fprintf(stderr,
            "Cannot get string binding - %s\n",error_text(st));
        exit(1);
    }
    printf("Bound to %s\n\n", (char *)string_binding);

/* Free string when done */
    rpc_string_free(&string_binding, &st);
    if (st != rpc_s_ok)
    {
        fprintf(stderr,
            "Cannot free string binding %s - %s\n",
             string_binding, error_text(st));
        exit(1);
    }
    for (k = 1; k <= passes; k++)
    {
        start_time = time(NULL);
        for (i = 1; i <= CALLS_PER_PASS; i++)
        {

/*Enclose rpc with exception-returning macros */
          TRY
          {
               binopfw_add(handle, i, i, &n);
               if (n != i+i)
                  printf("Two times %ld is NOT %ld\n", i, n);
           }
```

Figure 5–1. DCE RPC Version of binopfw/client.c (Continued)

```
CATCH (rpc_x_comm_failure)
            {
                printf ("Call failed: comm failure\n");
                exit(1);
            }
            CATCH_ALL
            {
                printf (" Call failed for unknown reason\n");
                exit(1);
            }
            ENDTRY
        } /* for i loop */
        stop_time = time(NULL);

/* Added this call to print the endpoint after each pass */
        rpc_binding_to_string_binding(handle,
            &string_binding, &st);
        if (st != rpc_s_ok)
        {
           fprintf(stderr,
           "Cannot get string binding - %s\n",error_text(st));
           exit(1);
        }

         printf("Pass %3d; Real/Call: %2ld ms\n",
            k, ((stop_time - start_time) * 1000) / CALLS_PER_PASS);
        printf("Bound to %s\n\n", (char *)string_binding);
    } /* for k loop */

} /* main */
```

Figure 5–1. DCE RPC Version of binopfw/client.c (Continued)

5.2 Converting the Server Code

The **binopfw** server is built from the following four source code modules:

- **server.c**
- **manager.c**
- **util.c**
- **binopfw_v2_sstub.c**

The **server.c** module performs the server initialization. The **manager.c** module contains the **binopfw_add** routine that executes the addition operations. The stub module is generated by the IDL compiler from the interface definition (as described in Chapter 4).

5.2.1 Initializing a DCE Server

A DCE server includes initialization code that prepares the server to receive remote calls. The initialization code typically includes these steps:

1. Register the interface with the DCE runtime library.
2. Specify which protocol sequences the server will use.
3. Obtain a list of binding handles.
4. Register endpoints in the local endpoint map.
5. Export binding information to an entry (or entries) in the Name Service database.
6. Finally, listen for client requests.

The converted code in the **server.c** module performs all of these steps except Step 5, which isn't needed because the user of the client program specifies a server host on the command line.

5.2.2 The server.c Module

Figure 5–2 shows the DCE RPC version of **server.c**; code comments highlight the changes from the previous version. The **server.c** program takes as an argument the textual name of the protocol sequence that you are using. To compare this code with the NCS 1.5.1 version, see Appendix B.

Some of the changes between the NCS 1.5.1 and DCE RPC versions of **server.c** include:

- Different header files to declare. The order in which the include files are listed is important, as some later include files rely on declarations contained in previous include files.
- The global variables are similar to NCS 1.5.1; however, the names support DCE RPC major and minor version numbers.
- To understand how to replace the NCS 1.5.1 routines, refer to the information on mapping NCS 1.5.1 to DCE RPC routines in Chapter 2.
- Rather than calling **pfm_$init** to handle faults, this module uses the DCE exception-returning macros to handle exceptions. (For details on the macros, see Section 5.7 and Appendix C).
- The second parameter of the DCE RPC **rpc_server_use_protseq** routine requires you to specify the maximum number of concurrent remote procedure call requests that the server wants to handle. The RPC runtime guarantees that the server can accept at least this number of concurrent call requests. Use the predefined constant **rpc_c_protseq_max_reqs_default** to specify the default value.
- The server code no longer references a global variable such as **uuid_$nil**. Calls which previously required **uuid_$nil** will typically accept NULL.
- The DCE RPC code makes calls that are analogous to the NCS 1.5.1 calls to get the name of the protocol sequence, get an endpoint on which to listen, register the manager with the RPC runtime library, register the server with **rpcd** so that **rpcd** can forward requests to the server, and listen for requests.

```
/* New include files */
#include <stdio.h>
#include <signal.h>
#include <dce/pthread_exc.h>
#include <dce/dce_error.h>
#include "binopfw_v2.h"
/* Change globalref to extern */
extern binopfw_v2_0_epv_t  binopfw_v2_0_manager_epv;
extern char *error_text();
int main(int argc, char *argv[])   /* ANSI C function */
{
    unsigned32           st;
    sigset_t             sigset;
    pthread_t            thread;
    rpc_binding_vector_t *bh_vector;    /* New vector of handles */
    unsigned_char_t      *string_binding; /* New string binding */
    int                  i;
    unsigned32           MAX_CALLS=5;
    if (argc != 2)
    {
        fprintf(stderr,
        "Usage: %s Protocol Sequence (ncadg_ip_udp | ncacn_ip_tcp)\n",
         argv[0]);
       exit(1);
    }

/* Map SIGINT and SIGHUP to cancels to catch asynch signals */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);
    sigaddset(&sigset, SIGHUP);
    thread = pthread_self();
    pthread_signal_to_cancel_np(&sigset, &thread);

/* Get protocol sequence */
    rpc_server_use_protseq((ndr_char *)argv[1],
                           rpc_c_protseq_max_reqs_default, &st);
    if (st != rpc_s_ok)
    {
        fprintf(stderr, "Cannot use protocol sequence: %s\n",
                argv[1], error_text(st));
        exit(1);
```

Figure 5–2. DCE RPC Version of binopfw/server.c

```
    }/* Register interface with RPC runtime */

    rpc_server_register_if(binopfw_v2_0_s_ifspec, NULL,
            (rpc_mgr_epv_t)&binopfw_v2_0_manager_epv, &st);
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot register interface: %s\n",
          error_text(st));
          exit(1);
    }
/* Get vector binding handles */
    rpc_server_inq_bindings(&bh_vector, &st);
    if (st != rpc_s_ok) {
      fprintf(stderr, "Cannot inquire about binding vectors:%s\n",
          error_text(st));
          exit(1);
    }

/* Print bindings */
    printf("Bindings:\n");
    for (i = 0; i < bh_vector->count; i++) {
        rpc_binding_to_string_binding(bh_vector->binding_h[i],
                &string_binding, &st);
        if (st != rpc_s_ok) {
              fprintf(stderr, "Cannot get string binding - %s\n",
                  error_text(st));
            exit(1);
        }
        printf("%s\n", (char *)string_binding);
        rpc_string_free(&string_binding, &st);
        if (st != rpc_s_ok) {
            fprintf(stderr, "Cannot free string binding %s - %s\n",
                    string_binding, error_text (st));
                    exit(1);
        }
    }  /* for */
/* Register interface with endpoint map database */
    rpc_ep_register(binopfw_v2_0_s_ifspec, bh_vector,
            (uuid_vector_t *) NULL,
            (unsigned_char_t *) "binop version 2.0 server", &st);
 if (st != rpc_s_ok)
    {
        fprintf(stderr,
                    "Cannot register interface with endpoint map: %s\n",
                error_text(st));
          exit(1);
    }
```

Figure 5–2. DCE RPC Version of binopfw/server.c (Continued)

```
/* Set up exception returning before listening on endpoint */
     TRY
    {
        printf("Listening...\n");
        rpc_server_listen(MAX_CALLS, &st);
        if (st != rpc_s_ok)
              fprintf(stderr, "Error: %s\n",error_text(st));
    }  /* try */
    FINALLY
    {
/* Do same clean-up operation in normal and exception exit */
        printf("Unregistering interface\n");
        rpc_server_unregister_if(binopfw_v2_0_s_ifspec,
             NULL,&st);
        if (st != rpc_s_ok)
        {
            fprintf(stderr,
           "Cannot unregister interface: %s\n",error_text(st));
            exit(1);
        }
        printf("Unregistering endpoint\n");
        rpc_ep_unregister(binopfw_v2_0_s_ifspec, bh_vector,
            (uuid_vector_p_t) NULL, &st);

        if (st != rpc_s_ok)
        {
            fprintf(stderr, "Cannot unregister endpoint: %s\n",
            error_text(st));
            exit(1);
        }
        exit(0);
    }  /* finally */
    ENDTRY;
} /* main */
```

Figure 5–2. DCE RPC Version of binopfw/server.c (Continued)

5.2.3 The manager.c Module

The **manager.c** module is similar in both NCS 1.5.1 and DCE RPC. The manager makes no DCE calls, so it includes only the **binopfw_v2.h** header file, which defines **binopfw_v2_0_epv_t** and declares the **binopfw_add** operation.

```
#include "binopfw_v2.h"  /* Use version created by idl compiler */

static void binopfw_add(/* Use same name as in interface definition file */
    handle_t h,
    ndr_long_int a,
    ndr_long_int b,
    ndr_long_int *c)
{
    *c = a + b;
}

extern binopfw_v2_0_epv_t binopfw_v2_0_manager_epv = {binopfw_add};
```

Figure 5–3. DCE RPC Version of binopfw/manager.c

5.3 Converting the util.c Module

Both NCS 1.5.1 and DCE RPC versions of this application define an additional module, **util.c**, which the client and server code use to generate the text of error messages. Figure 5–4 shows the DCE RPC version of **util.c**, which uses the DCE RPC **dce_error_inq_text** call. Appendix B shows the NCS 1.5.1 version.

```
#include <dce/dce_error.h>
#include "binopfw_v2.h"

char *error_text (st)

unsigned32      st;

{
    static dce_error_string_t error_string;
    int inq_st;

    dce_error_inq_text (st, error_string, &inq_st);
    return ((char *) error_string);
}
```

Figure 5–4. DCE RPC Version of binopfw/util.c

5.4 Building DCE RPC Applications

The following steps list the typical procedure for building a DCE RPC-based application. Refer to the Makefiles installed in the example directory to see how we build the programming examples supplied with DCE RPC.

1. For each interface, run the **idl** command to generate a header file and to generate source code for the server stub, client stub, and any auxiliary files. If you are converting an existing NCS 1.5.1 application and you want to ensure that this interface will interoperate with existing NCS 1.5.1 applications, convert the NCS 1.5.1 interface definition with the **nidl_to_idl** tool before running the **idl** command.

2. For each interface, use the C compiler to generate object modules for the server stub, client stub, and any auxiliary files.

3. For each interface, compile any routines that perform automatic binding or data type conversion.

4. Compile the client application code.

5. Compile the server initialization code and managers.

6. Link the client application object modules, client stubs, any automatic binding routines, and any type conversion routines to make the executable client.

7. Link the server and manager object modules, server stubs, and any type conversion routines to make the executable server.

5.5 Running the binopfw Program

To run the **binopfw** distributed application, you first start the server program and then the client program. Before starting the server program, be sure that the **rpcd** daemon is running on the server host. For details on how to start up **rpcd**, see the *OSF DCE Administration Guide*.

5.5.1 Starting the Server Program

The following is an example of starting the server program:

```
% ./server ncadg_ip_udp

Bindings:
ncadg_ip_udp:15.22.136.147[1032]
Listening...
```

5.5.2 Starting the Client Program

The following is an example of starting the client program:

```
% client ncadg_ip_udp:my_node 3

Bound to ncadg_ip_udp:15.22.136.147[]

Pass   1; Real/Call: 40 ms
Bound to ncadg_ip_udp:15.22.136.147[1032]

Pass   2; Real/Call: 50 ms
Bound to ncadg_ip_udp:15.22.136.147[1032]

Pass   3; Real/Call: 40 ms
Bound to ncadg_ip_udp:15.22.136.147[1032]
```

5.6 Improving the binopfw server.c Program

The previous **server.c** in Figure 5–2 uses the DCE RPC analog for each NCS 1.5.1 routine in the original NCS 1.5.1 **server.c** program. However, we could improve this program using other routines provided in the DCE RPC API.

For example, in the previous program, the server program expects the user to invoke the program specifying the protocol sequence to use. Alternatively, the server can be designed to support all protocol sequences available on a given host by using the DCE call **rpc_server_use_all_protseqs**. (**rpc_server_use_all_protseqs** is a general-purpose call and is typically used more frequently than **rpc_server_use_protseq**.) Using **rpc_server_use_all_protseqs** changes the first part of the **server.c** program as Figure 5–5 illustrates.

To run this new version of **server.c**, invoke the server as follows:

```
% ./server
```

When the server starts up, it prints out all the protocol sequences it supports:

```
Bindings:
ncadg_ip_udp:15.22.136.147[1036]
ncacn_ip_tcp:15.22.136.147[1084]
Listening...
```

```
/* New include files */
#include <stdio.h>
#include <signal.h>
#include <dce/pthread_exc.h>
#include <dce/dce_error.h>
#include "binopfw_v2.h"
/* Change globalref to extern */
extern binopfw_v2_0_epv_t  binopfw_v2_0_manager_epv;
extern char *error_text();
int main()
{
    unsigned32           st;
    sigset_t             sigset;
    pthread_t            thread;
    rpc_binding_vector_t *bh_vector;    /* New vector of handles */
    unsigned_char_t      *string_binding; /* New string binding */
    int                  i;
    unsigned32           MAX_CALLS=5;

/* Map SIGINT and SIGHUP to cancels to catch asynch signals */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);
    sigaddset(&sigset, SIGHUP);
    thread = pthread_self();
    pthread_signal_to_cancel_np(&sigset, &thread);


/* Use all supported protocol sequences */
    rpc_server_use_all_protseqs(MAX_CALLS, &st);
    if (st != rpc_s_ok)
    {
        fprintf(stderr, "Cannot support any protocol sequence: \n",
                error_text(st));
        exit(1);
    }
      }
.
.         /* Rest of program same as in Figure 5-2 */
.
```

Figure 5–5. Server Using rpc_server_use_all_protseqs

5.7 Handling Signals in DCE RPC

NCS 1.5.1 uses the Domain Process Fault Manager (PFM) to catch all UNIX signals (synchronous and asynchronous*) and converted them into faults. For NCS 1.5.1 programs running on the Domain/OS system, the PFM subsystem catches the faults automatically. NCS 1.5.1 programs running on other platforms use the **pfm_$init** routine to enable the Portable PFM (PPFM) subsystem.

The PPFM subsystem is not available with the DCE RPC API. Instead, the DCE RPC API uses the DCE exception-returning package. The DCE exception-returning package is not part of the DCE RPC component, but it accompanies the DCE Threads implementation. The exception-returning package can be used in any threaded application, which includes *all* DCE RPC programs since the DCE RPC product relies on the underlying threads implementation. (DCE Threads are described in Part 2 of the *OSF DCE Application Development Guide* and in *Programmer's Notes on HP DCE Threads*; the exception-returning package is described in the *OSF DCE Application Development Guide*.)

DCE Threads supports the POSIX **sigwait**(2) service to allow threads to perform activities similar to signal handling without having to deal with signals directly. DCE Threads also provides a jacket for **sigaction**(2) to allow each thread to have its own handler for synchronous signals. For more information about **sigwait** and **sigaction**, see the *OSF DCE Application Development Guide* and the **sigwait**(2) and **sigaction**(2) manual pages.

To handle synchronous UNIX signals on a per-thread basis, DCE Threads uses a thread **cancellation** mechanism. When one thread cancels another thread, it is requesting that the target thread terminate as soon as possible. One thread cancels another thread (or itself) by calling **pthread_cancel** with the target thread as the (only) argument. The target thread can control how quickly it terminates by controlling its general cancelability and its asynchronous cancelability. The DCE exception-returning package is integrated with this pthread cancel mechanism. Whenever a thread performs an instruction that induces a UNIX synchronous signal, the DCE exception-returning package maps the signal into a corresponding **exception**.

*__Synchronous__ signals are hardware, operating system, or program errors that occur within the process of an executing program. **Asynchronous** signals are generated *outside* the running process; typically they are in response to a user action (such as a CTRL/c). The UNIX signals SIGSEGV and SIGFPE are synchronous signals. SIGINT, SIGTERM, SIGHUP, and SIGQUIT are asynchronous signals.

DCE Threads supports two ways to obtain information about the status of a threads routine:

- The routine returns a status value to a thread by setting the external variable **errno** to an error code and returning a function value of –1.
- The routine raises an exception using the facilities of the DCE Threads exception-returning interface.

A multithreaded program can use either of these methods but cannot use both methods in the same code module.

The threads manual pages describe returning status using **errno**; this mechanism is the one described in the POSIX P1003.4a proposed standard.

The DCE Threads exception-returning package automatically sets up handlers on each thread for the UNIX synchronous signals and maps these signals to the corresponding exceptions. The package does not map UNIX asynchronous signals automatically, but you can map any asynchronous signals into a cancel exception (**pthread_cancel_e**). This is described in Section 5.7.2.

5.7.1 Using Exceptions with the DCE Exception-Returning Package

The DCE exception-returning package defines a set of macros for manipulating exceptions. For example, the EXCEPTION macro allows you to define exceptions that you want to handle; the TRY macro defines a block wherein exceptions may be caught; and the CATCH macro allows you to perform a specific action based on the exception caught.

When a program includes TRY/CATCH macros, an exception automatically propagates through any nested function calls until it is caught. Any function along the way can catch it, either by specifying it by name (with the CATCH macro) or by catching all exceptions (with the CATCH_ALL macro). Intermediate functions that don't need to release resources can ignore the exception.

When an exception is ignored, that is, if no signal handler exists for the signal, the thread is terminated with a message describing the error condition.

The DCE exception-returning package allows you to

- Declare and initialize exceptions objects
- Raise and catch exceptions
- Handle synchronous signals, which are predefined exceptions
- Handle cancels using the cancel exception
- Handle asynchronous signals by converting the signal into a cancel exception

For examples of the TRY/CATCH macros, see the previous programming examples (Figures 5–1 and 5–2). See the *OSF DCE Application Development Guide* for more information on the specific macros defined in the DCE exception-returning package.

To use the exception-returning macros, your program includes the **dce/exc_handling.h** header file. (This file is included automatically when you include **pthread_exc.h**.) Then you declare a block of code where the exceptions are to be caught with the TRY macro. Within this block (called the **exception scope**), you can define a block of code to process a specific exception or all exceptions with the CATCH or CATCH_ALL macros, respectively.

5.7.2 Handling Asynchronous Signals

By default, if a process receives an asynchronous signal for which no signal handler has been established, the process (with all its threads) simply terminates. If any thread has a signal handler for the asynchronous signal, the signal handler runs in the context of that thread.

You can handle asynchronous signals with the DCE exception-returning package by converting the asynchronous signal into a **pthread_cancel**, which in turn raises the **pthread_cancel_e** exception. Server programs will want to catch asynchronous signals to perform cleanup operations before exiting. The application can map these signals to a pthread cancel by using either a combination of the pthread **sigwait** and **pthread_cancel** routines orthe **pthread_signal_to_cancel_np** routine.

A client program may want to catch asynchronous signals while making a remote procedure call so that a server processing the remote procedure call can be informed that it is going away.

Figure 5–6 is a portion of a program that maps the UNIX signals using a **pthread_signal_to_cancel_np** routine. To use this routine, you must include the **pthread.h** header file.

```
/* New include files */
.
.
.
#include <pthread.h>

int main()

{

   pthread_t       thread;
    sigset_t        sigset;
.
.
.

/* Map SIGINT and SIGHUP to cancels  */

    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);
    sigaddset(&sigset, SIGHUP);
    thread = pthread_self();
    pthread_signal_to_cancel_np(&sigset, &thread);

.
.
.
```

Figure 5–6. Handling Asynchronous Signals

When handling asynchronous signals you need to be aware of *when* the specified asynchronous signals are caught. The corresponding NCS 1.5.1 routine **pfm_$inhibit** returns the signal immediately. However, the DCE Threads cancel mechanism delivers a cancel only at well defined cancellation points in a program, for example when a call to **pthread_cond_wait** is made.

While this default behavior of DCE Threads is considered safer in most cases, you need to be aware of the difference in behavior. In some cases, *not* responding to cancels immediately can cause unexpected behavior. That is, the initial thread may not call a function that can be cancelled in a timely fashion when it catches an asynchronous cancel. This would happen if your program goes into an infinite loop which doesn't contain a call to a function that can be cancelled.

The DCE RPC runtime automatically looks for threads cancels that are posted against a client thread while the thread is performing a remote procedure call. If a pending cancel is detected,

the client runtime forwards the cancel to the server runtime and the server runtime posts the cancel against the executing server call execution thread. The server thread detects the pending cancel if it calls an operation that can be cancelled. If the server calls such an operation, the remote procedure call terminates with a cancel exception and RPC processing continues as it does for all exceptions. If the server completes without handling the pending cancel, the remote procedure call completes successfully; the client continues (after the remote procedure call) with the cancel still pending.

5.8 No Replacement for rpc_$set_fault_mode

NCS 1.5.1 provided the **rpc_$set_fault_mode** routine to help debug server code. This routine allowed you to change the default fault-handling behavior of the NCS runtime so that the server would exit when it received a signal rather than propagating the signal to the client.

DCE does not provide a mechanism that allows you to change default fault-handling behavior.

For the HP DCE Developers' Environment release, the DCE library has been built so that if a synchronous terminating signal is generated, the entire process is terminated and a core dump produced. This has been done to aid application development and debugging. If you do not want this behavior, you can establish a signal handler to handle synchronous terminating signals. Refer to *Programmer's Notes on HP DCE Threads* for more details. We recommend that you do not rely on this behavior, as future releases of HP DCE may not handle synchronous terminating signals in the same manner.

5.9 Using NCS 1.5.1 and DCE RPC UUIDs

While the UUID data structure remains the same size in the DCE RPC API, its internal structure has changed, as Figure 5–7 illustrates.

You need to be aware of these differences in programs that use both **uuid_$t** and **uuid_t** data types. If you want your program to preserve the UUID created in the NCS 1.5.1 format and use it in a DCE RPC-based program, you can. A program might do this, for example, if a DCE RPC-based server must service requests from NCS 1.5.1-based clients or if an object can be accessed by both NCS 1.5.1 and DCE RPC-based servers.

NCS 1.5.1 uuid_$t

```
struct uuid_$t {
ndr_$ulong_int time_high;
ndr_$ushort_int time_low;
ndr_$ushort_int reserved;
ndr_$byte family;
ndr_$byte host[7];
};
```

DCE RPC uuid_t

```
typedef struct  {
unsigned32 time_low;
unsigned16 time_mid;
unsigned16 time_hi_and_version;
unsigned8 clock_seq_hi_and_reserved;
unsigned8 clock_seq_low;
idl_byte node[6];
} uuid_t;
```

Figure 5–7. Comparison of uuid_$t and uuid_t

To preserve the UUID that you used to initialize a variable of the NCS 1.5.1 type **uuid_$t** in a C program, you must change its format. You can easily make this change by moving a brace one field to the right. The last seven bytes of an NCS 1.5.1 **uuid_$t** initialization are enclosed in braces, whereas only the last six bytes of a DCE RPC **uuid_t** initialization are enclosed in braces. For example, Figure 5–8 shows how you would reformat an NCS 1.5.1 **uuid_$t** initialization to initialize a DCE RPC **uuid_t**. The arrows indicate the brace that is moved.

NCS 1.5.1 initialization:

```
{ 0x5c6da328, 0xa000, 0x0000, 0x0d, \
          {0x00, 0x04, 0x9e, 0x88, 0x00, 0x00, 0x00} }
```

DCE RPC initialization:

```
{ 0x5c6da328, 0xa000, 0x0000, 0x0d, 0x00, \
               {0x04, 0x9e, 0x88, 0x00, 0x00, 0x00} }
```

Figure 5–8. Changing the NCS 1.5.1 UUID C Initialization

Another solution is to convert the NCS 1.5.1 UUID to a string representation (using **uuid_$encode**) and pass that string to **uuid_from_string** to generate the same UUID in the DCE RPC format.

You may have programs that must use UUIDs in both NCS 1.5.1 and DCE RPC format. In programs attempting to use both **uuid_$t and uuid_t**, a DCE RPC **uuid_t** can be passed where an NCS 1.5.1 **uuid_$t** is expected and vice versa.

However, using an NCS 1.5.1 **uuid_$string_t** in a DCE RPC routine can lose information. This is because the string representation of an NCS 1.5.1 UUID does not represent all the fields that are represented in the DCE RPC UUID string format.

Specifically, the NCS 1.5.1 UUID string has 28 hexadecimal characters and 8 periods; the reserved field (which is all 0s) is not represented:

```
54c2c718f000.0d.00.01.e1.e9.00.00.00
```

The DCE RPC UUID string format has 32 hexadecimal characters and 4 dashes; all fields are represented:

```
847ec9f0-9203-11ca-8e74-08001e01e1e9
```

DCE RPC runtime routines accept UUID strings in either format.

Chapter 6

Writing Servers with Multiple Managers

DCE and NCS both allow you to write distributed applications that handle different types of objects. One server can implement an interface for several object types. You provide a separate manager implementation for each combination of interface and object type, and identify each manager with an object type UUID.

What you choose to be an object depends on your application. In DCE and NCS, an object is simply something that is identified by a UUID and that can be acted on through one or more sets of operations, called interfaces. Every object has a type, also identified by a UUID. All objects of a given type can be acted on through the same sets of interfaces. The application code that implements an interface for a particular object type is called a manager.

Writing servers with multiple managers in DCE RPC is similar to writing them in NCS 1.5.1. To illustrate, this chapter describes the DCE RPC version of the NCS 1.5.1 **stacks** program, in which a server manages two types of stacks, one based on lists and one based on arrays. Appendix B shows the NCS 1.5.1 version of the **stacks** example, and both the NCS 1.5.1 and DCE RPC versions are available online as part of the HP DCE Application Development Tools.

6.1 The stacks Interface Definition

Figure 6–1 shows the **stacks.idl** interface definition file. We created it by making minor changes to the **stacks_v2.idl** file generated by the **nidl_to_idl** tool from the NCS 1.5.1 **stacks.idl** file. (Chapter 4 describes the **nidl_to_idl** tool.)

```
[uuid(4438675B-F000-0000-0D00-00FEDA000000), version(2)]
interface stacks
{
    [idempotent]
        void stacks$init([in] handle_t h);

    /* stack functions return non-zero on error, zero otherwise */

        long stacks$push([in] handle_t h, [in] long value);

        long stacks$pop([in] handle_t h, [out] long *value);
}
```

Figure 6–1. The DCE RPC stacks/stacks.idl Interface Definition

The Makefile provided with the sample application specifies the **+e** C compiler flag, which allows the use of $ (the dollar sign) as a valid identifier character when compiling in ANSI C mode. This feature was used to minimize changes to the NCS 1.5.1 versions of the programs.

The **stacks** interface definition defines the operations that each manager will implement. Each operation has the same signature, but different implementations, in the two managers. All implementation details are in the manager module for each type, "hidden" from the interface.

When registering multiple managers, you must compile your interface definition with the IDL **–no_mepv** compiler option. The **–no_mepv** switch tells the IDL compiler not to create a manager EPV for the specified interface definition. You are supplying your own manager EPVs. (By default, the IDL compiler automatically generates a manager EPV in the stub file using the names of the operations in the IDL file.)

6.2 Generating the Object UUIDs in the stackdf.h File

Each manager must be associated with an object type UUID. We could have preserved our existing NCS 1.5.1 UUIDs, as described in Section 5.9, but in this case we chose to generate new UUIDs in the DCE UUID format. We use the **uuidgen** program with the **–s** option to generate a new UUID.

```
% uuidgen -s
= { /* 6d42cd38-8bad-11ca-b45b-08001e00d2f3 */
        0x6d42cd38,
        0x8bad,
        0x11ca,
        0xb4,
        0x5b,
        {0x08, 0x00, 0x1e, 0x00, 0xd2, 0xf3}
    }
```

As in the NCS 1.5.1 version of this program, we define macros for the C initializations of the UUIDs for the two kinds of stack objects and their types in the **stacksdf.h** header file, shown in Figure 6–2. The major difference between the two **stacksdf.h** header files is in the format of the UUID initializations. DCE RPC UUIDs have the last six bytes. rather than the last seven bytes, enclosed in braces.

```
/* the two stack objects and their types */

/* the array-based object */
/* V2.0 UUID -- 6d42cd38-8bad-11ca-b45b-08001e00d2f3 */
#define ASTACK   {0x6d42cd38, 0x8bad, 0x11ca, 0xb4, 0x5b, \
                                {0x08, 0x00, 0x1e, 0x00, 0xd2, 0xf3}}

/* V2.0 UUID -- f82f30d0-8bac-11ca-b8c1-08001e00d2f3 */
#define ASTACKT  {0xf82f30d0, 0x8bac, 0x11ca, 0xb8, 0xc1, \
                                {0x08, 0x00, 0x1e, 0x00, 0xd2, 0xf3}}

/* the list-based object */
/* V2.0 UUID -- 1fccea88-8bae-11ca-bb38-08001e00d2f3 */
#define LSTACK   {0x1fccea88, 0x8bae, 0x11ca, 0xbb, 0x38, \
                                {0x08, 0x00, 0x1e, 0x00, 0xd2, 0xf3}}

/* V2.0 UUID -- 362669f8-8bae-11ca-a1c4-08001e00d2f3 */
#define LSTACKT  {0x362669f8, 0x8bae, 0x11ca, 0xa1, 0xc4, \
                                {0x08, 0x00, 0x1e, 0x00, 0xd2, 0xf3}}
```

Figure 6–2. The stacks/stacksdf.h Header File

6.3 The stacks Client Module

The **stacks** client is built from **client.c**, **util.c** (used to generate the text of error messages), and the client stub generated by the IDL compiler, **stacks_cstub.c**. The DCE RPC version of the **util.c** module is shown in Section 6.5.

The **client.c** program, shown in Figure 6–3, lets the user access both types of stacks within one session; it maintains a separate binding handle for each stack. Both the binding handles and UUIDs for the stack types are kept in arrays.

When the client program attempts to make a remote procedure call, the object UUID in the binding handle determines the stack to be accessed. In this example we use a partially bound handle to initiate the remote procedure calls. Since the server has previously registered its object UUIDs in the local endpoint map database, the **rpcd** forwarding mechanism uses the object identifier in a partially bound handle, together with the interface UUID, to route the packet to the server.

In the DCE RPC version of this program, NCS 1.5.1 **socket_$** calls have been replaced with DCE RPC string binding manipulation routines, and the client uses a command-line argument to provide the protocol sequence and host ID of the server, instead of the Location Broker routines used in the NCS 1.5.1 version. The program also illustrates the use of the TRY/CATCH exception-handling mechanisms.

```
#include <stdio.h>
#include <dce/dce_error.h>
#include <dce/pthread_exc.h>
#include <dce/rpcexc.h>
#include <dce/uuid.h>
#include "stacks.h"
#include "stackdf.h"

#define NUM_UUIDS 2  /* Number of object UUIDs */

ndr_char nils[] = "";
extern char *error_text();

void main(int argc, char *argv[])        /* Get host from command line */
{
    /*  array of uuids
    */
    static uuid_t           objects[2] = {ASTACK, LSTACK};
    static unsigned_char_t  *obj_strs[2];
    rpc_binding_handle_t    handle[2];          /* binding handle */
    ndr_char                *string_binding;    /* New string binding */
    unsigned32              st;
    ndr_long_int            val;
    char                    command[100], which[100], value[100];
    int                     i, s;


    if (argc != 3)     /* Get command line args */
    {
        fprintf(stderr, "Usage: %s ProtocolSequence HostID \n", argv[0]);
        exit(1);
    }


    for (i=0; i<2; i++) /* While less than number of objects */ {

        /* Convert object UUID to its string representation */

        uuid_to_string (&objects[i], &obj_strs[i], &st);
        if (st != uuid_s_ok) {
            fprintf(stderr, "Cannot compose uuid string %s\n", error_text(st));
            exit(1);
        }
```

Figure 6–3. The stacks/client.c Module

```
/*Build string binding from command line input */

    rpc_string_binding_compose(
        obj_strs[i], (ndr_char *)argv[1], (ndr_char *)argv[2],
        nils, nils, &string_binding, &st
    );
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot compose string binding %s\n", error_text(st));
        exit(1);
    }

    /* Create binding handle from string  */

    rpc_binding_from_string_binding(string_binding, &handle[i], &st);
    if (st != rpc_s_ok) {
        fprintf(
            stderr, "Cannot convert name \"%s\" to binding - %s\n",
            string_binding, error_text(st)
        );
        exit(1);
    }
    printf("%d-th binding is %s\n", i, string_binding);
} /* for loop */

/* Free string binding when done */

rpc_string_free(&string_binding, &st);
if (st != rpc_s_ok) {
    fprintf(
        stderr, "Cannot free string binding %s - %s\n",
        string_binding, error_text(st)
    );
    exit(1);
}

/* Free uuid strings */

for (i=0; i<2; i++) { /* While less than number of objects */
    rpc_string_free(&obj_strs[i], &st);
    if (st != rpc_s_ok) {
        fprintf(
            stderr, "Cannot free string binding %s - %s\n",
            string_binding, error_text(st)
        );
        exit(1);
    }
}
```

Figure 6–3. The stacks/client.c Module (Continued)

```
printf("Initialize stack objects (y/n)? ");
gets(command);

if (*command != 'n' && *command != 'N') {
    TRY {
        stacks$init(handle[0]);
        stacks$init(handle[1]);
    }
    CATCH (rpc_x_comm_failure) {
        fprintf (stderr, "Call failed: communication failure\n");
        exit(1);
    }
    CATCH_ALL {
        fprintf (stderr, "Call failed\n");
        exit(1);
    }
    ENDTRY;
}

do {
    printf("push, pop, or quit: ");
    gets(command);

    if (!strcmp(command, "quit"))
        break;

    printf("astack or lstack: ");
    gets(which);

    if (!strcmp(which, "astack"))
        s = 0;
    else
        s = 1;

    if (!strcmp(command, "push")) {
        printf("value: ");
        gets(value);
        val = (ndr_long_int)atoi(value);
        printf("Pushing %d onto %s...", val, s?"lstack":"astack");
        TRY {
            if (stacks$push(handle[s], val))
                printf("stack full!\n");
            else
                printf("successful\n");
        }
```

Figure 6–3. The stacks/client.c Module (Continued)

```
            CATCH (rpc_x_comm_failure) {
                fprintf (stderr, "Call failed: communication failure\n");
                exit(1);
            }
            CATCH_ALL {
                fprintf (stderr, "Call failed\n");
                exit(1);
            }
            ENDTRY;
        } /* push */

        else if (!strcmp(command, "pop")) {
            printf("Popping off of %s...", s?"lstack":"astack");
            TRY {
                if (stacks$pop(handle[s], &val))
                    printf("nothing on stack!\n");
                else
                    printf("value is %d\n", val);
            }
            CATCH (rpc_x_comm_failure) {
                fprintf(stderr, "Call failed: communication failure\n");
                exit(1);
            }
            CATCH_ALL {
                fprintf(stderr, "Call failed\n");
                exit(1);
            }
            ENDTRY;
        } /* pop */

    } while (strcmp(command, "quit"));

}
```

Figure 6–3. The stacks/client.c Module (Continued)

6.4 The stacks Server Module

The **server.c** program, shown in Figure 6–4, declares two manager EPVs as external variables. Each EPV is defined in its own manager module. The **stacks** server is built from **server.c**, **util.c**, **stacks_sstub.c** (the server stub), and the two manager modules, **amanager.c** and **lmanager.c**.

In the NCS 1.5.1 version of **stacks**, we use the **rpc_$register_mgr** routine to register each manager with the RPC runtime library and the **rpc_$register_object** routine to tell the runtime library what the type of each object supported by the server is.

In the DCE version of **stacks**, we use the **rpc_server_register_if** routine to register each manager with the RPC runtime library. The second and third arguments to this routine are the type UUID of the object and the unique manager entry point vector (EPV) for that type. By default, **rpc_server_register_if** assumes you are registering a single implementation of the interface, and both these arguments are NULL. If you want to create multiple managers for an interface (as in this example), the server program must call **rpc_server_register_if** once for each manager, passing in a unique object type identifier and manager EPV. The server must also call **rpc_object_set_type** once for each object to associate the object UUID of the object with a single type UUID.

Before using **rpc_ep_register** to register the interface and the object UUIDs with the local endpoint map, the program constructs an object UUID vector of type **uuid_vector_t**. This data structure contains a count member followed by an array of pointers to UUIDs. Initially, the pointer array contains one element. The program must allocate more memory to hold pointers to subsequent UUIDs.

Finally, the server sets up exception-returning mechanisms to handle cleanup operations and listens for requests using **rpc_server_listen.**

```
#include <stdio.h>
#include <signal.h>
#include <dce/pthread_exc.h>
#include <dce/dce_error.h>
#include "stackdf.h"
#include "stacks.h"

#define NUM_UUIDS 2  /* Number of object UUIDs */
#define MAX_CALLS 5

/* Each manager EPV defined in its own manager module */

extern stacks_v2_0_epv_t stacks_v2_0_amanager_epv;
extern stacks_v2_0_epv_t stacks_v2_0_lmanager_epv;
char *error_text(unsigned st);

void main ()
{
    unsigned32              st;
    rpc_binding_vector_t    *bh_vector;
    uuid_vector_t           *obj_uuid_vector;
    unsigned_char_t         *string_binding;
    unsigned_char_t         *uuid_string;
    int                     i;
    sigset_t                sigset;
    pthread_t               thread;
    static uuid_t  astack = ASTACK, astackt = ASTACKT;
    static uuid_t  lstack = LSTACK, lstackt = LSTACKT;

  /* Map asynch SIGINT and SIGHUP signals to cancels to get an exception */
    sigemptyset(&sigset);
    sigaddset(&sigset, SIGINT);
    sigaddset(&sigset, SIGHUP);

    thread = pthread_self();
    pthread_signal_to_cancel_np(&sigset, &thread);
   /* Start up server on all supported protocol sequences */

    rpc_server_use_all_protseqs(MAX_CALLS, &st);
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot use protocol sequence: %s\n", error_text(st)
        );
        exit(1);
    }
```

Figure 6–4. The stacks/server.c Module

```
/* Register manager and object type for array-based stack object... */

 rpc_server_register_if(
     stacks_v2_0_s_ifspec, &astackt,
     (rpc_mgr_epv_t)&stacks_v2_0_amanager_epv, &st
 );
 if (st != rpc_s_ok) {
     fprintf(stderr, "Can't register array stack manager: %s\n", error_text(st)
     );
     exit(1);
 }

 rpc_object_set_type(&astack, &astackt, &st);
 if (st != rpc_s_ok) {
     fprintf(stderr, "Can't register array stack object: %s\n", error_text(st)
     );
     exit(1);
 }

/* Register interface manager and object type of list-based stack object */

 rpc_server_register_if(
     stacks_v2_0_s_ifspec, &lstackt,
     (rpc_mgr_epv_t)&stacks_v2_0_lmanager_epv, &st
 );
 if (st != rpc_s_ok) {
     fprintf(stderr, "Can't register list stack manager: %s\n", error_text(st)
     );
     exit(1);
 }

 rpc_object_set_type(&lstack, &lstackt, &st);
 if (st != rpc_s_ok) {
     fprintf(stderr, "Can't register list stack object: %s\n", error_text(st)
     );
     exit(1);
 }
   /* Get binding handle to register with the local endpoint database */
 rpc_server_inq_bindings(&bh_vector, &st);
 if (st != rpc_s_ok) {
     fprintf(stderr, "Cannot inquire about binding vectors: %s\n",
                     error_text(st)
     );
     exit(1);
 }
 printf("Bindings:\n");
```

Figure 6–4. The stacks/server.c Module (Continued)

```
for (i = 0; i < bh_vector->count; i++) {
    rpc_binding_to_string_binding(
        bh_vector->binding_h[i], &string_binding, &st
    );
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot get string binding - %s\n",error_text(st));
        exit(1);
    }
    printf("%s\n", (char *)string_binding);
}

rpc_string_free(&string_binding, &st);
if (st != rpc_s_ok) {
    fprintf(
        stderr, "Cannot free string binding %s - %s\n",
        string_binding, error_text (st)
    );
    exit(1);
}

/*  Allocate memory for the size of the vector plus pointers
 *  to additional UUIDs --minus the initial pointer.
 */
obj_uuid_vector = (uuid_vector_p_t)malloc(
        sizeof(uuid_vector_t) + ((sizeof(uuid_p_t)) * (NUM_UUIDS - 1))
);
if (!obj_uuid_vector) {
    fprintf(stderr, "Can't get memory for object UUIDs.\n");
    exit(1);
}

obj_uuid_vector->uuid[0] = &astack;
obj_uuid_vector->uuid[1] = &lstack;
obj_uuid_vector->count = NUM_UUIDS;
    /* Process the UUIDs and convert UUID to string */

for (i = 0; i < obj_uuid_vector->count; i++) {
    uuid_to_string(obj_uuid_vector->uuid[i], &uuid_string, &st);
    if (st != uuid_s_ok) {
        fprintf(stderr, "Cannot get object UUID - %s\n",error_text(st));
        exit(1);
    }
    printf("%s\n", (char *)uuid_string);
}
```

Figure 6–4. The stacks/server.c Module (Continued)

```
    /* Register interface and objects with endpoint database */

    rpc_ep_register(
        stacks_v2_0_s_ifspec, bh_vector, obj_uuid_vector,
        (unsigned_char_t *)"stack managers", &st
    );
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot register interface with endpoint map: %s\n",
            error_text(st)
        );
        exit(1);
    }

    TRY {
        /* listen */
        printf("Listening...\n");
        rpc_server_listen(MAX_CALLS, &st);
        if (st != rpc_s_ok)
            fprintf(stderr, "Error: %s\n", error_text(st));
    }  /* try */

    CATCH (pthread_cancel_e) {
        printf("Caught a cancel exception\n");
    }
    FINALLY {
        /* unregister */
        printf("Unregistering interface...\n");
        rpc_server_unregister_if(stacks_v2_0_s_ifspec, &astackt, &st);
        if (st != rpc_s_ok) {
            fprintf(
                stderr, "Can't unregister array manager/interface: %s\n",
                error_text(st)
            );
            exit(1);
        }
rpc_server_unregister_if(stacks_v2_0_s_ifspec, &lstackt, &st);
        if (st != rpc_s_ok) {
            fprintf(
                stderr, "Can't unregister list manager/interface: %s\n",
                error_text(st)
            );
            exit(1);
        }
```

Figure 6–4. The stacks/server.c Module (Continued)

```
        printf("Unregistering endpoint\n");
        rpc_ep_unregister(
            stacks_v2_0_s_ifspec, bh_vector, obj_uuid_vector, &st
        );
        if (st != rpc_s_ok) {
            fprintf(
                stderr, "Can't unregister endpoint: %s\n", error_text(st)
            );
            exit(1);
        }
        exit(0);
    }  /* finally */

    ENDTRY;

}  /* main */
```

Figure 6–4. The stacks/server.c Module (Continued)

A separate manager module implements the **stacks** interface for each type of stack. The **amanager.c** module, shown in Figure 6–5, manages stacks based on arrays, and the **lmanager.c** module, shown in Figure 6–6, manages stacks based on linked lists. Each manager defines a manager EPV, which specifies the names under which the **stacks** operations are implemented. Since we are linking both managers together in one server, the two implementations of each operation have different names.

```
#include "stacks.h"

void stacks$astack_init();
ndr_long_int  stacks$astack_push(), stacks$astack_pop();

extern stacks_v2_0_epv_t stacks_v2_0_amanager_epv =
    {stacks$astack_init, stacks$astack_push, stacks$astack_pop};

#define STACKSIZE    1000

static struct
{
    int head;
    ndr_long_int  values[STACKSIZE];
} the_stack;

void stacks$astack_init(h)
handle_t h;
{
    the_stack.head = STACKSIZE;
}

ndr_long_int  stacks$astack_push(h, value)
handle_t h;
ndr_long_int  value;
{
    if (the_stack.head == 0) return -1;             /* stack is full */

    the_stack.values[--the_stack.head] = value;

    return 0;
}

ndr_long_int  stacks$astack_pop(h, value)
handle_t h;
ndr_long_int  *value;
{
    if (the_stack.head == STACKSIZE) return -1;     /* stack is empty */

    *value = the_stack.values[the_stack.head++];

    return 0;
}
```

Figure 6–5. The stacks/amanager.c Module

```
#include "stacks.h"

void stacks$lstack_init();
ndr_long_int  stacks$lstack_push(), stacks$lstack_pop();

extern stacks_v2_0_epv_t stacks_v2_0_lmanager_epv =
    {stacks$lstack_init, stacks$lstack_push, stacks$lstack_pop};

#define NULL (struct node *)0
extern struct node *malloc();

static struct node{
    ndr_long_int  value;
    struct node *next;
} the_stack;

void stacks$lstack_init(h)
handle_t h;
{
    the_stack.next = NULL;
}

ndr_long_int  stacks$lstack_push(h, value)
handle_t h;
ndr_long_int  value;
{
    struct node *head = malloc(sizeof(struct node));
    if (head == NULL) return -1;              /* stack is full */
    head->value = value;
    head->next = the_stack.next;
    the_stack.next = head;
    return 0;
}

ndr_long_int  stacks$lstack_pop(h, value)
handle_t h;
ndr_long_int  *value;
{
    struct node *head = the_stack.next;
    if (head == NULL) return -1;             /* stack is empty */
    *value = head->value;
    the_stack.next = head->next;
    free(head);
    return 0;
}
```

Figure 6–6. The stacks/lmanager.c Module

6.5 The stacks util.c Module

The **stacks** client and server both use the **util.c** module to generate the text of error messages. Figure 6–7 shows the DCE RPC version of the **util.c** module.

```
#include "stacks.h"
#include <dce/dce_error.h>

char *error_text(st)
unsigned  st;
{
    static dce_error_string_t error_string;
    int inq_st;

    dce_error_inq_text (st, error_string, &inq_st);
    return ((char *) error_string);
}
```

Figure 6–7. The stacks/util.c Module

Chapter 7

Using DCE Location Services

NCS 1.5.1 includes **Location Broker** services that provide clients with information about the locations of objects and interfaces. The **Local** Location Broker is a server that maintains a database of information about objects and interfaces residing on the local host. The **Global** Location Broker maintains information about objects and interfaces throughout the network or internet. The DCE analogs to the Local Location Broker and the Global Location Broker are the interfaces to the DCE **endpoint map service** and the **name service**.

In this chapter we present the major differences between NCS and DCE location services, and summarize the steps clients and servers must implement to use the DCE services. To illustrate the use of some of the services, we refer to the examples available online as part of the HP DCE Application Development Tools, specifically the **stacks**, **string_conv**, and **lookup** applications. For more information about these examples, refer to the README files accompanying the examples. For more information about the endpoint map and name services, refer to the *OSF DCE Application Development Guide* and the *OSF DCE Application Development Reference.*

While the services performed by the NCS and DCE location services have much in common, there is one major difference of which application programmers must be aware. Local and global location services are provided in NCS 1.5.1 with a single set of calls, the **lb_$** calls. In DCE RPC, there are two sets of calls: the **rpc_ep** calls provide access to the local endpoint map database, and the **rpc_ns** calls provide access to the global name service database.

7.1 The Endpoint Map Service

An **endpoint** is the address of a specific instance of a server executing in a particular address space on a given host.The endpoint map service maintains the local endpoint map for RPC servers, looks up endpoints for RPC clients, and forwards messages to servers. These services are provided by the **rpcd** daemon. The **rpcd** daemon replaces the NCS **llbd** daemon.

Each element in the local endpoint map can contain the following:

- An interface identifier, consisting of an interface UUID and version numbers (major and minor)
- Binding information
- An optional object UUID
- An optional annotation containing up to 64 characters of user-defined information

Entries in an NCS Location Broker database contain two additional pieces of information: a flag indicating whether the object is global, and a type UUID that specifies the type of the object.This type UUID enables an application to locate entries in a GLB database by type. There is no DCE RPC equivalent for this feature.

7.1.1 Mapping NCS 1.5.1 lb_$ Calls to DCE RPC rpc_ep Routines

The NCS 1.5.1 API provides the **lb_$** calls as the interface to the Location Broker services. These calls allow clients and servers to look up, register, or unregister entries in Local or Global Location Broker databases. The **lb_$** calls handle registration of both local (local-host-only) and global (network-wide) entries. The DCE RPC **rpc_ep** calls only register entries in a local database.

Table 7–1 shows the **rpc_ep** or **rpc_mgmt_ep** calls that have an **lb_$** analog, and shows whether they are used by clients, servers, management code or all of these. As we noted above, there is no complete correspondence between **rpc_** and **lb_$** routines. Refer to the *OSF DCE Application Development Reference* for a complete description of each **rpc_** routine. In the following sections we show how some of these routines can be used in clients, servers, and management code.

Table 7–1. RPC Endpoint Map Equivalents to Location Broker Calls

DCE RPC Routine	NCS Routine	Where Used
rpc_ep_register	**lb_$register**	Server
rpc_ep_register_no_replace	None	Server
rpc_ep_resolve_binding	**lb_$lookup_interface**	All
rpc_ep_unregister	**lb_$unregister**	Server
rpc_mgmt_ep_elt_inq_begin	**lb_$lookup_object_local**	Management
rpc_mgmt_ep_elt_inq_done	**lb_$lookup_object_local**	Management
rpc_mgmt_ep_elt_inq_next	**lb_$lookup_object_local**	Management
rpc_mgmt_ep_unregister	**lb_$unregister**	Management

7.1.2 Using rpc_ep Routines in a Client

The only **rpc_ep** routine used by a client is **rpc_ep_resolve_binding**. Clients can call this routine to resolve a partially bound server binding handle into a fully bound handle. The call can also be used by servers and managers.

7.1.3 Using rpc_ep Routines in a Server

Servers can use all four **rpc_ep** routines. The extracts in Figure 7–1 are from the **server.c** program in the **stacks** example shown in Chapter 6. They illustrate how a server registers and unregisters with the endpoint map database using the **rpc_ep_register** and **rpc_ep_unregister** routines.

The **rpc_ep_register** call can be used to replace an entry in the endpoint map if only one instance of the server will run on this host. If multiple instances of a server will run on the same host, a server should use the **rpc_ep_register_no_replace** routine instead of **rpc_ep_register**. If you use **rpc_ep_register_no_replace**, it's particularly important to remember to call **rpc_ep_unregister** to remove each instance of the server before stopping.

```
        .
        .
        .
rpc_ep_register(
        stacks_v2_0_s_ifspec, bh_vector, obj_uuid_vector,
        (unsigned_char_t *)"stack managers", &st
    );
    if (st != rpc_s_ok) {
        fprintf(stderr, "Cannot register interface with endpoint map: %s\n",
            error_text(st)
        );
        exit(1);
    }
          .
          .
          .
 rpc_ep_unregister(
            stacks_v2_0_s_ifspec, bh_vector, obj_uuid_vector, &st
        );
        if (st != rpc_s_ok) {
            fprintf(
                stderr, "Can't unregister endpoint: %s\n", error_text(st)
            );
            exit(1);
          .
          .
          .
```

Figure 7–1. Registering and Unregistering with the Local Endpoint Map Database

Servers don't need to call these routines if binding handles are registered by another means. For example, the **endpoint** attribute in the client's interface specification may specify a well-known endpoint, and the server may establish a well-known endpoint with **rpc_server_use_all_protseqs_if** or **rpc_server_use_protseq_if**. Refer to the *OSF DCE Application Development Guide* or the other documents listed in the preface for more detailed information about these alternatives.

A server should call **rpc_ep_resolve_binding** to resolve a partially bound server binding handle into a fully bound handle under some circumstances. For example, the **server.c** program in the **lookup** application calls **rpc_ep_resolve_binding** to obtain the fully bound handle required by the **rpc_mgmt_is_server_listening** routine.

7.1.4 Using rpc_mgmt_ep Routines in a Manager

An alternative method of managing an endpoint map is to use the **rpc_mgmt_ep_elt_inq_** routines. These routines create, delete, and use an inquiry context that enables application code to inspect each element in turn in a local or remote endpoint map. An application-specific routine can then select one or more binding handles according to specific criteria. Management routines call **rpc_mgmt_ep_unregister** to remove an interface ID, if a server is no longer available, or to remove object UUIDs if a server no longer supports the object, from local or remote endpoint maps.

7.2 The Name Service

A name service maintains and provides access to a database of information about RPC servers, interfaces, and objects. The DCE RPC Name Service Interface (NSI) uses the Cell Directory Service (CDS) as its database. This database is maintained by **cdsd**, the CDS daemon, which plays a similar role to that played in NCS by the Global Location Broker daemon, **glbd**. There must be at least one **cdsd** daemon running on the network.

The information stored by the Cell Directory Service differs from that stored in the GLB database. There are three kinds of name service entries:

- **Server** entries store an interface identifier and binding information for an RPC server. They *do not* include an endpoint, so that, unlike entries in a Location Broker database, they are not associated with a specific server process.
- **Group** entries contain one or more **group members**, each of which is a name referring to another server or group name service entry. A group usually contains members offering the same interface or object, so that a client search for a server offering a specific service can begin with the name of a group entry representing several servers offering that service.
- **Profile** entries contain a collection of **profile elements**, each of which is a record containing an **interface identifier**, a **member name** referring to a server, group, or profile entry name, a **priority value**, and an optional **annotation** string. Profile entries enable users or administrators to construct a customized search path for a particular service.

You can create group and profile entries manually using the **rpccp** (RPC control program) command.

The names you choose for your server, group, and profile entries can be descriptive, but they must conform to certain conventions imposed by the CDS name syntax. Refer to the *OSF DCE Application Development Guide* for more information about **rpccp** and CDS.

7.2.1 Mapping of lb_$ Calls to rpc_ns Routines

Table 7–2 shows the **rpc_ns** calls that have an **lb_$** analog, and shows whether they are used by clients, servers, management code or all of these. As we noted above, there is no complete correspondence between **rpc_** and **lb_$** routines. Refer to the *OSF DCE Application Development Reference* for a complete description of each **rpc_** routine. In the following sections we show how some of the most frequently used routines can be used in clients and servers.

Table 7–2. RPC Name Service Equivalents to Location Broker Calls

DCE RPC Routine	NCS Routine	Where Used
rpc_ns_binding_export	**lb_$register**	Server
rpc_ns_binding_import_begin	None	Client
rpc_ns_binding_import_done	None	Client
rpc_ns_binding_import_next	None	Client
rpc_ns_binding_inq_entry_name	None	Client
rpc_ns_binding_lookup_begin	**lb_$lookup_interface**	Client
rpc_ns_binding_lookup_done	**lb_$lookup_interface**	Client
rpc_ns_binding_lookup_next	**lb_$lookup_interface**	Client
rpc_ns_binding_select	None	Client
rpc_ns_binding_unexport	**lb_$unregister**	Server
rpc_ns_entry_expand_name	None	All
rpc_ns_entry_object_inq_begin	**lb_$lookup_object**	All
rpc_ns_entry_object_inq_done	**lb_$lookup_object**	All
rpc_ns_entry_object_inq_next	**lb_$lookup_object**	All
rpc_ns_group_delete	None	All
rpc_ns_group_mbr_add	None	All
rpc_ns_group_mbr_inq_begin	None	All
rpc_ns_group_mbr_inq_done	None	All

Table 7–2. RPC Name Service Equivalents to Location Broker Calls (cont.)

DCE RPC Routine	NCS Routine	Where Used
rpc_ns_group_mbr_inq_next	None	All
rpc_ns_group_mbr_remove	None	All
rpc_ns_mgmt_binding_unexport	None	Management
rpc_ns_mgmt_entry_create	None	Management
rpc_ns_mgmt_entry_delete	None	Management
rpc_ns_mgmt_entry_inq_if_ids	None	All
rpc_ns_mgmt_handle_set_exp_age	None	All
rpc_ns_mgmt_inq_exp_age	None	All
rpc_ns_mgmt_set_exp_age	None	All
rpc_ns_profile_delete	None	All
rpc_ns_profile_elt_add	None	All
rpc_ns_profile_elt_inq_begin	None	All
rpc_ns_profile_elt_inq_done	None	All
rpc_ns_profile_elt_inq_next	None	All
rpc_ns_profile_elt_remove	None	All
rpc_ns_set_authn	None	All

7.3 Client Name Service Routines

The Name Service Interface provides routines that enable a client to look up binding information using any of the name service database entries listed in Section 7.2: server, group, or profile entries.

7.3.1 Importing Binding Handles

A typical client uses three name service routines to obtain binding information:

- **rpc_ns_binding_import_begin**
- **rpc_ns_binding_import_next**
- **rpc_ns_binding_import_done**

One of the sample applications included in the HP DCE Programmers' Environment, the **string_conv** application, illustrates how these calls are used. We show only the **rpc_ns** routines and the associated **rpc_binding_free** call from the **client.c** program here, in Figure 7–2. The complete code is available online.

The string conversion client program makes two **rpc_ns_binding_import_begin** calls. The first call uses an entry name in the local Cell Directory Service as its second parameter. The **client.c** program includes code to obtain this name if a local entry exists. If there is no local server available, the next call to **rpc_ns_binding_import_begin** has NULL as its second parameter. This means it will use the default profile element defined in RPC_DEFAULT_ENTRY to begin the search for a suitable server.

The RPC_DEFAULT_ENTRY environment variable specifies the default name service entry with which clients begin searches for binding information. Please refer to the *OSF DCE Application Development Guide* and *OSF DCE Application Development Reference* for more information about environment variables provided by the NSI.

The **rpc_ns_binding_import_next** call uses the import context obtained by the **rpc_ns_binding_import_begin** call to obtain, from the name service database, binding information for a server that will support the client. For every call a client makes to **rpc_ns_binding_import_next**, you must provide a call to **rpc_binding_free** to release the memory holding the binding information and set the binding handle to NULL.

The **rpc_ns_binding_import_done** routine *must* be called for every **rpc_ns_binding_import_begin** call when the client has completed its search in the name service database. This routine frees the space allocated for the import context by the corresponding **rpc_ns_binding_import_begin** call.

```
    .
    .
    .
/* Get the current DCE host name.  This hostname will be relative to the
 * root of the cell, so it will be a string "hosts/<hostname>", where
 * hostname is your DCE host name.  The first parameter is a string that
 * is allocated to hold the return value.  The application must free
 * this string.  The second parameter is the DCE error status.
 */
dce_cf_get_host_name(&dce_host_name, &st);
if (st != rpc_s_ok) {
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE, "Cannot get DCE hostname: %s\n",
      dce_err_string);
   exit(1);
}

/* Construct a pathname to the local server using the directory entry
 * name defined in common.h and the prefix "/.:/" to define this as a
 * cell-relative entry name.
 */

sprintf(cds_entry_name, "/.:/%s/%s", dce_host_name, str_conv_cds_entry);

/* Free the dce_host_name string allocated by dce_cf_get_host_name. */

free(dce_host_name);

/*    rpc_ns_binding_import_begin() --
 *
 * Contact the directory service to determine the location information
 * for the server.  This call caches information from the directory
 * entry named into a local database.  We can then walk through the
 * cached information in the following loop.
 * /

rpc_ns_binding_import_begin(rpc_c_ns_syntax_default, /* name syntax */
          (unsigned_char_t *)cds_entry_name,
          string_conv_v1_0_c_ifspec,
          NULL,      /* No object UUID required */
          &import_context, /* used by import_next() */
          &st);      /* error status for this call */
if (st != rpc_s_ok) {
```

Figure 7–2. Name Service Routines in the string_conv/client.c Program

```
#ifdef DEBUG
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
   PRINT_FUNC(PRINT_HANDLE, "Could not import bindings from %s:\n\t%s\n",
         cds_entry_name, dce_err_string);
#endif /* DEBUG */

   /*
    * This search was unsuccessful, meaning either there is no
    * directory entry here or there are no server bindings located in
    * the directory entry.  Now look in the default profile element for
    * a matching interface UUID.
    */

   rpc_ns_binding_import_begin(rpc_c_ns_syntax_default,
                NULL, /* Use the RPC_DEFAULT_ENTRY */
                string_conv_v1_0_c_ifspec,
                NULL, /* No object UUID */
                &import_context, /* used by import_next() */
                &st); /* error status for this call */

   if (st != rpc_s_ok) {
       /*
        * Still no matching entry was found -- we must exit now since
        * we cannot locate a suitable server for this client.
        */
       dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
       PRINT_FUNC(PRINT_HANDLE, "Could not find a server: %s\n",
             dce_err_string);
       exit(1);
   }
}
    /*
     * Otherwise we have found a directory entry containing information
     * about this interface.  Now find a suitable binding for this client.
     * The following loop will continue until a successful RPC is made
     * and the flag "done" is set to true (following the RPC) OR until one
     * of the rpc_ns_ routines detects that there are no more bindings and
     * decides to break out of the loop.
     */

    done = false;
    while (done != true) {
```

Figure 7–2. Name Service Routines in the string_conv/client.c Program (Continued)

```
/*  rpc_ns_binding_import_next() --
 *
 * Attempt to import a binding for the server.  The first parameter
 * is the context returned by the import_begin call above.  The
 * second parameter is a binding handle data structure that will be
 * allocated.  This application must free the binding handle after
 * we are done with it.  The last parameter is the DCE error status.
 *
 * This call will randomly return one of the bindings found in the
 * directory (if there are more than one).  It returns the status
 * rpc_s_no_more_bindings when the bindings have been exhausted.
 */

rpc_ns_binding_import_next(import_context, /* context from begin() */
            &bh,                    /* single handle returned */
            &st);                   /* error status for this call */

if (st != rpc_s_ok) {
    if (st == rpc_s_no_more_bindings) {

    /* There are no more possible bindings to choose from.  Free
     * the binding handle allocated by import_next() and break
     * out of the while loop.
     */

    PRINT_FUNC(PRINT_HANDLE,"There are no more server bindings\n");
    rpc_binding_free(&bh, &_ignore); /* no longer needed */
    break;

} else {

    /* We did not find a compatible binding handle; instead we
     * encountered some other serious error.  Print an error
     * message and exit this client.
     */

    dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE, "Could not import a binding: %s\n",
          dce_err_string);
    exit(1);
     }
}

        .
        .
        .
```

Figure 7–2. Name Service Routines in the string_conv/client.c Program (Continued)

```
        .
        .
        .
    /* Free the binding handle allocated by rpc_ns_binding_import_next()
     * above.  The RPC has been made so it is no longer needed.
     */
        rpc_binding_free(&bh, &_ignore);
    }

    /* We have either exhausted all the server bindings in the
     * name service or completed a successful remote
     * procedure call to one of them.
     */

    /*    rpc_ns_binding_import_done() --
     *
     * Close down the association with the name server and free the space
     * allocated by DCE for the import context. The first parameter is the
     * import context; the second is the DCE error status.
     */

    rpc_ns_binding_import_done(&import_context,
                               &_ignore);   /* ignore any errors */

    exit(0);
}
```

Figure 7–2. Name Service Routines in the string_conv/client.c Program (Continued)

7.3.2 Looking up a Set of Binding Handles

The Name Service Interface provides four routines that enable clients to obtain a set of binding handles and select one or more servers from the set that meet specific criteria:

- **rpc_ns_binding_lookup_begin**
- **rpc_ns_binding_lookup_next**
- **rpc_ns_binding_lookup_done**
- **rpc_ns_binding_select**

For example, you could use the set of **lookup** routines to locate a server in a particular building or supporting specific printer capabilities. You can then use either **rpc_ns_binding_select** or an application-specific routine to select the server that meets your needs.

The Name Service Interface supplies routines with **inq** in their names (such as **rpc_ns_binding_inq_entry_name**) that can be used to obtain more information about the name service entries or set up inquiry contexts for viewing the contents of an entry.

7.4 Server Name Service Routines

The Name Service Interface provides routines that enable a server to make binding information available to clients. There are several ways to advertise a server to clients, for example by storing binding information in a database, or by printing or displaying binding information so that clients can use it. In this section we focus on the most common way of making binding information accessible: by exporting binding information and an interface specification to a name service database.

A typical server uses at least two calls to the name service database: **rpc_ns_binding_export** and **rpc_ns_binding_unexport**. These routines are analogous to using **lb_$register** and **lb_$unregister** to register and unregister servers with the Global Location Broker. Because the name service database supports group and profile entries as well as server entries, a server may also call **rpc_ns_group** or **rpc_ns_profile** routines to add the server entry to a group or profile and later delete it.

The extracts from the **server.c** program in the **string_conv** sample application in Figure 7–3 illustrate how to use the **rpc_ns_binding_export** and **rpc_ns_binding_unexport** routines to register and unregister a server in the name service database under a Cell Directory Service entry name obtained earlier in the program. The extracts also show the code used to add and later remove the entry from a profile element in the name service database using **rpc_ns_profile_elt_add** and **rpc_ns_profile_elt_remove**. The complete code for this program is available online.

```
        .
        .
        .

/*
 * Get the current cell-relative DCE host name and construct a CDS entry
 * name from it.  This is where the server will register its bindings.
 */

dce_cf_get_host_name(&dce_host_name, &st);
if (st != rpc_s_ok) {
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
    PRINT_FUNC(PRINT_HANDLE, "Cannot get DCE hostname: %s\n",
      dce_err_string);
   exit(1);
}

sprintf(cds_entry_name, "/.:/%s/%s", dce_host_name, str_conv_cds_entry);
free(dce_host_name);

/*    rpc_ns_binding_export() --
 *
 * Export the binding vector and interface specification to the name
 * server.  Register in the name service under the host-specific entry
 * name just computed above.  The first parameter is the syntax to use;
 * in the first release of DCE there is only one supported syntax.  The
 * second parameter is the entry name to look under; it was created
 * above.  The third parameter is the server interface specification,
 * with the UUID from the IDL file.  The fourth parameter is used to
 * specify an object UUID if the server exports multiple objects; this
 * server does not export multiple objects, so NULL is used.
 *
 * NOTE: If multiple servers export bindings to the same entry name,
 * only the last one will be heeded (the first server will become
 * unreachable).  However, if the first server were to terminate and
 * unregister itself, it would unregister the endpoint and CDS
 * information for the second server as well, making it now unreachable.
 * To avoid this in your applications see the code in the lookup sample.
 */

rpc_ns_binding_export(rpc_c_ns_syntax_default, /* default name syntax */
        (unsigned char *)cds_entry_name, /* created above */
        string_conv_v1_0_s_ifspec, /* IDL-generated ifspec */
        bvec,              /* this server's bindings */
        NULL,              /* No object UUID */
        &st);              /* error status for this call */
```

Figure 7–3. Name Service Routines in the string_conv/server.c Program

```
if (st != rpc_s_ok) {
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
   PRINT_FUNC(PRINT_HANDLE,"Cannot export bindings in NSI under %s: %s\n",
      cds_entry_name, dce_err_string);
   exit(1);
}

/*    rpc_if_inq_id() --
 *
 * Get the interface identifier for an interface specification.  This
 * identifier is required by the call to add the server entry above to
 * the profile element done below.  The first parameter is the server's
 * interface specification.  The second is a return parameter, the
 * interface identifier.  The final parameter is the DCE error status.
 */

rpc_if_inq_id(string_conv_v1_0_s_ifspec,  /* IDL-generated if spec. */
     &if_id,        /* fills in the interface ID */
     &st);             /* error status for this call */
if (st != rpc_s_ok) {
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
   PRINT_FUNC(PRINT_HANDLE,"Cannot determine interface id: %s\n",
      dce_err_string);

/*
 * Note: This will leave the server entry in CDS.
 */

   exit(1);
}

/*
 * Get the value of the user's RPC_DEFAULT_ENTRY environment variable.
 * Use this to determine the default (profile) name to register this
 * server with.  If the user does not have that variable set, print an
 * error message and exit.  The RPC_DEFAULT_ENTRY must be set for the
 * client to function properly.
 */

cds_profile_name = getenv("RPC_DEFAULT_ENTRY");
if (cds_profile_name == NULL) {
   PRINT_FUNC(PRINT_HANDLE,
            "Error: RPC_DEFAULT_ENTRY not set.  Set it and try again\n");
   exit(1);
```

Figure 7–3. Name Service Routines in the string_conv/server.c Program (Continued)

```
} else {

/*
 * Make a copy of the profile name string using malloc.  The
 * function getenv() may return a pointer to static data -- this
 * data is overwritten on each call, so we make sure to save the
 * data before calling getenv() again in this application.
 */

   cds_profile_name = (char *)strdup(cds_profile_name);
}

/*    rpc_ns_profile_elt_add() --
 *
 * Add the entry just exported to the name service as an element of a
 * profile.  This enables clients to look up this server using either
 * the entry name unique to this server or just the profile name.  The
 * profile can contain bindings for multiple instances of this service
 * registered under different entries in the name space.
 *
 * The first parameter is the name syntax to use for the profile element
 * (default since there is only one valid syntax).  The second parameter
 * is the profile name in which to register this server's entry.  The
 * third parameter is the interface identifier retrieved earlier.  The
 * fourth parameter is the name syntax to use for the server entry name
 * (default again).  The fifth parameter is the actual server entry name
 * registered in CDS above.  The sixth parameter is a priority value, an
 * integer between 0-7 that determines in which order bindings will be
 * returned in a lookup operation.  The seventh parameter is a string
 * annotation useful when browsing in the name space (though DCE RPC
 * does not use this string during lookup or import operations, or for
 * enumerating profile elements).  The last parameter is the DCE status.
 */

rpc_ns_profile_elt_add(rpc_c_ns_syntax_default,  /* syntax of profile name */
        (unsigned char *)cds_profile_name,              /* profile name */
        &if_id,                              /* interface id from ifspec */
        rpc_c_ns_syntax_default,                /* syntax of entry name */
        (unsigned char *)cds_entry_name,                /* entry name */
        my_profile_element_priority,             /* element's priority */
        (unsigned char *)str_conv_description,
        &st);                                /* error status for this call */
```

Figure 7–3. Name Service Routines in the string_conv/server.c Program (Continued)

```
if (st != rpc_s_ok) {
   dce_error_inq_text(st, dce_err_string, (int *)&_ignore);
   PRINT_FUNC(PRINT_HANDLE,"Cannot add entry %s to CDS profile %s: %s\n",
            cds_entry_name, cds_profile_name, dce_err_string);

/*
 * Note: This will leave the server entry in CDS.
 */

   exit(1);
}
      .
      .
      .

/*  rpc_ns_profile_elt_remove() --
 *
 * Remove this entry from the profile element we registered it in.
 * The parameters are the same as for the profile_elt_add().
 */

    rpc_ns_profile_elt_remove(rpc_c_ns_syntax_default, /* profile syntax */
              (unsigned char *)cds_profile_name,
              &if_id,                  /* interface id from ifspec */
              rpc_c_ns_syntax_default,          /* entry syntax */
              (unsigned_char_t *)cds_entry_name,
              &_ignore);                      /* ignore any errors */

/*  rpc_ns_binding_unexport() --
 *
 * Unregister this service from the namespace.  See binding_export
 * for parameter description.
 *
 * NOTE: If another server is running on this host and is registered
 * in the same server entry, this will wipe out that information.
 */

rpc_ns_binding_unexport(rpc_c_ns_syntax_default,          /* default syntax */
             (unsigned char *)cds_entry_name,
             string_conv_v1_0_s_ifspec,                  /* IDL ifspec */
             NULL,                                      /* No object UUID */
             &_ignore);                               /* ignore any errors */

      .
      .
      .
```

Figure 7–3. Name Service Routines in the string_conv/server.c Program (Continued)

7.5 The lookup Sample Application

In the sections above we use the string conversion sample application to illustrate how clients and servers use the Name Service Interface. Another sample application, the **lookup** application, incorporates more DCE RPC features, including the client's use of automatic binding to contact the server.

The **lookup** application uses the HP Name Service Access (NSA) utilities, a set of routines that are provided with the sample applications in the **../ns** directory. The NSA routines are used to export and retrieve server bindings from CDS. For example, the **server.c** program uses the NSA routine **nsa_export_service** to register the server's interface with the RPC runtime and with CDS, and to register the CDS entry in the appropriate host and cell profiles.

For more information about the **lookup** application and the Name Service Access routines, refer to the **README** files in the **lookup** and **ns** subdirectories in the examples directory.

Appendix A

DCE RPC Routines

This appendix lists the routines defined in each part of the DCE RPC API. Section A.10 contains lists of the routines according to the kind of program (server, client, management application) in which they are used. For complete information on these routines, see the *OSF DCE Application Development Reference.*

The DCE RPC API can be grouped into the following services:

Authentication Services	Provide routines that support authenticated communication between clients and servers.
Communications Services	Provide all services related to establishing a binding. It further categorizes its services into those for binding, the interface, network, object UUIDs, and servers.
Endpoint Map Services	Provide routines for registering with and updating the local endpoint map.
Error Services	Provide a routine for handling status information from all other DCE RPC routines.
Management Services	Provide local management routines that an application can use to manage itself, and local/remote management routines that can be called by a remote application wanting to manage the application.

Name Services	Provide import routines for adding binding information to a global name-service database and export routines to lookup and return binding information to clients.
String Services	Provide a string routine that client, server, and management applications can use to free a character string allocated by the DCE RPC runtime.
Stub Support Services	Provide routines that permit applications to use the Stub Memory Management Scheme.
UUID Services	Provide routines that allow client, server, and management applications to manipulate UUIDs.

Some of the routines in the DCE RPC API depend on other DCE components; for example, the routines in the authentication services and naming services depend on the DCE Security and Naming components, respectively.

A.1 Authentication Services

DCE RPC supports authenticated communication between clients and servers. Authenticated RPC works with the authentication and authorization services supplied by the DCE Security component.

Two management routines **rpc_mgmt_inq_dflt_protect_level** (descrbied in Section A.5.1) and **rpc_mgmt_inq_server_princ_name** (described in Section A.5.2.1) are also associated with authentication.

A server application makes itself available for authenticated communication by calling the **rpc_server_register_ auth_info** routine, described in Section A.2.5, to register its principal name and the authentication service that it supports.

A client application must establish the authentication service, protection level, and authorization service that it wishes to use in its communication with a server. A client application identifies the intended server by its principal name.

rpc_binding_inq_auth_client
A server application uses this routine to return authentication and authorization information from the binding handle for an authenticated client.

rpc_binding_inq_auth_info A client application uses this routine to return authentication and authorization information from a server binding handle.

rpc_binding_set_auth_info A client application uses this routine to set authentication and authorization information into a server binding handle.

rpc_mgmt_set_authorization_fn
A server application uses this routine to establish an authorization function for processing remote calls to a server's management routines.

rpc_ns_set_authn A client, server, or management application uses this routine to turn authentication on and off for RPC name service routines.

A.2 Communication Services

This section describes the routines provided by the DCE RPC Communication Services. These routines are further divided into the following:

- Binding (for string representation of binding handles)
- Interface
- Network
- Object UUID
- Server

A.2.1 Binding Routines

The routines described in this section permit you to manage bindings.

A.2.1.1 Create a Binding Handle from a String Binding

Use the following routine to create binding handles from string bindings.

rpc_binding_from_string_binding
: Client and server applications use this routine to create a server binding handle from a string representation of a binding handle. If the specified binding information does not contain an endpoint, this routine returns a **partially bound** handle. If the specified binding information contains an endpoint, the routine returns a **fully bound** handle.

A.2.1.2 Release a Binding Handle

Once you have finished with a binding handle, you release it using either of the following routines.

rpc_binding_free
: Client and server applications use this routine to release memory used by a server binding handle and the referenced binding information that was dynamically created. Applications call this routine when they have finished with a binding handle.

rpc_binding_vector_free
: Client and server applications use this routine to release memory used to store a vector of server binding handles.

A.2.1.3 Copy a Binding Handle

Use the following routine to create a copy of a binding handle.

rpc_binding_copy
: Client and server applications use this routine to copy the specified server binding information and return a new handle associated with the copied information. Use this routine to

prevent changes made to binding information in a single thread from affecting other threads.

A.2.1.4 Change a Server Binding

Use the following routine to change a server binding.

rpc_binding_reset A client calls this routine to disassociate a particular server instance from the specified server binding handle. This routine removes the endpoint portion of the server address, but the host remains unchanged, thus creating a partially-bound handle. When the client makes the next remote procedure call using this partially-bound handle, the DCE RPC runtime uses a well-known endpoint or uses the DCE RPC daemon (**rpcd**) to get an endpoint from the endpoint database. Applications may use this routine, for example, to rebind to a new server instance.

If a server wants to be available to clients who use this reset routine, it must register its binding handle in the endpoint map.

A.2.1.5 Convert a Binding Handle

Use the following routines to perform conversions on string binding handles.

rpc_binding_to_string_binding
Client and server applications use this routine to convert a client or server binding handle to its string representation. To parse the returned handle, use **rpc_string_binding_parse**.

rpc_string_binding_compose
Client and server applications use this routine to combine string binding handle components into a string binding handle.

rpc_string_binding_parse Client and server applications use this routine to parse a string representation of a binding handle into its component fields (object UUID, protocol sequence, network address, endpoint, and network options).

A.2.1.6 Get Binding Information

Use the following routines to get specific binding information.

rpc_binding_inq_object Client and server applications use this routine to view the object UUID associated with a client or server binding handle.

rpc_binding_inq_auth_client
A server application calls this routine to obtain the principal name or privilege attributes of the authenticated client that made a remote procedure call.

rpc_binding_inq_auth_info
A client application calls this routine to view the authentication and authorization information associated with a server binding handle that it previously specified by calling the **rpc_binding_set_auth_info** routine.

A.2.1.7 Set Binding Information

Use the following routines to set specific binding information.

rpc_binding_set_auth_info A client application calls this routine to set up a server binding handle for making authenticated remote procedure calls.

rpc_binding_set_object Client and server applications use this routine to associate an object UUID with a server binding handle.

A.2.1.8 Convert a Client Binding Handle

Use the following routine in manager applications to convert a client binding handle to a server binding handle.

rpc_binding_server_from_client
A server application calls this routine to convert a client binding handle to a server binding handle. The DCE RPC runtime creates and provides a *client* binding handle to a called server procedure (server manager routine). If the server wants to respond with a remote procedure call to the client, the called server must first call this routine to get a server binding handle.

A.2.2 Interface Routines

Use the following routines for handling interface specifications.

rpc_if_inq_id Client and server applications use this routine to obtain a copy of the interface identification from the provided interface specification. The returned interface identification consists of the interface UUID and any interface version numbers specified in the IDL file.

rpc_if_id_vector_free Client, server, and management applications use this routine to release the memory used to store a vector of interface identifications. An application obtains a vector of interface identifications by calling either **rpc_ns_mgmt_entry_inq_if_ids** or **rpc_mgmt_inq_if_ids**.

A.2.3 Network Routines

Use the following routines for handling the supported protocol sequences.

rpc_network_inq_protseqs A server application calls this routine to obtain a vector containing protocol sequences supported by both the DCE RPC runtime and the operating system.

rpc_network_is_protseq_valid Client and server applications use this routine to determine if an individual protocol sequence is available for making remote procedure calls. A protocol sequence is valid if both the DCE RPC runtime and the operating system support the specified protocols.

rpc_protseq_vector_free A client or server application uses this routine to free the memory used to store a vector of protocol sequences. This routine frees the memory allocated by a **rpc_network_inq_protseqs** routine.

A.2.4 Object and Type UUID Routines

Use the following routines in servers that support several object types.

rpc_object_set_type — A server application calls this routine to assign a type UUID to an object UUID, when the server application contains multiple implementations of an interface. It calls this routine once for each different object UUID that the server supports.

rpc_object_inq_type — A server application calls this routine to obtain the type UUID of an object. If the object was previously registered with the DCE RPC runtime using the **rpc_object_set_type** routine, the registered object is returned.

rpc_object_set_inq_fn — A server application calls this routine to specify a function to determine an object's type. Use this routine if an application function privately maintains object/type registrations. With this routine, the specified inquiry function returns the type UUID of an object. The DCE RPC runtime automatically calls this function when the application calls **rpc_object_inq_type** and the object of interest was not previously registered with the **rpc_object_set_type** routine.

A.2.5 Server Routines

A server application must perform the following initialization steps:

- Register the protocol sequences that it supports with the DCE runtime library.
- Create server binding information.
- Advertise the server location so that clients can find it.
- Register the binding handles with the endpoint map.
- Finally, listen for client requests.

A.2.5.1 Register Protocol Sequences

A server application calls one or more of the **rpc_server_use_*** routines to register protocol sequences with the DCE RPC runtime. To receive remote procedure calls, a server must register at least one protocol sequence with the DCE RPC runtime.

Use one of the following routines to register a server.

rpc_server_use_all_protseqs

A server application calls this routine to register *all* of the supported protocol sequences with the DCE RPC runtime. The DCE RPC runtime creates a binding handle (one for each protocol sequence), with a dynamically generated endpoint.

rpc_server_use_all_protseqs_if

A server application calls this routine to register with the DCE RPC runtime *all* protocol sequences *and* associated well-known endpoint address information provided in the IDL file.

rpc_server_use_protseq A server application calls this routine to register *one* protocol sequence with the DCE RPC runtime. A server application can call this routine multiple times to register additional protocol sequences. The DCE RPC runtime creates a binding handle with a dynamically generated endpoint.

rpc_server_use_protseq_ep

A server application calls this routine to register with the DCE RPC runtime one protocol sequence *with* a specified endpoint address. A server application can call this routine multiple times to register additional protocol sequences and endpoints. The binding handle returned contains a well-known endpoint.

rpc_server_use_protseq_if A server application calls this routine to register with the DCE RPC runtime one protocol sequence *with* an endpoint address that is provided in the IDL file. A server application can call this routine multiple times to register additional protocol sequences for additional interfaces. The binding handle returned contains a well-known endpoint from the IDL file.

A.2.5.2 Other Server Initialization

For each protocol sequence registered, the DCE RPC runtime creates a binding through which the server receives remote procedure requests. This binding contains an endpoint which was either specified in the routine or dynamically generated by the DCE RPC runtime or operating system.

After registering protocol sequences, a server typically calls other routines in the following order:

1. **rpc_server_inq_bindings** to obtain a vector containing all of the server's binding handles.
2. **rpc_server_register_if** to register with the DCE RPC runtime those interfaces offered by the server.
3. **rpc_ep_register** or **rpc_ep_register_no_replace** to register the binding handles with the endpoint map.
4. **rpc_ns_binding_export** to place the binding handles in the name-service database so that the client can access them. (DCE RPC uses the Cell Directory Service (CDS) as the name service component.)
5. **rpc_binding_vector_free** to free the vector of server binding handles.
6. **rpc_server_listen** to begin receiving remote procedure call requests.

Use the following server routines to get information about the server binding handles.

rpc_server_inq_bindings A server application calls this routine to obtain a vector of server binding handles.

Use the following routines to register server interfaces and managers with the DCE RPC runtime.

rpc_server_register_if A server application calls this routine to register with the DCE RPC runtime each implementation of the interface. To register an interface implementation, the server provides the interface specification, which was generated by the IDL compiler, and the manager type UUID and manager entry point vector (EPV), which determines which manager routine executes when a given remote procedure call executes. (When multiple managers are registered, the DCE RPC runtime matches the UUID fot the type of caller's object with a manager's type UUID.)

rpc_server_register_auth_info
: A server application calls this routine to register an authentication service to use for authenticating remote procedure calls. A server calls this routine once for each authentication service and/or principal name the server wants to register.

rpc_server_inq_if
: A server application calls this routine to determine the manager EPV for a registered interface and manager type UUID.

rpc_server_unregister_if
: A server application calls this routine to remove the association between an interface and a manager EPV. When an interface is unregistered, the DCE RPC runtime stops accepting new remote procedure calls for that interface.

Use the following routine to have the server begin listening on the registered protocol sequences and endpoints.

rpc_server_listen
: A server application calls this routine when it is ready to process remote procedure calls. The DCE RPC runtime allows a server to simultaneously process multiple calls concurrently. The server application is responsible for concurrency control between the server manager routines since each executes in a separate thread. This routine does *not* return to the server until one of the server application's manager routines calls the **rpc_mgmt_stop_server_listening** routine, or until a client makes a remote **rpc_mgmt_stop_server_listening** call to the server.

A.3 Endpoint Map Services

The Endpoint Map Services provide routines that manipulate the local endpoint map, which is a database that lists endpoints on which servers running on the local host are listening. As part of the server initialization, the server application registers its binding handles with the DCE RPC daemon (**rpcd**), which maintains the local endpoint map.

An **endpoint** identifies a specific server instance (address space) on a host. In most cases, an endpoint is dynamically allocated when a server registers a protocol sequence by the DCE

RPC runtime or operating system and it expires when the server instance stops running. Because these endpoints change frequently (whenever a server instance starts and stops), the server must store the dynamic endpoints with its local endpoint map.

The endpoint map is managed by the DCE RPC daemon (the **rpcd** process). The **rpcd** creates and deletes database entries at the request of server and management routines and periodically removes entries that contain expired endpoints.

When a client makes a remote procedure call without specifying an endpoint (using a partially bound handle), the client's DCE RPC runtime queries the **rpcd** at the server's host for the endpoint of a compatible server instance. The server's **rpcd** attempts to match a client and server's interface and object UUIDs, and if a match is found, **rpcd** checks that the server is compatible. If the **rpcd** finds a compatible server instance, the endpoint of that instance is inserted into the partially bound handle, making it fully bound.

The endpoint map object defines operations that server applications can use to add, modify, or remove information in the local endpoint map database. It also provides a routine for client and management applications to get a fully bound handle from a partially bound handle.

rpc_ep_register

A server application uses this routine to add or replace entries in the local host's endpoint map database. Servers use this routine to reduce the likelihood that a client will be given a "stale" entry, that is, an endpoint to a non-existent server instance. Servers also use this routine to replace entries when they are certain that no other server instance will be using the endpoint map.

Servers that previously registered binding handles using the client's interface specification (rather than a dynamic allocation) do not need to call this routine.

rpc_ep_register_no_replace

A server application uses this routine to add entries to the local host's endpoint map database. It does *not* replace existing entries. Servers use this routine when multiple instances of a server will run on the same host. If a server uses this routine, it must be sure to unregister itself from the database before stopping to prevent the database from having stale data.

Servers that previously registered binding handles using the client's interface specification (rather than a dynamic allocation) do not need to call this routine.

rpc_ep_resolve_binding Client, server, and management applications use this routine to resolve a partially bound server binding handle into a fully bound binding handle. A binding handle is returned if the incoming interface UUID and object UUID match that of an endpoint database entry.

Typically, management applications use the **rpc_mgmt** routines, but they can use this routine to avoid using an **rpc_mgmt** routine with a partially bound handle.

rpc_ep_unregister A server application uses this routine to remove its own entries from the local host's endpoint map database. The server uses this routine only if it wants to explicitly remove an endpoint that it had previously registered. Removing entries from the endpoint map database makes servers unavailable to client applications that have not previously communicated with the server.

A.4 Error Services

The Error Services provides an error routine to handle status information from DCE RPC-based programs. The error routine defines an operation that client, server, and management applications can use to get the message text for a DCE RPC status code.

dce_error_inq_text Client, server, and management applications use this routine to return the null-terminated character string message for the specified status code. Before calling this routine, the application must define the environment variable NLSPATH to specify locations of the DCE message catalog files.

A.5 Management Services

DCE Management Services contain two types of routines:

- Local management routines
- Local/remote management routines

Local routines are called only by the application to manage itself. Local/remote management routines are called by the application to manage itself or by a remote application (the caller) wishing to manage the application (the callee). The remote management routines are available to callers only if the callee has previously called the **rpc_server_listen** routine to begin accepting remote procedure calls.

A.5.1 Local Management Routines

Local management routines are called only by an application to manage itself. For example, the applications can get information related to the value of the binding communications timeout in a binding handle.

rpc_mgmt_inq_com_timeout
: A client application uses this routine to view the timeout value in a server binding handle. The timeout value specifies the relative amount of time that should be spent to establish a binding to the server before giving up.

rpc_mgmt_inq_dflt_protect_level
: Client and server applications use this routine to obtain the default authentication level for a specified application. If the level is inappropriate for the client application, the client can specify a different authentication level when calling an authentication routine (**rpc_binding_set_auth_info** or **rpc_server_register_auth_info**).

  Called remote procedures within a server application use this routine to get the default authentication level for a specified service to determine whether the client is authorized before executing a remote procedure.

rpc_mgmt_set_authorization_fn

A server application calls this routine to specify an authorization function to control remote access to the server's remote management routines rather than having the DCE RPC runtime use default authorizations. The DCE RPC runtime automatically calls the function specified by this routine to control the execution of *all* management routines called by clients.

rpc_mgmt_set_cancel_timeout

A client application calls this routine to specify the amount of time for the DCE RPC runtime to wait for a server to acknowledge a cancel before orphaning a call. The application specifies a number of seconds, where a value of 0 means that the runtime orphans the remote procedure call as soon as it receives a cancel and returns to the client program immediately. This timeout value applies to the current thread; each thread must set its own timeout value explicitly.

rpc_mgmt_set_com_timeout

A client application calls this routine to change the communication timeout value for a server binding handle. This value specifies how much time the DCE RPC runtime should spend in trying to establish a relationship to the server before giving up.

rpc_mgmt_set_server_stack_size

A server application calls this routine to specify the thread stack size to use when the DCE RPC runtime creates call threads for executing remote procedure calls. It does so to ensure that each thread has an adequate stack size for handling the stack requirements of the manager routines for each interface implementation. If a server calls this routine, it must do so before calling **rpc_server_listen.**

rpc_mgmt_stats_vector_free

Client, server, or management applications use this routine to release the memory used to store statistics.

A.5.2 Local/Remote Management Routines

Local/remote management routines are called either by an application managing itself, or by a remote application (the caller) wanting to manage the application (the callee). Typically, applications use these routines to get DCE RPC runtime statistics.

A.5.2.1 Routines Used by All Applications

The following routines can be used by all DCE applications.

rpc_mgmt_inq_if_ids Client, server and management applications use this routine to obtain a vector of interface IDs listing the interfaces that a server has previously registered with the DCE RPC runtime. By default, the DCE RPC runtime allows all clients to remotely call this routine. To restrict remote calls to this routine, a server application supplies an authorization function using the **rpc_mgmt_set_authorization_fn** routine.

rpc_mgmt_inq_server_princ_name Client, server and management applications use this routine to obtain a principal name that a server registered for a specified authentication service. A client uses this routine to establish one-way authentication: the server will verify that the client is who it claims to be, but the client does not care which principal server receives the remote procedure call request. After calling this routine, a client calls **rpc_binding_set_auth_info** to set up one-way authentication.

rpc_mgmt_inq_stats Client, server and management applications use this routine to obtain statistics about the specified server from the DCE RPC runtime. Statistics include the number of remote procedure calls that a server receives and initiates and the number of network packets sent and received.

rpc_mgmt_is_server_listening Client, server and management applications use this routine to determine if the specified server is listening for remote procedure calls. By default, the DCE RPC runtime allows all

clients to remotely call this routine. To restrict remote calls to this routine, a server application can supply an authorization function by using the **rpc_mgmt_set_authorization_fn** routine.

rpc_mgmt_stop_server_listening

Client, server and management applications use this routine to direct a server to stop listening for remote procedure calls. On receiving this request, the DCE RPC runtime stops accepting new remote procedure calls for all registered interfaces. By default, the DCE RPC runtime does *not* allow any client to call this routine remotely. To allow clients to execute this routine, a server application can supply an authorization function by using the **rpc_mgmt_set_authorization_fn** routine.

A.5.2.2 Routines Used by Management Applications

The following routines are used by management applications.

rpc_mgmt_ep_elt_inq_begin

With this routine, the management application specifies which elements in the database it wants to inquire about before calling **rpc_mgmt_ep_elt_inq_next**. It can specify all elements or those elements with a specified interface ID, object UUID or both interface ID and object UUID.

rpc_mgmt_ep_elt_inq_next

Management applications use this routine to view the selected endpoint map database entries. This routine returns one element at a time, so the application calls this routine repeatedly until all entries are received.

rpc_mgmt_ep_elt_inq_done

Management applications use this routine to free an inquiry context that was previously created by the **rpc_mgmt_elt_inq_begin** routine.

rpc_mgmt_ep_unregister Management applications use this routine to remove a server address from an endpoint map database. They call this routine

when the server is no longer available or to remove addresses of servers that support objects that are no longer offered. To view the elements, the management applications call the **rpc_mgmt_ep_elt** routines above.

A.6 Name Service

To locate a server program and establish a relationship with that server, a client program must know the binding information (which includes network address and protocol information) of that server. In many cases, a client application will obtain this binding information from the global name-service database. The DCE RPC Name Service Interface (NSI) provides **import** operations, to add binding information of a service to the global name-service database, and **export** operations, to look up and return binding information to clients. The NSI also provides operations to manage DCE RPC data stored in the global name-service database.

The DCE RPC NSI is independent of any name service and supports any name service that is hierarchical and has multi-valued attributes. Name Services provide routines that operate on the following objects:

Binding objects	Allow client or server applications to obtain binding handles for compatible servers. The binding objects provide import operations, which return one binding handle at a time to a client application, and lookup operations, which return multiple binding handles allowing the client to select the appropriate handle itself.
Group objects	Allow client, server, or management applications to view, add, delete, or modify DCE RPC group entries of the name-service database.
Profile objects	Allow client, server or management applications to view, add, delete, or modify profile attributes to/from the specified name service entry.
Entry objects	Allow client or server applications to view information from a named name-service database object. The application determines whether the entry will be used as a server, object, DCE RPC group, or profile object or any combination of the above.

Management objects	Allow client, server, or management applications to view, create, delete, or modify name-service database entries.

A **group** is a collection of servers that usually offer the same interface. A group permits you to refer to a collection of servers that may reside on different systems by a single name.

A client must start its name service search from a known entry. A **profile** defines a search list for finding servers in the name service database. A profile permits you specify a single, general entry name for the starting point of all name service database searches; with profiles, clients do not need to know specific entry names. The DCE RPC-specific environment variable RPC_DEFAULT_ENTRY is usually set to a profile name. You can create a profile either manually using **rpccp** commands or in a program using **rpc_ns_profile_*** routines.

Name Service objects are Cell Directory Service objects.

For more information about NSI usage, see the *OSF DCE Application Development Guide* and the reference pages for the individual routines.

A.6.1 Export a Server to the Name Service

The NSI enables any RPC server with the necessary name service permissions to create and maintain its own server entries in a name service database. A server can use as many server entries as necessary to advertise the combinations of RPC interfaces and objects that the server provides. Servers use the **rpc_ns_binding_export** to create entries, and **rpc_ns_binding_unexport** to remove bindings.

rpc_ns_binding_export	A server application uses this routine to establish a name service database entry with binding handles or object UUIDs for a server.
rpc_ns_binding_unexport	A server application uses this routine to remove the binding handles for an interface, or object UUIDs, from an entry in the name service database.

A.6.2 Search a Name Service Database for Binding Information

A client can obtain binding information from a name service database in one of the following ways:

- Use the automatic method of binding.
- Call the import routines (**rpc_ns_binding_import_***) to obtain a binding handle for a compatible server.
- Call the lookup routines (**rpc_ns_binding_lookup_***) to obtain a list of binding handles for a compatible server and then select a binding handle.

A.6.2.1 Automatic Binding

In the automatic method of binding management the client stub transparently manages binding information. In this case, the client code does not make any calls to the NSI interface. The client must, however, specify the starting name service entry by setting the RPC_DEFAULT_ENTRY environment variable.

A.6.2.2 Import Routines

To use the import routines, first call **rpc_ns_binding_import_begin** to create an import context, then call **rpc_ns_binding_import_next** to return the handle, and last of all, call **rpc_ns_binding_input_done** to delete the import context.

rpc_ns_binding_import_begin
: A client application uses this routine to create an import context for an interface and an object in the name service database.

rpc_ns_binding_import_next
: A client application uses this routine to return a binding handle of a compatible server (if one is found) from the name service database.

rpc_ns_binding_import_done
: A client application uses this routine to delete the import context for searching the name service database.

A.6.2.3 Lookup Routines

To use the lookup routines, first call **rpc_ns_binding_lookup_begin** to create a lookup context, then call **rpc_ns_binding_lookup_next** to obtain a list of binding handles for a compatible server. Finally call **rpc_ns_binding_lookup_done** to delete the lookup context.

rpc_ns_binding_lookup_begin
: A client application uses this routine to create a lookup context for an interface and an object in the name service database.

rpc_ns_binding_lookup_next
: A client application uses this routine to return a list of binding handles of one or more compatible servers (if one is found) from the name service database.

rpc_ns_binding_lookup_done
: A client application uses this routine to delete the lookup context for searching the name service database.

Select a binding handle from the list returned by **rpc_ns_binding_lookup_next** using either **rpc_ns_binding_select** or a user-defined select routine that implements an appropriate selection algorithm.

rpc_ns_binding_select
: A client application uses this routine to return a binding handle from a list of compatible server binding handles.

A.6.3 Manage Name Service Entries

These routines manage name service entries.

A.6.3.1 Find Entries

A client application uses the following routine to find the name of an entry in the name service database for which the application has the binding handle.

rpc_ns_binding_inq_entry_name
: A client application uses this routine to return the name of an entry in the name service database from which the server binding handle came.

A.6.3.2 Create and Delete Entries

Management applications use the following routines to create and delete entries in a name service database.

rpc_ns_mgmt_entry_create A management application uses this routine to create an entry in the name service database.

rpc_ns_mgmt_entry_delete A management application uses this routine to delete an entry from the name service database.

A.6.3.3 View Objects of an Entry

Applications use the following sequence of calls to view the objects of an entry in the name service database.

rpc_ns_entry_object_inq_begin
A client, server, or management application uses this routine to create an inquiry context for viewing the objects of an entry in the name service database.

rpc_ns_entry_object_inq_next
A client, server, or management application uses this routine to return one object at a time from an entry in the name service database.

rpc_ns_entry_object_inq_done
A client, server, or management application uses this routine to delete the inquiry context for viewing the objects of an entry in the name service database.

rpc_ns_entry_expand_name
A client, server, or management application uses this routine to expand the name of a name service entry.

A.6.3.4 Get Information from Entries

Applications use these routines to get information from entries in the name service database.

rpc_ns_mgmt_binding_unexport
A management application uses this routine to remove multiple binding handles, or object UUIDs, from an entry in the name service database.

rpc_ns_mgmt_entry_inq_if_ids
A client, server, or management application returns the list of interfaces exported to an entry in the name service database.

A.6.4 Managing Name Service Groups

A group is a name service entry that corresponds to one or more RPC servers all of which offer the same RPC interface, type of RPC object, or both. A group can include server entries and other group entries. The name service run-time routines search the members of a group to find a server. A group makes it possible to store server entries from many systems under a single name. You use **rpccp** commands to create, manage, and view groups. When a server is installed, the installer creates one or more groups for the application.

A.6.4.1 Delete a Group

The following routine deletes a group.

rpc_ns_group_delete A client, server, or management application uses this routine to delete a group attribute.

A.6.4.2 Add and Remove Group Members

The following two routines add and remove group members.

rpc_ns_group_mbr_add A client, server, or management application uses this routine to add an entry name to a group; if the entry does not exist, this routine creates the entry.

rpc_ns_group_mbr_remove
A client, server, or management application uses this routine to remove an entry name from a group.

A.6.4.3 View Members of a Group

Applications use the **rpc_ns_group_mbr_inq_*** routines to view members of groups.

rpc_ns_group_mbr_inq_begin
: A client, server, or management application uses this routine to create an inquiry context for viewing group members.

rpc_ns_group_mbr_inq_next
: A client, server, or management application uses this routine to return one member name at a time from a group.

rpc_ns_group_mbr_inq_done
: A client, server, or management application uses this routine to delete the inquiry context for a group.

A.6.5 Managing Name Service Expirations

Previously requested name service data are sometimes stored on the system where the request originated. Such local copies are not automatically updated at each request; the local copy is updated only when it exceeds its expiration age. These routines permit applications to set the global expiration age, to set the expiration age for a particular handle, and to inquire about the expiration age.

rpc_ns_mgmt_set_exp_age A client, server, or management application uses this routine to modify the application's global expiration age for local copies of name service data.

rpc_ns_mgmt_handle_set_exp_age
: A client, server, or management application uses this routine to set a handle's expiration age for local copies of name service data.

rpc_ns_mgmt_inq_exp_age A client, server, or management application uses this routine to return the application's global expiration age for local copies of name service data.

A.6.6 Managing Name Service Profiles

Profiles are tools for managing NSI searches. Profiles are created by name service administrator or owners of applications. An RPC profile is an entry in the name service database that

contains a collection of profile elements. A profile element is a database record that corresponds to a single RPC interface; each profile element contains identification information for the interface and an annotation string that describes the purpose of the profile element. For detailed information about RPC profiles, see the *OSF DCE Application Development Guide* and each routine's manual page.

A.6.6.1 Delete a Profile Attribute

Use this routine to delete a profile attribute.

rpc_ns_profile_delete A client, server, or management application uses this routine to delete a profile attribute.

A.6.6.2 Add and Remove Profile Elements

Applications use **rpc_ns_profile_elt_add** and **rpc_ns_profile_elt_remove** to add and remove profile elements.

rpc_ns_profile_elt_add A client, server, or management application uses this routine to add an element to a profile. If necessary, creates the entry.

rpc_ns_profile_elt_remove A client, server, or management application uses this routine to remove an element from a profile.

A.6.6.3 Obtain Profile Elements

Applications use the **rpc_ns_profile_elt_inq_*** routines to obtain profile elements.

rpc_ns_profile_elt_inq_begin
A client, server, or management application uses this routine to create an inquiry context for viewing the elements in a profile.

rpc_ns_profile_elt_inq_next
A client, server, or management application uses this routine to return one element at a time from a profile.

rpc_ns_profile_elt_inq_done
A client, server, or management application uses this routine to delete the inquiry context for a profile.

A.7 String Services

All DCE RPC API routines that return large and/or variable-length character strings allocate the storage for the string from the heap. Applications are obligated to free this storage. The String Services provide a routine that client, server, and management applications use to free a character string allocated by the DCE RPC runtime.

rpc_string_free — Client, server and management applications use this routine to free the memory containing a null-terminated character string returned by the DCE RPC runtime. Applications must call this routine for *each* character string allocated and returned by calls to other DCE RPC runtime routines.

A.8 Stub Support Services

Stub Support Services provide routines that permit applications to use the Stub Memory Management Scheme. Since a DCE full pointer can change its value across a call, DCE applications that use full pointers may need to allocate memory for pointed-to data, and stubs must be able to manage this memory.

A.8.1 Using the Stub Memory Management Scheme

C applications normally use **malloc** and **free** to allocate and free memory that pointers refer to. Using the stub support routines **rpc_ss allocate** and **rpc_ss_free** to allocate and free memory on a server permits applications to use the Stub Memory Management Scheme. Detailed rules for using these routines are described in the *OSF DCE Application Development Guide.*

A.8.1.1 Allocate and Free Memory on a Server

These routines permit you to allocate and free memory for pointed-to data.

rpc_ss_allocate — This routine is usually used by a server application but may be used by a client application to allocate memory within the RPC stub memory management scheme. Memory that is allocated by this routine is released by the server stub after any output parameters have been marshalled by the stubs.

rpc_ss_free This routine is usually used by a server application but may be used by a client application to free memory allocated by **rpc_ss_allocate**; this routine can also be used to release storage pointed to by a full pointer.

A.8.1.2 Enable and Disable Allocation by rpc_ss_allocate

Client applications use the following routines to enable and disable allocation by **rpc_ss_allocate**.

rpc_ss_enable_allocate A client application uses this routine to enable the allocation of memory by **rpc_ss_allocate** when not in manager code.

rpc_ss_disable_allocate A client application uses this routine to release resources and allocated memory.

A.8.1.3 Establish Routines that Free and Allocate Memory

Clients use the following routine to establish the routines used in allocating and freeing memory.

rpc_ss_set_client_alloc_free
A client application uses this routine to set the memory allocation and freeing mechanism used by the client stubs, thereby overriding the default routines the client stub uses to manage memory for pointed-to nodes.

A.8.1.4 Change the Current Allocation and Freeing Mechanism

The following routine exchanges the current client allocation and freeing mechanism for one supplied in the routine. This primary purpose of this routine is to simplify writing modular routine libraries that use RPC calls.

rpc_ss_swap_client_alloc_free
A client application uses this routine to exchange the current memory allocation and freeing mechanism used by the client stubs with one supplied by the client.

A.8.2 Using Thread Handles in Memory Management

There are two situations where the control of memory management requires the use of thread handles:

- The manager thread spawns additional threads.
- A client application becomes a server application and then reverts to being a client application.

In both instances, the **rpc_ss_get_thread handle** and **rpc_ss_set_thread_handle** routines permit applications to control memory management. See the *OSF DCE Application Development Guide* for more information.

rpc_ss_set_thread_handle This routine is usually used by a server application but may be used by a client application to set the thread handle for either a newly created spawned thread or for a server that was formerly a client and is ready to be a client again.

rpc_ss_get_thread_handle This routine is usually used by a server application but may be used by a client application to get a thread handle for the manager code before it spawns additional threads, or for the client code when it becomes a server.

A.8.3 Other Memory Management Routines

The following routines perform a variety of memory management functions.

rpc_ss_client_free This routine is usually used by a server application but may be used by a client application to free memory returned from a client stub.

rpc_ss_destroy_client_context
A client application uses this routine to reclaim the client memory resources for a context handle, and sets the context handle to null.

A.9 UUID Services

The UUID Services provide routines that allow client, server, and management applications to manipulate UUIDs. Routines include: creating new UUIDs, comparing two UUIDs, creating a hash value for a UUID, converting UUIDs to strings and vice versa.

uuid_create	Client, server and management applications use this routine to create a new UUID.
uuid_create_nil	Client, server and management applications use this routine to obtain a nil-valued UUID.
uuid_compare	Client, server and management applications use this routine to determine the lexical order of two UUIDs. Applications can use this routine for sorting data using UUIDs as a key.
uuid_equal	Client, server and management applications use this routine to compare two UUIDs and determine if they are equal.
uuid_is_nil	Client, server and management applications use this routine to determine whether the specified UUID is a nil-valued UUID. This routine yields the same result as if an application called **uuid_create_nil** and then **uuid_equal**.
uuid_to_string	Client, server and management applications use this routine to convert a binary UUID to a string UUID.
uuid_from_string	Client, server and management applications use this routine to convert a string UUID to a binary UUID.
uuid_hash	Client, server and management applications use this routine to generate a hash value for a specified UUID.

A.10 DCE RPC Runtime Routine Summary

This section contains lists of DCE RPC routines according to which type of application (client, server, or management) uses them.

A.10.1 DCE RPC Client Runtime Routines

These are the DCE RPC routines used in DCE client applications.

A.10.1.1 Binding Routines

```
rpc_binding_copy
rpc_binding_free
rpc_binding_from_string_binding
rpc_binding_inq_auth_info
rpc_binding_inq_object
rpc_binding_reset
rpc_binding_set_auth_info
rpc_binding_set_object
rpc_binding_to_string_binding
rpc_binding_vector_free
rpc_string_binding_compose
rpc_string_binding_parse
```

A.10.1.2 Interface Routines

```
rpc_if_id_vector_free
rpc_if_inq_id
```

A.10.1.3 Network Routines

```
rpc_network_inq_protseqs
rpc_network_is_protseq_valid
rpc_protseq_vector_free
```

A.10.1.4 Endpoint Map Services

```
rpc_ep_resolve_binding
```

A.10.1.5 Error Services

```
dce_error_inq_text
```

A.10.1.6 Inquire of Protocol Sequences

```
rpc_network_inq_protseqs
rpc_network_is_protseq_valid
```

A.10.1.7 Local Management Services

```
rpc_mgmt_inq_com_timeout
rpc_mgmt_inq_dflt_protect_level
rpc_mgmt_set_cancel_timeout
rpc_mgmt_set_com_timeout
rpc_mgmt_stats_vector_free
```

A.10.1.8 Local/Remote Management Services

```
rpc_mgmt_inq_if_ids
rpc_mgmt_inq_server_princ_name
rpc_mgmt_inq_stats
rpc_mgmt_is_server_listening
rpc_mgmt_stop_server_listening
rpc_mgmt_stats_vector_free
```

A.10.1.9 String Services

```
rpc_string_free
```

A.10.1.10 Name Service Routines

Find Servers from a Name Service

```
rpc_ns_binding_import_begin
rpc_ns_binding_import_done
```

rpc_ns_binding_import_next
rpc_ns_binding_inq_entry_name
rpc_ns_binding_lookup_begin
rpc_ns_binding_lookup_done
rpc_ns_binding_lookup_next
rpc_ns_binding_select

Manage Name Service Entries

rpc_ns_entry_expand_name
rpc_ns_entry_object_inq_begin
rpc_ns_entry_object_inq_done
rpc_ns_entry_object_inq_next
rpc_ns_mgmt_entry_inq_if_ids

Manage Name Service Groups

rpc_ns_group_delete
rpc_ns_group_mbr_add
rpc_ns_group_mbr_inq_begin
rpc_ns_group_mbr_inq_done
rpc_ns_group_mbr_inq_next
rpc_ns_group_mbr_remove

Manage Name Service Profiles

rpc_ns_profile_delete
rpc_profile_elt_add
rpc_profile_elt_inq_begin
rpc_profile_elt_inq_done
rpc_profile_elt_inq_next
rpc_profile_elt_remove

Manage Name Service Expirations

rpc_ns_mgmt_handle_set_exp_age
rpc_ns_mgmt_inq_exp_age
rpc_ns_mgmt_set_exp_age

A.10.1.11 UUID Services

```
uuid_compare
uuid_create
uuid_create_nil
uuid_equal
uuid_from_string
uuid_hash
uuid_is_nil
uuid_to_string
```

A.10.1.12 Stub Support Routines

```
rpc_ss_allocate
rpc_ss_client_free
rpc_ss_destroy_client_context
rpc_ss_disable_allocate
rpc_ss_enable_allocate
rpc_ss_free
rpc_ss_get_thread_handle
rpc_ss_set_client_alloc_free
rpc_ss_set_thread_handle
rpc_ss_swap_client_alloc_free
```

A.10.2 DCE RPC Server Runtime Routines

These are the DCE RPC routines used in DCE server applications.

A.10.2.1 Binding Routines

```
rpc_binding_copy
rpc_binding_free
rpc_binding_from_string_binding
rpc_binding_inq_auth_client
rpc_binding_inq_object
```

rpc_binding_server_from_client
rpc_binding_to_string_binding
rpc_binding_vector_free
rpc_string_binding_compose
rpc_string_binding_parse

A.10.2.2 Interface Routines

rpc_if_id_vector_free
rpc_if_inq_id

A.10.2.3 Network Routines

rpc_network_inq_protseqs
rpc_network_is_protseq_valid
rpc_protseq_vector_free

A.10.2.4 Object UUID Routines

rpc_object_inq_type
rpc_object_set_inq_fn
rpc_object_set_type

A.10.2.5 Server Routines

rpc_if_inq_id
rpc_server_inq_bindings
rpc_server_inq_if
rpc_server_listen
rpc_server_register_auth_info
rpc_server_register_if
rpc_server_unregister_if
rpc_server_use_all_protseqs
rpc_server_use_all_protseqs_if
rpc_server_use_protseq
rpc_server_use_protseq_ep
rpc_server_use_protseq_if

A.10.2.6 Endpoint Map Services

```
rpc_ep_register
rpc_ep_register_no_replace
rpc_ep_unregister
```

A.10.2.7 Managing Binding Handles

```
rpc_binding_to_string_binding
rpc_binding_copy
rpc_binding_free
rpc_binding_inq_object
rpc_binding_vector_free
rpc_string_binding_compose
rpc_string_binding_parse
```

A.10.2.8 Error Services

```
dce_error_inq_text
```

A.10.2.9 Local Management Services

```
rpc_mgmt_inq_dflt_protect_level
rpc_mgmt_set_authorization_fn
rpc_mgmt_set_server_stack_size
rpc_mgmt_stats_vector_free
```

A.10.2.10 Local/Remote Management Services

```
rpc_mgmt_inq_if_ids
rpc_mgmt_inq_server_princ_name
rpc_mgmt_inq_stats
rpc_mgmt_is_server_listening
rpc_mgmt_stats_vector_free
rpc_mgmt_stop_server_listening
```

A.10.2.11 String Services

```
rpc_string_free
```

A.10.2.12 Managing the Server

uuid_compare
uuid_create
uuid_create_nil
uuid_equal
uuid_hash
uuid_is_nil
uuid_from_string
uuid_to_string

A.10.2.13 Name Service Routines

Export Servers to Name Service

rpc_ns_binding_export
rpc_ns_binding_unexport

Manage Name Service Expirations

rpc_ns_mgmt_handle_set_exp_age
rpc_ns_mgmt_inq_exp_age
rpc_ns_mgmt_set_exp_age

Manage Name Service Entries

rpc_ns_entry_expand_name
rpc_ns_entry_object_inq_begin
rpc_ns_entry_object_inq_done
rpc_ns_entry_object_inq_next
rpc_ns_mgmt_entry_inq_if_ids

Manage Name Service Groups

rpc_ns_group_delete
rpc_ns_group_mbr_add
rpc_ns_group_mbr_inq_begin
rpc_ns_group_mbr_inq_done
rpc_ns_group_mbr_inq_next
rpc_ns_group_mbr_remove

Manage Name Service Profiles

rpc_ns_profile_delete
rpc_profile_elt_add
rpc_profile_elt_inq_begin
rpc_profile_elt_inq_done
rpc_profile_elt_inq_next
rpc_profile_elt_remove

A.10.2.14 Stub Support Routines

rpc_ss_allocate
rpc_ss_client_free
rpc_ss_free
rpc_ss_get_thread_handle
rpc_ss_set_client_alloc_free
rpc_ss_set_thread_handle

A.10.2.15 UUID Services

uuid_compare
uuid_create
uuid_create_nil
uuid_equal
uuid_from_string
uuid_hash
uuid_is_nil
uuid_to_string

A.10.3 DCE Management Application Runtime Routines

These are the DCE RPC routines used in DCE management applications.

A.10.3.1 Binding Routines

```
rpc_binding_from_string_binding
rpc_binding_reset
rpc_binding_to_string_binding
```

A.10.3.2 Interface Routines

```
rpc_if_id_vector_free
```

A.10.3.3 Error Services

```
dce_error_inq_text
```

A.10.3.4 Endpoint Map Services

```
rpc_ep_resolve_binding
```

A.10.3.5 Local Management Services

```
rpc_mgmt_stats_vector_free
```

A.10.3.6 Local/Remote Management Services

```
rpc_mgmt_ep_elt_inq_begin
rpc_mgmt_ep_elt_inq_next
rpc_mgmt_ep_elt_inq_done
rpc_mgmt_ep_unregister
rpc_mgmt_inq_if_ids
rpc_mgmt_inq_is_server_listening
rpc_mgmt_inq_if_ids
rpc_mgmt_inq_server_princ_name
rpc_mgmt_inq_stats
rpc_mgmt_is_server_listening
rpc_mgmt_stats_vector_free
```

A.10.3.7 String Services

rpc_string_free

A.10.3.8 Name Service Routines

Manage Name Service Entries

rpc_ns_entry_expand_name
rpc_ns_binding_import_begin
rpc_ns_binding_import_done
rpc_ns_binding_import_next
rpc_ns_mgmt_binding_unexport
rpc_ns_mgmt_entry_create
rpc_ns_mgmt_entry_delete
rpc_ns_mgmt_entry_inq_if_ids

Manage Name Service Expirations

rpc_ns_mgmt_handle_set_exp_age
rpc_ns_mgmt_inq_exp_age
rpc_ns_mgmt_set_exp_age

Manage Name Service Groups

rpc_ns_group_delete
rpc_ns_group_mbr_add
rpc_ns_group_mbr_inq_begin
rpc_ns_group_mbr_inq_done
rpc_ns_group_mbr_inq_next
rpc_ns_group_mbr_remove

Manage Name Service Profiles

rpc_ns_profile_delete
rpc_profile_elt_add
rpc_profile_elt_inq_begin
rpc_profile_elt_inq_done
rpc_profile_elt_inq_next
rpc_profile_elt_remove

A.10.3.9 UUID Services

uuid_compare
uuid_create
uuid_create_nil
uuid_equal
uuid_from_string
uuid_hash
uuid_is_nil
uuid_to_string

Appendix B

NCS 1.5.1 Client and Server Programs

This appendix illustrates the NCS 1.5.1 versions of the client and server programs that have been converted to DCE RPC. For the DCE RPC versions, see Chapters 4 and 5.

B.1 NCS 1.5.1 binopfw Program

The **binopfw** program is a simple distributed application. In this program, the client specifies a server host on the command line, and the server listens on a port that is dynamically allocated by the RPC runtime. The server registers with the Local Location Broker on its host so that the LLB can forward calls to the server port.

B.1.1 Client Code

Figure B–1 shows the client module, **client.c**.

```
#include <stdio.h>
#include "nbase.h"
#include "binopfw.h"
#include "socket.h"
#include <ppfm.h>

#define CALLS_PER_PASS 100

globalref uuid_$t uuid_$nil;
extern long time();
extern char *error_text();

main(argc, argv)
int argc;
char *argv[];
{
        handle_t h;
        status_$t st;
        socket_$addr_t loc;
        unsigned long llen;
        socket_$string_t name;
        unsigned long namelen = sizeof(name);
        unsigned long port;
        ndr_$long_int i, n;
        int k, passes;
        long start_time, stop_time;

        if (argc != 3) {
        fprintf(stderr, "usage: client hostname passes\n");
        exit(1);
        }

        passes = atoi(argv[2]);

        pfm_$init((long) pfm_$init_signal_handlers);

        socket_$from_name((long)socket_$unspec, (ndr_$char *)argv[1],
            (long)strlen(argv[1]), (long)socket_$unspec_port, &loc,
            &llen, &st);
```

Figure B–1. NCS 1.5.1 Version of binopfw/client.c

```
    if (st.all != status_$ok){
        fprintf(stderr, "Can't convert name to sockaddr - %s\n",
                    error_text(st));
    exit(1);
    }

    h = rpc_$bind(&uuid_$nil, &loc, llen, &st);
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't bind - %s\n", error_text(st));
        exit(1);
    }

    rpc_$inq_binding(h, &loc, &llen, &st);
    if (st.all != status_$ok) {
       fprintf(stderr,"Can't inq binding - %s\n",error_text(st));
    exit(1);
    }
    socket_$to_name(&loc, llen, name, &namelen, &port, &st);
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't convert sockaddr to name - %s\n",
                    error_text(st));
        exit(1);
    }
    name[namelen] = 0;
    printf("Bound to port %ld at host %s\n", port, name);

    for (k = 1; k <= passes; k++) {
        start_time = time(NULL);
        for (i = 1; i <= CALLS_PER_PASS; i++) {
                binopfw$add(h, i, i, &n);
                if (n != i+i)
                      printf("Two times %ld is NOT %ld\n", i, n);
        }

        stop_time = time(NULL);
        printf("pass %3d; real/call: %2ld ms\n",
                k, ((stop_time - start_time) * 1000)/CALLS_PER_PASS);
    }
}
```

Figure B–1. NCS 1.5.1 Version of binopfw/client.c (Continued)

B.1.2 NCS 1.5.1 util.c

Figure B–2 shows the **util.c** module.

```
#include "binopfw.h"
char *error_text(st)
status_$t st;

{
    static char buff[200];
    extern char *error_$c)text();

    return(error_$c_text(st, buff, sizeof buff));
}
```

Figure B–2. NCS 1.5.1 Version of binopfw/util.c

B.1.3 NCS 1.5.1 server.c

Figure B–3 shows the **server.c** module.

```
#include <stdio.h>
#include "nbase.h"
#include "binopfw.h"
#include "lb.h"
#include "socket.h"
#include <ppfm.h>

globalref uuid_$t uuid_$nil;
globalref binopfw_v1$epv_t binopfw_v1$manager_epv;
extern char *error_text();

main(argc, argv)
int argc;
char *argv[];
{
        status_$t st;
        socket_$addr_t loc;
        unsigned long llen;
        unsigned long family;
        boolean validfamily;
        socket_$string_t name;
        unsigned long namelen = sizeof(name);
        unsigned long port;
        lb_$entry_t lb_entry;
        pfm_$cleanup_rec crec;

        if (argc != 2) {
           fprintf(stderr, "usage: server family\n");
           exit(1);
        }

        pfm_$init((long)pfm_$init_signal_handlers);

        family = socket_$family_from_name((ndr_$char *)argv[1],
                        (long)strlen(argv[1]), &st);
        if (st.all != status_$ok) {
           fprintf(stderr, "Can't get family from name - %s\n",
                 error_text(st));
        exit(1);
        }
        validfamily = socket_$valid_family(family, &st);
        if (st.all != status_$ok) {
           fprintf(stderr, "Can't check family - %s\n", error_text(st));
           exit(1);
```

Figure B–3. NCS 1.5.1 Version of binopfw/server.c

```
}   if (!validfamily) {
    printf("Family %s is not valid\n", argv[1]);
    exit(1);
}

rpc_$use_family(family, &loc, &llen, &st);
if (st.all != status_$ok) {
    fprintf(stderr, "Can't use family - %s\n", error_text(st));
exit(1);
}

rpc_$register_mgr(&uuid_$nil, &binopfw_v1$if_spec,
                  binopfw_v1$server_epv,
                  (rpc_$mgr_epv_t)&binopfw_v1$manager_epv,&st);
if (st.all != 0) {
    printf("Can't register manager - %s\n", error_text(st));
    exit(1);
}

lb_$register(&uuid_$nil, &uuid_$nil, &binopfw_v1$if_spec.id,
                  (long)lb_$server_flag_local,
                  (ndr_$char *)"binopfw example",
                  &loc, llen, &lb_entry, &st);
if (st.all != 0) {
    printf("Can't register - %s\n", error_text(st));
    exit(1);
}
socket_$to_name(&loc, llen, name, &namelen, &port, &st);
if (st.all != status_$ok) {
    fprintf(stderr, "Can't convert sockaddr to name - %s\n",
          error_text(st));
    exit(1);
}

name[namelen] = 0;

printf("Registered: name='%s', port=%ld\n", name, port);

st = pfm_$cleanup(&crec);
if (st.all != pfm_$cleanup_set) {
    status_$t stat;
    fprintf(stderr,"Server received signal- %s\n",error_text(st));
    lb_$unregister(&lb_entry, &stat);
    rpc_$unregister(&binopfw_v1$if_spec, &stat);
    pfm_$signal(st);
    }
```

Figure B–3. NCS 1.5.1 Version of binopfw/server.c (Continued)

```
        rpc_$listen((long) 5, &st);
}       rpc_$use_family(family, &loc, &llen, &st);
        if (st.all != status_$ok) {
           fprintf(stderr, "Can't use family - %s\n", error_text(st));
        exit(1);
        }

        rpc_$register_mgr(&uuid_$nil, &binopfw_v1$if_spec,
                          binopfw_v1$server_epv,
                          (rpc_$mgr_epv_t)&binopfw_v1$manager_epv,&st);
        if (st.all != 0) {
           printf("Can't register manager - %s\n", error_text(st));
           exit(1);
        }

        lb_$register(&uuid_$nil, &uuid_$nil, &binopfw_v1$if_spec.id,
                          (long)lb_$server_flag_local,
                          (ndr_$char *)"binopfw example",
                          &loc, llen, &lb_entry, &st);
        if (st.all != 0) {
           printf("Can't register - %s\n", error_text(st));
           exit(1);
        }
        socket_$to_name(&loc, llen, name, &namelen, &port, &st);
        if (st.all != status_$ok) {
           fprintf(stderr, "Can't convert sockaddr to name - %s\n",
                error_text(st));
           exit(1);
        }

        name[namelen] = 0;

        printf("Registered: name='%s', port=%ld\n", name, port);

        st = pfm_$cleanup(&crec);
        if (st.all != pfm_$cleanup_set) {
           status_$t stat;
           fprintf(stderr,"Server received signal- %s\n",error_text(st));
           lb_$unregister(&lb_entry, &stat);
           rpc_$unregister(&binopfw_v1$if_spec, &stat);
           pfm_$signal(st);
           }

        rpc_$listen((long) 5, &st);
}
```

Figure B–3. NCS 1.5.1 Version of binopfw/server.c (Continued)

B.1.4 NCS 1.5.1 manager.c

Figure B–4 shows the **manager.c** module.

```
#include "binopfw.h"

globaldef binopfw_v1$epv_t
    binopfw_v1$manager_epv = {binopfw$add};

void binopfw$add(h, a, b, c)
handle_t h;
ndr_$long_int a, b, *c;
{
    *c = a + b;
}
```

Figure B–4. NCS 1.5.1 Version of binopfw/manager.c

B.2 NCS 1.5.1 stacks Program

The **stacks** program demonstrates how you can use NCS to implement an interface for several types of objects. A separate manager implements each combination of interface and type. The server registers its objects and their types with the RPC runtime and the Location Broker; it registers its managers with the RPC runtime.

In the **stacks** program, a server manages two types of stacks, one based on lists and one based on arrays.

B.2.1 The stacks Interface Definition

Figure B–5 shows **stacks.idl**, the NIDL definition for the **stacks** interface. Different object types require different implementations of operations, but not different signatures.

```
%c
[uuid(4438675bf000.0d.00.00.fe.da.00.00.00), version(1)]
interface stacks
{
[idempotent]
   void stacks$init(handle_t [in] h);

/* stack functions return non-zero on error, zero otherwise */

   int stacks$push(handle_t [in] h, int [in] value);

   int stacks$pop(handle_t [in] h, int [out] *value);
}
```

Figure B–5. The NCS 1.5.1 stacks.idl Interface Definition

B.2.2 The stacksdf.h Header File

The **stacksdf.h** header file, shown in Figure B–6, defines symbolic constants to represent UUIDs for the two stack objects and their types. The replacement texts for these constants are C representations of UUIDs, which were generated by invoking **uuid_gen** with the **–C** option.

```
/* the two stack objects and their types */

/* the array-based object */
#define ASTACK   {0x44349d2c, 0x2000, 0x0000, 0x0d, \
                      {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

#define ASTACKT  {0x44349e25, 0x0000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

/* the list-based object */
#define LSTACK   {0x44349e48, 0x2000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}

#define LSTACKT  {0x44349eed, 0x6000, 0x0000, 0x0d, \
                     {0x00, 0x00, 0xfe, 0xda, 0x00, 0x00, 0x00}}
```

Figure B–6. The NCS 1.5.1 stacksdf.h Header File

B.2.3 The stacks Client Module

The **stacks** client module, shown in Figure B–7, works the same way as its DCE RPC counterpart shown in Chapter 6. It lets the user access two types of stacks within one session. When the client program calls **stacks$push** or **stacks$pop**, the object UUID in the handle determines the stack to be accessed.

```
#include <stdio.h>
#include "nbase.h"
#include "stacks.h"
#include "stackdf.h"
#include "lb.h"
#include "socket.h"
#include <ppfm.h>
#include "uuid.h"

#define CALLS_PER_PASS     100
#define MAXENTRIES         5    /* how many LB entries we can handle */

extern long time();
extern char *error_text();

main()
{
      handle_t handle[2];
      status_$t st;
      lb_$entry_t entries[MAXENTRIES];
      lb_$lookup_handle_t ehandle = lb_$default_lookup_handle;
      unsigned long nresults;
      socket_$addr_t loc;
      unsigned long llen;
      socket_$string_t name;
      unsigned long namelen = sizeof(name);
      unsigned long port;
      static uuid_$t types[2] = {ASTACKT, LSTACKT};
      int s, t, k, found_if;
      ndr_$long_int val;
      char command[100], which[100], value[100];

      pfm_$init((long) pfm_$init_signal_handlers);
```

Figure B–7. The NCS 1.5.1 stacks/client.c Module

```
/* bind handles for each object type */
    for (t = 0; t < 2; t++) {
    /* find lb entries for the type */
    lb_$lookup_type(&types[t], &ehandle, MAXENTRIES,
                                &nresults, entries, &st);
    if (st.all != status_$ok) {
       fprintf(stderr, "Can't lookup type[%d] - %s\n", t,
                        error_text(st));
       exit(1);
    }
    if (nresults < 1) {
       fprintf(stderr, "Couldn't find interfaces for type[%d]\n", t);
       exit(1);
    }

    /* check for appropriate interface for the type */
    for (k = 0, found_if = 0; k < nresults; k++)
        if (uuid_$equal(&entries[k].obj_interface,
                        &stacks_v1$if_spec.id)
            && socket_$valid_family(entries[k].saddr.family, &st)) {
            found_if = 1; /* found appropriate interface */
            break;
    }

    if (!found_if) {
       fprintf(stderr, "Couldn't find appropriate interface\n");
       exit(1);
    }

    /* bind handle */
    handle[t] = rpc_$bind(&entries[k].object, &entries[k].saddr,
                          entries[k].saddr_len, &st);
    if (st.all != status_$ok) {
       fprintf(stderr, "Can't bind handle - %s\n",
                 error_text(st));
       exit(1);
    }

    rpc_$inq_binding(handle[t], &loc, &llen, &st);
    if (st.all != status_$ok) {
       fprintf(stderr, "Can't inq binding - %s\n", error_text(st));
       exit(1);
    }
```

Figure B–7. The NCS 1.5.1 stacks/client.c Module (Continued)

```
        socket_$to_name(&loc, llen, name, &namelen, &port, &st);
        if (st.all != status_$ok) {
            fprintf(stderr, "Can't convert sockaddr to name - %s\n",
                    error_text(st));
            exit(1);
        }
        name[namelen] = 0;
        printf("%s handle bound to port %ld at host %s\n",
                t?"lstack":"astack", port, name);
}

printf("Initialize stack objects (y/n)? ");
gets(command);
if (*command != 'n' && *command != 'N') {
        stacks$init(handle[0]);
        stacks$init(handle[1]);
}

do {
        printf("push, pop, or quit: ");
        gets(command);
        if (!strcmp(command, "quit")) break;
        printf("astack or lstack: ");
        gets(which);

        if (!strcmp(which, "astack")) s = 0;
        else s = 1;

        if (!strcmp(command, "push")) {
            printf("value: ");
            gets(value);
            val = (ndr_$long_int)atoi(value);
            printf("Pushing %d onto %s...", val, s?"lstack":"astack");
            if (stacks$push(handle[s], val)) printf("stack full!\n");
            else printf("successful\n");
        }
        else if (!strcmp(command, "pop")) {
            printf("Popping off of %s...", s?"lstack":"astack");
            if (stacks$pop(handle[s], &val))
               printf("nothing on stack!\n");
            else printf("value is %d\n", val);
        }

    } while (strcmp(command, "quit"));
}
```

Figure B–7. The NCS 1.5.1 stacks/client.c Module (Continued)

B.2.4 The stacks Server Module

The **server.c** module, shown in Figure B–8, is linked together with two manager modules to form the **stacks** server program. The server module declares two manager EPVs as external variables; each manager EPV is defined in its own manager module.

```
#include <stdio.h>
#include "nbase.h"
#include "stackdf.h"
#include "stacks.h"
#include "lb.h"
#include "socket.h"
#include <ppfm.h>

globalref stacks_v1$epv_t stacks_v1$amanager_epv;
globalref stacks_v1$epv_t stacks_v1$lmanager_epv;
extern char *error_text();

main(argc, argv)
int argc;
char *argv[];
{
       status_$t st;
       socket_$addr_t loc;
       unsigned long llen;
       unsigned long family;
       boolean validfamily;
       socket_$string_t name;
       unsigned long namelen = sizeof(name);
       unsigned long port;
       lb_$entry_t lb_entry[2];
       pfm_$cleanup_rec crec;
       static uuid_$t astack = ASTACK, astackt = ASTACKT;
       static uuid_$t lstack = LSTACK, lstackt = LSTACKT;

       if (argc != 2) {
          fprintf(stderr, "usage: server family\n");
          exit(1);
       }

       pfm_$init((long) pfm_$init_signal_handlers);
       family = socket_$family_from_name((ndr_$char *)argv[1],
                                         (long)strlen(argv[1]), &st);
```

Figure B–8. The NCS 1.5.1 stacks/server.c Module

```
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't get family from name - %s\n",
                                error_text(st));
        exit(1);
    }

validfamily = socket_$valid_family(family, &st);
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't check family - %s\n",
                                error_text(st));
        exit(1);
    }
    if (!validfamily) {
        printf("Family %s is not valid\n", argv[1]);
        exit(1);
    }

    rpc_$use_family(family, &loc, &llen, &st);
    if (st.all != status_$ok) {
        fprintf(stderr, "Can't use family - %s\n", error_text(st));
        exit(1);
    }

/* register manager and object for array-based stack object...*/

    rpc_$register_mgr(&astackt, &stacks_v1$if_spec,
                      stacks_v1$server_epv,
                      (rpc_$mgr_epv_t)&stacks_v1$amanager_epv, &st);
    if (st.all != 0) {
        printf("Can't register astack manager - %s\n",
                      error_text(st));
        exit(1);
    }

    rpc_$register_object(&astack, &astackt, &st);
    if (st.all != 0) {
        printf("Can't register astack object - %s\n",
                       error_text(st));
        exit(1);
    }
    rpc_$register_object(&lstack, &lstackt, &st);
    if (st.all != 0) {
        printf("Can't register lstack object - %s\n", error_text(st));
    exit(1);
    }
```

Figure B–8. The NCS 1.5.1 stacks/server.c Module (Continued)

```
/* ...and list-based stack object *
/
       rpc_$register_mgr(&lstackt, &stacks_v1$if_spec,
                    stacks_v1$server_epv,
                    (rpc_$mgr_epv_t)&stacks_v1$lmanager_epv, &st);
       if (st.all != 0) {
          printf("Can't register lstack manager - %s\n",
                  error_text(st));
          exit(1);
       }

       rpc_$register_object(&lstack, &lstackt, &st);
       if (st.all != 0) {
          printf("Can't register lstack object - %s\n",
                           error_text(st));
          exit(1);
       }
  /* register array-based stack object/interface with the lb... */

       lb_$register(&astack, &astackt, &stacks_v1$if_spec.id, 0L,
       (ndr_$char *)"astack example", &loc, llen, &lb_entry[0], &st);
       if (st.all != 0){
          printf("Can't register astack - %s\n", error_text(st));
          exit(1);
       }

       socket_$to_name(&loc, llen, name, &namelen, &port, &st);
       if (st.all != status_$ok) {
          fprintf(stderr, "Can't convert sockaddr to name - %s\n",
                error_text(st));
          exit(1);
       }
       name[namelen] = 0;
       printf("astack registered: name='%s', port=%ld\n", name, port);

       /* and list-based stack object/interface */

       lb_$register(&lstack, &lstackt, &stacks_v1$if_spec.id, 0L,
       (ndr_$char *)"lstack example", &loc, llen, &lb_entry[1], &st);
       if (st.all != 0)
 {
          printf("Can't register lstack - %s\n", error_text(st));
          exit(1);
}
```

Figure B–8. The NCS 1.5.1 stacks/server.c Module (Continued)

```
        socket_$to_name(&loc, llen, name, &namelen, &port, &st);
        if (st.all != status_$ok) {
            fprintf(stderr, "Can't convert sockaddr to name - %s\n",
                  error_text(st));
            exit(1);
        }

        name[namelen] = 0;
        printf("lstack registered: name='%s', port=%ld\n", name, port);
        st = pfm_$cleanup(&crec);
        if (st.all != pfm_$cleanup_set) {
            status_$t stat;
            fprintf(stderr, "Server received signal - %s\n",
                  error_text(st));
            lb_$unregister(&lb_entry[0], &stat);
            lb_$unregister(&lb_entry[1], &stat);
            rpc_$unregister(&stacks_v1$if_spec, &stat); /*once for each*/
            rpc_$unregister(&stacks_v1$if_spec, &stat);  /* manager */
            pfm_$signal(st);
        }

        rpc_$listen((long)5, &st);
}
```

Figure B–8. The NCS 1.5.1 stacks/server.c Module (Continued)

The **lmanager.c** module, shown in Figure B–9, illustrates the stack manager for the list-based stack.

```
/* the list-based stack manager module */
#include "stacks.h"

void stacks$lstack_init();
ndr_$long_int stacks$lstack_push(), stacks$lstack_pop();

globaldef stacks_v1$epv_t stacks_v1$lmanager_epv =
    {stacks$lstack_init, stacks$lstack_push, stacks$lstack_pop};

#define NULL (struct node *)0
extern struct node *malloc();

static struct node {
       ndr_$long_int value;
       struct node *next;
} the_stack;

void stacks$lstack_init(h)
handle_t h;
{
       the_stack.next = NULL;
}

ndr_$long_int stacks$lstack_push(h, value)
handle_t h;
ndr_$long_int value;
{
       struct node *head = malloc(sizeof(struct node));

       if (head == NULL) return -1;              /* stack is full */
       head->value = value;
       head->next = the_stack.next;
       the_stack.next = head;
       return 0;
}
ndr_$long_int stacks$lstack_pop(h, value)
handle_t h;
ndr_$long_int *value;
{
       struct node *head = the_stack.next;

       if (head == NULL) return -1;             /* stack is empty */
       *value = head->value;
       the_stack.next = head->next;
       free(head);

       return 0;
}
```

Figure B–9. The NCS 1.5.1 stacks/lmanager.c Module

The **amanager.c** module, shown in Figure B–10, illustrates the stack manager for the array-based stack.

```
/* the array-based stack manager module */

#include "stacks.h"

void stacks$astack_init();
ndr_$long_int stacks$astack_push(), stacks$astack_pop();

globaldef stacks_v1$epv_t stacks_v1$amanager_epv =
    {stacks$astack_init, stacks$astack_push, stacks$astack_pop};

#define STACKSIZE    1000

static struct {
       int head;
       ndr_$long_int values[STACKSIZE];
} the_stack;

void stacks$astack_init(h)
handle_t h;
{
       the_stack.head = STACKSIZE;
}

ndr_$long_int stacks$astack_push(h, value)
handle_t h;
ndr_$long_int value;
{
       if (the_stack.head == 0) return -1;        /* stack is full */
       the_stack.values[--the_stack.head] = value;
       return 0;
}

ndr_$long_int stacks$astack_pop(h, value)
handle_t h;
ndr_$long_int *value;
{
       if (the_stack.head == STACKSIZE) return -1;/*stack is empty */
          *value = the_stack.values[the_stack.head++];
       return 0;
}
```

Figure B–10. The NCS 1.5.1 stacks/amanager.c Module

Appendix C

IDL and ACF Attribute Summary

This appendix summarizes IDL and ACF attributes.

C.1 DCE IDL Attribute Summary

Table C–1 contains a list of the DCE IDL attributes and what kinds of definitions they are used in. The rest of this section gives a brief description of the attributes.

Table C–1. DCE IDL Attributes

Attribute	Where Used
uuid **version** **endpoint** **pointer_default** **local**	Interface definition headers
broadcast **maybe** **idempotent**	Operations
in **out**	Parameters

Table C–1. DCE IDL Attributes (Cont.)

Attribute	Where Used
ignore	Structures
max_is **size_is** **first_is** **last_is** **length_is** **string**	Arrays
ptr **ref**	Pointers
handle	Customized handles
context_handle	Context handles
transmit_as	Type declarations
v1_array **v1_enum** **v1_string** **v1_struct**	Migration

C.1.1 IDL Attributes in Interface Definition Headers

uuid Specifies the UUID that is assigned to the interface

version Specifies a particular version of a remote interface

endpoint Specifies the well-known endpoint or endpoints on which servers that export the interface listen

pointer_default Specifies the default semantics for pointers that are declared in the interface definition; either **ref** or **ptr**

local Indicates that an interface definition does not declare any remote operations and that the IDL compiler should generate a header file but no stub files

C.1.2 IDL Attributes for Operations

A operation can also have the **ptr**, **string**, or **context_ handle** attribute.

idempotent	Specifies that the operation is idempotent
broadcast	Specifies that the operation is always to be broadcast
maybe	Specifies that the caller of the operation does not require and will not receive any response

C.1.3 IDL Attributes for Parameters

A parameter can also have any of the array attributes, any of the migration attributes, or the **ref**, **ptr**, **string**, or **context_ handle** attribute.

in	Specifies that the parameter is an input parameter
out	Specifies that the parameter is an output parameter

C.1.4 IDL Attributes for Structures

Structure members can also have any of the array attributes, any of the migration attributes, or the **ref**, **ptr**, or **string** attribute.

ignore	Specifies that the pointer member being declared is not to be transmitted in remote procedure calls

C.1.5 IDL Attributes for Unions

A union member can have the **ptr**, **string**, **v1_array**, or **v1_string** attribute.

C.1.6 IDL Attributes for Arrays

An array can have the **string** attribute.

string	Specifies that the array has the properties of a string

C.1.6.1 Conformant Array Attributes

A conformant array can also have the **string** attribute.

max_is	Specifies the name of a variable that contains the maximum possible upper bound for the major dimension of the array
size_is	Specifies the name of a variable that contains the maximum possible element count for the major dimension of the array

C.1.6.2 Varying Array Attributes

A varying array can also have the **string** attribute.

last_is	Specifies the name of a variable that contains the highest index value of the transmitted data
first_is	Specifies the name of a variable that contains the lowest index value of the transmitted data
length_is	Specifies the name of a variable that contains the actual number of elements in the transmitted data

C.1.7 IDL Attributes for Pointers

A pointer can also have the **string** attribute to indicate that it points to a string.

ptr	Specifies that the pointer is a full pointer
ref	Specifies that the pointer is a reference pointer

C.1.8 IDL Attributes for Customized Handles

handle	Specifies that the type being declared is a user-defined, non-primitive handle type, to be used in place of the predefined, primitive handle type **handle_t**

C.1.9 IDL Attributes for Context Handles

context_handle Identifies a parameter or type that functions as a context handle. The context handle provides a handle to state information that is maintained for a client by management code

C.1.10 IDL Attributes for Type Declarations

A type declaration may also specify any of the pointer type attributes or the **context_handle** or **handle** attribute.

transmit_as Specifies the "transmitted type" that the stub passes over the network as differentiated from the "presented type" that clients and servers manipulate

C.1.11 IDL Attributes for Compatibility with NCS 1.5.1

These attributes, which are intended primarily for use by the **nidl_to_idl** utility, enable DCE RPC applications to interoperate with NCS 1.5.1 programs. Each attribute tells the IDL compiler to generate code that marshalls and unmarshalls the NCS 1.5.1 network data representation for a particular data type.

v1_array Specifies an NCS 1.5.1 array; can be specified for a type definition or a parameter or field definition

v1_enum Specifies an enumeration compatible with the NCS 1.5.1 **long enum** data type

v1_string Specifies an NCS 1.5.1 null-terminated array of elements whose type resolves to **char**

v1_struct Specifies an alternate data alignment, compatible with NCS 1.5.1, for a structure

C.2 DCE ACF Attribute Summary

Table C–2 summarizes the attributes that can be specified in a DCE attribute configuration file.

Table C–2. DCE ACF Attributes

Attribute	Usage
auto_handle **explicit_handle** **implicit_handle**	Controls binding
comm_status **fault_status**	Specifies parameters to hold status conditions occurring in the call
code **nocode**	Controls which operations in the IDL file are compiled
in_line **out_of_line**	Controls the marshalling of data
represent_as	Controls conversion between local and network data types
enable_allocate	Forces the initialization of the stub memory management routines
heap	Specifies objects to be allocated from heap memory

Glossary

This glossary defines terms for the DCE Remote Procedure Call (RPC) Component and compares them to NCS 1.5.1 terms, noting differences where they exist. Unless otherwise noted, the term is the same for both NCS 1.5.1 and DCE RPC.

This glossary is restricted to terms associated with applications programming. For definitions of other DCE terms, see the glossary in *Introduction to OSF DCE*. For a complete NCS 1.5.1 glossary, see the *Networking Computing System Tutorial* and *Network Computing System Reference Manual*.

ACF

See **attribute configuration file.**

active context handle

In DCE RPC applications, a context handle that the remote procedure has set to a non-null value and passed back to the calling program; the calling program supplies the active context handle in any future calls to procedures that share the same client context.

address family

See **network address family.**

alias (of a pointer)

Two full pointers used in the same operation are aliases if they point to the same storage area. *See also* **full pointer** and **reference pointer.**

aliasing

In DCE RPC, aliasing occurs when two pointers of the same operation point at the same storage.

allocate (a handle)

In NCS 1.5.1, to create an RPC handle that identifies an object but not a location. Such a handle is said to be "allocated" or "unbound."

DCE RPC does *not* provide this functionality; you cannot create an unbound handle.

API

Application programming interface.

application thread

In DCE, a thread of execution created and managed by application code. *See also* **client application thread, local application thread, RPC thread, server application thread.**

array

See **conformant array, varying array.**

at-most-once semantics

A characteristic of a procedure that restricts it to executing once, partially, or not at all—never more than once. *See also* **idempotent semantics.**

attribute

(1) An IDL or ACF syntax element, occurring within [] (brackets) and conveying information about an interface, type, field, parameter, or operation.

(2) An attribute of an entry in a name service database that stores binding, group, object, or profile information for an RPC application and identifies the entry as an RPC server entry; an NSI attribute.

attribute configuration file

An **.acf** file; an optional companion file to a DCE interface definition (**.idl**) file that modifies how the DCE IDL compiler locally interprets the interface definition.

attribute configuration language

A high-level declarative language that provides syntax for DCE attribute configuration files. *See also* **attribute configuration file.**

automatic binding method

In NCS 1.5.1, this is a binding technique in which the client uses generic handles. Each time the client makes a remote procedure call, the client stub invokes an autobinding routine that converts the generic handle to an RPC handle. (In DCE RPC, this technique is called "binding with customized handles." *See also* **customized binding handle.**)

In DCE RPC, automatic binding is the simplest method for a client to manage bindings for its remote procedure calls. The automatic method completely hides binding management from client application code. If the client makes a series of remote procedure calls, the stub passes the same binding handle with each call. The binding management is all in the client stub.

binding

In NCS 1.5.1, the representation of a server location in a handle. To bind a handle or to set its binding is to establish this representation.

In DCE RPC, a relationship between a client and a server involved in a remote procedure call.

See also **binding state** and **handle.**

binding handle

A reference to binding information that defines one possible binding. DCE RPC distinguishes between **client binding information** and **server binding information.** (While the functionality of client and server binding handles exists in NCS 1.5.1, no specific terms are defined.)

binding handle vector

In DCE RPC, a data structure that contains an array of binding handles and the size of the array.

In NCS 1.5.1, application programmers manually create arrays of binding handles when necessary. There is no predefined type.

binding information

In DCE RPC, information about one or more possible bindings, including an RPC protocol sequence, a network address, an endpoint, at least one transfer syntax, and an RPC protocol version number.

binding state

In NCS 1.5.1, a state reflecting the degree to which an RPC handle represents a server location. The three possible binding states are unbound, bound-to-host, and fully bound.

In DCE RPC, a binding handle can be only partially bound or fully bound.

See also **binding, customized binding handle, partially bound binding handle, primitive binding handle.**

bound-to-host handle

In NCS 1.5.1, an RPC handle that identifies an object and a host but not a port.

In DCE RPC, this is known as a **partially bound binding handle.**

bound-to-server handle

In NCS 1.5.1, an RPC handle that identifies an object, a host, and a port.

In DCE RPC, this is known as a **fully bound handle.**

broadcast

To send a remote procedure call request to all hosts in a network simultaneously.

Also, in DCE threads, to wake all threads waiting on a condition variable. *See also* **signal.**

broker

In NCS 1.5.1, a server that provides information about resources. A location broker is a broker.

call thread

In DCE RPC, a thread created by a server's RPC runtime to execute remote procedures. When engaged by a remote procedure call, a call thread temporarily forms part of the RPC thread of the call. *See also* **application thread, RPC thread.**

cancel

In DCE, a mechanism by which a client thread notifies a server thread (the cancelled thread) to terminate as soon as possible.

In DCE Threads, a mechanism by which a thread informs either itself or another thread to terminate as soon as possible. If a cancel arrives during an important operation, the cancelled thread may continue until it can terminate in a controlled manner.

cleanup handler

In NCS 1.5.1, a piece of code that allows a program to terminate gracefully when it receives an error. The **pfm_$cleanup** call establishes a cleanup handler. *See also* **Process Fault Manager**.

DCE replaces cleanup handlers with the DCE exception-returning package. The package is sometimes referred to by the names of some of its macros, TRY/CATCH.

client

The party that initiates a remote procedure call. A given application can act as both a client and a server. *See also* **server**.

client application thread

In DCE, a thread executing client application code that makes one or more remote procedure calls. *See also* **application thread, local application thread, server application thread.**

client binding information

In DCE RPC, information about a calling client provided by the client runtime to the server runtime, including the address where the call originated, the RPC protocol used for the call, the requested object UUID, and any client authentication information. *See also* **server binding information.**

client stub

The surrogate code for an RPC interface that is linked with and called by the client application code. In addition to general operations such as marshalling data, a client stub calls the RPC runtime to perform remote procedure calls, and optionally, manage bindings. *See also* **server stub, stub.**

communications link

A network path between an RPC client and server that uses a valid combination of transport and network protocols that are available to both the client and server RPC runtimes. *See also* **binding handle.**

compatible server

A server that offers the requested RPC interface and RPC object and that is available over a valid combination of network and transport protocols that are supported by the client and server RPC runtimes.

Concurrent Programming Support (CPS)

In NCS 1.5.1, a set of calls that create and manage a multitasking environment within a single process. These calls are especially useful in servers that simultaneously manage multiple remote requests. CPS is implemented on only a subset of the systems for which NCS is available.

In DCE, multitasking support is provided by POSIX threads.

conformant array

In DCE RPC, an array whose size is determined at run time. In NCS 1.5.1, this is called an "open array." A structure containing a conformant array as a member is a **conformant structure.** *See also* **varying array.**

context handle

In DCE RPC, a reference to client context maintained across calls by a server on behalf of a client. *See also* **server context.**

customized binding handle

In DCE RPC, a handle of a user-defined type from which a primitive binding handle can be derived by user-defined routines in application code. A customized binding handle enables application programmers to manage additional binding information that cannot be handled by the primitive binding handle.

This is called a **generic handle** in NCS 1.5.1.

DCE exception-returning package

A package, for use with DCE-based applications, to map signals to exceptions and then catch them when performing cleanup operations.

The exception-returning package replaces the Process Fault Manager provided in NCS 1.5.1.

dynamic endpoint

In DCE RPC, an endpoint that is generated by the RPC runtime for an RPC server when the server registers its protocol sequences and that expires when the server stops running. This is similar functionality to the NCS 1.5.1 **opaque port**. *See also* **endpoint, well-known endpoint.**

endpoint

In DCE RPC, an address of a specific server instance on a host. This is called a **port** in NCS 1.5.1. *See also* **dynamic endpoint, well-known endpoint.**

entry point vector (EPV)

A list of addresses for the entry points of a set of remote procedures that implements the operations declared in an interface definition. The addresses are listed in the same order as the corresponding operation declarations.

EPV

See **entry point vector.**

exception

An object that describes an error condition. With the **DCE exception-returning package**, your program can perform operations on exceptions to report and handle errors.

exception handling

See **DCE exception-returning package.**

exception scope

A block where one or more exceptions may be caught; that is, functions that may raise an exception. An exception scope is delimited by the macros TRY and ENDTRY. Any exceptions raised within the block, or within any functions that are called by the block, pass through this scope.

explicit binding method

The explicit method of managing the binding for a remote procedure call in which a remote procedure call passes a **binding handle** as its first parameter. The binding handle must be initialized in the application code. *See also* **automatic binding method, binding handle, implicit binding method.**

explicit handle

A handle that is passed as an operation parameter, rather than represented as a global variable in the client process. *See also* **implicit handle.**

export (an interface)

To provide access to an RPC interface. A server exports an interface to a client. *See also* **import.**

forward

To dispatch a remote procedure call request to a server that exports the requested interface for the requested object. In NCS 1.5.1, the Local Location Broker forwards remote procedure calls that are sent to the LLB forwarding port on a server host. In DCE, the RPC daemon (**rpcd**) forwards remote procedure calls.

full pointer

In DCE RPC, a pointer without the restrictions of a reference pointer. A full pointer value can change during a call, its value can be NULL, and it can be an alias. *See also* **alias** and **reference pointer.**

fully bound binding handle

In DCE RPC, a server binding handle that contains a complete server address including an endpoint. *See also* **partially bound binding handle.**

generic handle

In NCS 1.5.1, a handle of a data type other than **handle_t**. Generic handles appear in applications that use NCS 1.5.1 automatic binding (which differs from automatic binding in DCE RPC).

In DCE RPC, a generic handle is called a **customized handle.**

Global Location Broker (GLB)

In NCS 1.5.1, a server that maintains a database of location information about NCS servers in a network. The **glbd** maintains this database (also called the GLB database) as a replicated database, with different **glbd** daemons running on different machines throughout an internet. The **glbd** daemons collaborate to manage multiple copies (called replicas) of the GLB database.

handle

An opaque reference to information. The possessor of the handle is not able to manipulate the data inside it and is unaware of its internal structure; however, the handle is passed to other routines that make use of the references it contains in order to perform tasks requested by the possessor of the handle. *See also* **binding, handle, context handle.**

host

A computer that is attached to a network.

host ID

See also **network address.**

idempotent semantics

A characteristic of an RPC procedure in which executing it more than once with identical input produces no undesirable side effects. *See also* **at-most-once semantics.**

IDL

See **interface definition language.**

IDL compiler, DCE

A DCE RPC compiler that processes an interface definition language (**.idl**) file and an optional attribute configuration (**.acf**) file to generate client and server stubs, header files, and auxiliary files. *See also* **attribute configuration file, interface definition language, NIDL compiler.**

implement (an interface)

To provide the routines that execute the operations in an interface. A manager implements one interface for one type.

implicit binding method

The implicit method of managing the binding for a remote procedure call in which a global variable in the client application holds a server binding handle, which the stub passes to the RPC runtime. *See also* **automatic binding method, binding handle, explicit binding method.**

implicit handle

A handle that is represented as a global variable in the client process, rather than passed as an operation parameter.

import (an interface)

(1) To incorporate constant, type, and import declarations from one RPC interface definition into another RPC interface definition by means of the IDL or NIDL import statement.

(2) To request the operations defined by an interface. A client imports an interface from a server. *See also* **export.**

interface

A set of operations. The DCE Network Computing Architecture (NCA) specifies the Interface Definition Language (IDL) for defining interfaces.The NCS 1.5.1 Network Computing Architecture (NCA) specifies the Network Interface Definition Language (NIDL) for defining interfaces.

interface definition

A description of an interface written in the DCE Interface Definition Language (IDL) or in the Network Interface Definition Language (NIDL).

Interface Definition Language

In DCE, a high-level declarative language that provides syntax for interface definition files. NCS 1.5.1 provides a similar language, the Network Interface Definition Language (NIDL).

interface UUID

The universal unique identifier (UUID) that identifies a particular interface. Application programmers generate UUIDs by using the UUID generator (**uuidgen** in DCE, **uuid_gen** in NCS 1.5.1).

local application thread

In DCE RPC, an application thread that executes within the confines of one address space on a local system and passes control exclusively among local code segments. *See also* **application thread, client application thread, server application thread.**

local endpoint map

In DCE, a database that is located on the local system and maintained by the RPC daemon (**rpcd**). It associates binding information of local servers with their **dynamic endpoints.** (In NCS 1.5.1, similar information is called the LLB database and maintained by the **LLB.**)

local interface

An interface whose operations cannot be called remotely. A local interface is assigned the **local** interface attribute in its interface definition. When compiling this **.idl** file, no stubs are generated.

Local Location Broker (LLB)

In NCS 1.5.1, the server that maintains information about objects on the local host. The LLB also provided the Location Broker forwarding facility. Any host that runs NCS 1.5.1 servers should run the LLB daemon, **llbd.**

In DCE, the same functionality is provided by the RPC daemon, **rpcd.** It too, must run on every NCS server host.

Local Location Broker daemon (llbd)

In NCS 1.5.1, a daemon that implements the **Local Location Broker.** The **llbd** also provides a forwarding facility. All hosts that run NCS 1.5.1 servers must run **llbd.**

In DCE, the same functionality is provided by the RPC daemon, **rpcd.** It too must run on all DCE server hosts.

Location Broker

In NCS 1.5.1, a set of software including the Local Location Broker, the Global Location Broker, and the Location Broker Client Agent. The Location Broker maintains information about the locations of objects and interfaces.

In DCE, the RPC daemon, **rpcd**, maintains binding information for both DCE RPC–based and NCS 1.5.1-based servers on the local host. For DCE-based applications, global naming service is provided by the DCE Directory Service component.

Location Broker Client Agent

In NCS 1.5.1, the part of the RPC runtime library through which programs communicate with Global Location Brokers and Local Location Brokers.

manager

A set of remote procedures that implements the operations of an RPC interface for a given object type. *See also* **interface, object type.**

manager entry point vector (EPV)

The run-time code on the server side uses this entry point vector to dispatch incoming remote procedure calls.

manual binding

In NCS 1.5.1, a binding technique in which the client uses RPC handles (**handle_t**) to make RPC run-time calls to create and bind the handle.

In DCE RPC, this technique is referred to as binding with a **primitive handle.**

marshalling

The process by which an RPC stub converts local arguments into network data and packages the network data for transmission. *See also* **unmarshalling.**

multi-threaded server

A server implemented on a multi-threading system to respond to multiple RPC requests concurrently.

In NCS 1.5.1, on systems that support multi-threading, the RPC runtime starts a new thread of execution for each incoming RPC request.

In DCE RPC, the RPC runtime requires that the operating system support multi-threading and every server is assumed to be multi-threaded.

multi-threading

Supporting multiple threads of execution sharing an address space within a single process. NCS 1.5.1 supports, but does not require, a multi-threading system.

DCE RPC requires threads. Multi-threading raises issues of re-entrancy and synchronization that are beyond the scope of this book. For more information, see the documentation for the threads software the DCE uses on your operating system.

name service interface (NSI)

A part of the application programming interface of the DCE RPC runtime. NSI routines access a name service, such as CDS, for RPC applications.

NCA

See **Network Computing Architecture.**

NDR

See **Network Data Representation.**

network address

An address that identifies a specific RPC host on a network.

In NCS 1.5.1, a network address is represented in a **socket address**, in binary format, or as a character string. In DCE RPC, it is represented in a binding handle or a string binding.

network address family

A set of identifiers and address formats that correspond to the communications protocols of a particular network architecture; for example, the Internet network address family corresponds to the TCP/IP and UDP/IP protocols. The network address families supported by DCE or NCS depend on the platform. *See also* **protocol family.**

Network Computing Architecture (NCA)

An architecture for distributing software applications across heterogeneous collections of networks, computers, and programming environments. NCA specifies the DCE remote procedure call architecture. Early versions of NCA specified the NCS remote procedure call architecture.

Network Computing Kernel (NCK)

The runtime components of the Network Computing System. These components include the Remote Procedure Call runtime library, the RPC daemon, and the Location Broker (for programs using NCS 1.5.1). NCK contains all the software needed to support a distributed application.

Network Computing System (NCS)

A set of software components, developed by Hewlett-Packard, that conforms to the Network Computing Architecture. These components include the RPC runtime library, the IDL compiler, the Local and Global Location Brokers and the NIDL compiler.

Network Data Representation (NDR)

A transfer syntax defined by the Network Computing Architecture.

Network Interface Definition Language (NIDL)

In NCS 1.5.1, a declarative language for defining interfaces. For DCE RPC, a similar **Interface Definition Language** replaces NIDL.

NIDL

See **Network Interface Definition Language.**

NIDL compiler

In NCS 1.5.1, a compiler that takes an interface definition written in NIDL as input and generates as output C source code modules, including client and server stubs. DCE provides the IDL compiler for similar functionality.

NSI

See **name service interface.**

object

For RPC applications, an object can be anything that an RPC server defines and identifies to its clients (using an object UUID). Often, an RPC object is a physical computing resource such as a database, directory, device, or processor. Alternatively, an RPC object can be an abstraction that is meaningful to an application.

object type

A class of objects that is accessible through one or more interfaces.

object type UUID

A universal unique identifier (UUID) that identifies a particular type of object. Every object has a type.

object UUID

A universal unique identifier (UUID) that identifies a particular RPC object. A server specifies a distinct object UUID for each of its RPC objects; to access a particular RPC object, a client uses the object UUID to find the server that offers the object. *See also* **object, Universal Unique Identifier.**

opaque port

In NCS 1.5.1, a port that is dynamically assigned to a server by the RPC runtime library. The port number is said to be opaque because there is no need for either clients or servers to know the number. In DCE RPC, this is called a **dynamic endpoint.**

open array

In NCS 1.5.1, an array whose declaration does not specify an explicit fixed length. The length of an open array is not determined until an operation that uses it is called.

In DCE RPC, this is called a **conformant array.**

operation

The task performed by a given RPC function or procedure.

partially bound binding handle

In DCE RPC, a server binding handle that contains an incomplete server address lacking an endpoint. In NCS 1.5.1, this is called a **bound-to-host binding handle.**

pointer

See **reference pointer** and **full pointer**.

port

In NCS 1.5.1, a specific communications endpoint within a given host. A port is identified by a port number. In DCE RPC, this is called an **endpoint**.

port number

In NCS 1.5.1, the part of a socket address that identifies a port within a host. In DCE RPC, this is called an **endpoint**.

presented type

For RPC data types with the NIDL or IDL **transmit_as** attribute, the data type that clients and servers manipulate. Stubs invoke conversion routines to convert the presented type to a transmitted type, which is passed over the network.

primitive binding handle

The DCE uses this term to refer to a binding handle whose data type in IDL is **handle_t** and in application code is **rpc_binding_handle_t**. The NCS 1.5.1 analog to this is the **RPC handle**. *See also* **customized binding handle**.

Process Fault Manager (PFM)

In NCS 1.5.1, a set of calls that allows programs to manage signals, faults, and exceptions by establishing cleanup handlers. The **pfm_$** calls provided with NCS 1.5.1 are a portable subset of the Apollo Domain/OS **pfm_$** calls.

In DCE RPC, the PFM calls are replaced by the **DCE exception-returning package**.

protocol family

A set of communications protocols such as the Internet Protocol. All members of a protocol family use a common addressing mechanism to identify endpoints. We use the terms "protocol family" and "address family" synonymously.

protocol sequence

See **RPC protocol sequence**.

protocol sequence vector

A data structure that contains an array-size count and an array of pointers to RPC protocol-sequence strings. *See also* **RPC protocol sequence.**

reference pointer

A non-null pointer whose value is invariant during a remote procedure call and cannot be an alias. *See also* **alias** and **full pointer.**

register (an interface with the RPC runtime library)

To list an RPC interface with the RPC runtime. In the DCE RPC API, servers use the **rpc_server_register_if** routine to register interfaces, which specifies the manager that implements a particular interface for a particular type.

register (an object and an interface with the local endpoint map)

To place server-addressing information into the endpoint map. In the DCE RPC API, servers use the **rpc_ep_register** routine to register objects with the local endpoint map.

register (an object with the RPC runtime library)

In the DCE RPC API, servers use the **rpc_object_set_type** routine to register objects; this call specifies an object and its type.

remote procedure call (RPC)

A procedure call executed by a procedure located in a separate address space from the calling code.

RPC Communication Services

A service of the RPC runtime. It provides a set of routines that control each type of network operation, regardless of what RPC protocol handles a remote procedure call. *See also* **RPC protocol.**

RPC control program (rpccp)

An interactive management facility for managing name service entries and endpoint maps for RPC applications.

RPC daemon (rpcd)

The process that provides the endpoint map service for a system. The **rpcd** daemon provides the same functionality as the **Local Location Broker daemon** in NCS 1.5.1.

RPC handle

In NCS 1.5.1, a handle of the data type **handle_t**. RPC handles appear in applications that use manual binding.

DCE RPC calls this a **primitive binding handle**.

RPC interface

A logical grouping of operation, data type, and constant declarations that serves as a network contract for calling a set of remote procedures. *See also* **interface definition**.

RPC Management Services

A set of services of the RPC runtime. The Management Service provides a set of routines for basic management operations such as monitoring and stopping servers.

RPC Naming Services

A set of services of the RPC runtime. The Naming Service provides a set of routines that use a name service to perform operations on RPC entries in a namespace. Naming Service operations export information about RPC servers, interfaces, and objects; import information about servers that offer a specific interface (and, optionally, object); and manage RPC entries.

RPC protocol

An RPC-specific communications protocol that supports the semantics of DCE RPC and runs over either connectionless (datagram) or connection-oriented communications protocols. The RPC protocol is specified in the first part of the **RPC protocol sequence**, **ncadg** for datagram service, and **ncacn** for connection-oriented service.

RPC protocol sequence

A valid combination of communications protocols represented by a character string. Each protocol sequence typically includes three protocols: a network protocol, a transport protocol, and an RPC protocol that works with those network and transport protocols.

RPC runtime

A set of operations that manages communications, provides access to the name service database, and performs other tasks, such as managing servers and accessing security information, for RPC applications.

RPC runtime library

Routines of the RPC runtime that support the RPC application on a system. The runtime library provides a public interface to application programmers, the application programming interface (API), and a private interface to stubs the Stub Programming Interface (SPI).

RPC thread

In DCE RPC, a logical thread within which a remote procedure call executes. *See also* **call thread, thread.**

rpccp

See **RPC control program.**

rpcd

See **RPC daemon.**

server

The party that receives remote procedure calls. A given application can act as both an RPC server and an RPC client. *See also* **client.**

server address

Information that indicates one way to access an RPC server over the network. The server address includes an RPC protocol sequence, network address, and endpoint. A server can have several server addresses and can use a separate binding handle to refer to each address. *See also* **binding handle, endpoint, network address, RPC protocol sequence.**

server application thread

In the DCE RPC API, a thread executing the server application code that initializes the server and listens for incoming calls. *See also* **application thread, client application thread, local application thread.**

server binding information

Binding information for a particular DCE RPC server. *See also* **client binding information.**

server context

State in a server's address space generated by a set of remote procedures (manager) and maintained across a set of calls on behalf of a particular client.

server stub

The surrogate calling code for an RPC interface that is linked with server application code containing one or more sets of remote procedures (managers) that implement the interface. *See also* **client stub, manager, stub.**

service

An integral set of RPC interfaces offered together by a server to meet a specific goal.

set (a binding)

To set the representation of a server location in an RPC handle.

signal

In DCE Threads, to wake only one thread waiting on a condition variable. *See also* **broadcast.**

signature

The syntax of an operation, that is, its name, the data type it returns, and the order and types of its parameters. The definition of an operation specifies only its signature, not its implementation.

socket address

In NCS 1.5.1, a data structure that uniquely identifies a socket. A socket address consists of an address family identifier, a network address, and a port number.

In DCE RPC, the socket address is an internal data structure that is no longer manipulated directly by programmers using the API.

status parameter

A parameter with the **comm_status** or **fault_status** attribute. If an operation has a **comm_status** status parameter, communications errors that occur during execution of the operation are passed to the client in this parameter. If an operation has a **fault_status** status parameter, certain errors in the remote application code are passed to the client in this parameter.

stub

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

In NCS 1.5.1, the **NIDL compiler** generates client and server stub code from an interface definition.

switch

In NCS 1.5.1, a module in client programs used to eliminate naming conflicts. DCE programs do not use such a module.

task

In NCS 1.5.1, one of several threads of execution within a single process. Concurrent Programming Support provides calls that create and manage a multitasking environment. DCE RPC depends on a POSIX threads implementation for multitasking. *See also* **thread**.

thread

In DCE, a single sequential flow of control within a process.

transfer syntax

A set of encoding rules used for transmitting data over a network and for converting application data to and from different local data representations.

transmitted type

For data types with the NIDL or IDL **transmit_as** attribute, the data type that stubs pass over the network. Stubs invoke conversion routines to convert the transmitted type to a presented type, which is manipulated by clients and servers.

transport independence

The capability, without changing application code, to use any transport protocol that both the client and server systems support, while guaranteeing the same call semantics.

transport protocol

A communications protocol from the transport layer of the OSI network architecture, such as the Transmission Control Protocol (TCP) or the User Datagram Protocol (UDP).

transport service

A network service that provides end-to-end communications between two parties, while hiding the details of the communication network.

type

A class of object. All objects of a specific type can be accessed though the same interface or interfaces.

type UUID

The universal unique identifier that identifies the particular type of object and the associated manager.

unbound handle

In NCS 1.5.1, an RPC handle that identifies an object but not a location. This term is synonymous with "allocated handle."

DCE RPC does not support unbound handles.

Universal Unique Identifier (UUID)

An identifier that is immutable and unique across time and space. A UUID can uniquely identify an entity such as an RPC interface or object.

unmarshalling

The process by which an RPC stub disassembles incoming network data and converts it into local data in the appropriate local data representation. *See also* **marshalling.**

varying array

An array whose elements do not all need to be transmitted during a remote procedure call.

well-known endpoint

In DCE RPC, a preassigned endpoint that a server uses every time it runs. Well-known endpoints typically are assigned by a central authority responsible for a transport protocol. An application declares a well-known endpoint either as an attribute in an RPC interface header or as a variable in the server application code. In NCS 1.5.1, this is a **well-known port.** *See also* **dynamic endpoint, endpoint.**

well-known port

In NCS 1.5.1, a port whose port number is part of the definition of an interface. Clients of the interface always send to that port; servers always listen on that port. In DCE RPC, this is called a **well-known endpoint**. *See also* **opaque port**.

Index

Symbols

#define, replacement in IDL, 4–11

$ (dollar sign), ANSI C compliance, 4–4, 6–2

+e, C compiler flag, 6–2

–i, uuidgen option, 4–9

–I, idl option, 3–11, 4–12

–idir, NIDL option, 3–11, 4–12

–keep c_source, idl option, 4–12

–m, NIDL option, 4–12

–no_cpp, idl option, 4–12

–no_mepv, idl option, 4–12, 6–2

–s, uuidgen option, 6–2

–s, NIDL option, 4–12

_v2.acf extension, 4–6

_v2.idl extension, 4–4

A

absolute pathnames, avoiding, 3–11

ACF. *See* attribute configuration file

active context handle, GL–1

address
- family, GL–13
  - specifying in DCE RPC, 2–3
- network, 2–2, GL–13
- of server instance, 2–2
- server, GL–19
- socket, GL–20

alias, 3–4, GL–1

allocating, a handle, GL–2

allocating memory
- with **enable_allocate**, 3–19
- from the heap, 3–19
- for object UUID vector, 6–9

ANSI C compliance, 4–4, 6–2

API, GL–2
- concepts, 2–1 to 2–22
- DCE RPC routines, A–1 to A–40
- differences, 1–4
- overview of DCE RPC, 2–6 to 2–9

application programming interface. *See* API

application thread, GL–2
- client, GL–5
- local, GL–11
- server, GL–19

arrays, GL–2
and attributes, 3–4 to 3–5
conformant, GL–6
fixed, conformant, varying, 3–4 to 3–5
marshalling, 3–5
DCE RPC, GL–3
open, GL–6, GL–15
and pointers, 3–4
string attribute, 3–5
support in DCE, 1–5
varying, GL–22

asynchronous signals, handling, 5–18 to 5–22

at-most-once semantics, GL–2

attribute configuration file, 1–5 to 1–6, GL–2
attributes, 3–16, C–6
example, 3–20, 4–2
generated by **nidl_to_idl**, 4–6
using different, 3–16
writing, 3–15 to 3–19, 4–11

attribute configuration language, GL–3

attributes, GL–2
DCE IDL, 3–2 to 3–7

attributes for compatibility, 2–21, 3–7, 3–12, 3–15
when not to use, 4–4

authentication services, 2–9

authorization routines, 1–4

auto_handle attribute, 3–17, C–6

automatic binding, GL–3

auxiliary file, IDL generated, 3–21

B

basic types, 3–2

Berkeley socket abstraction, 2–1 to 2–2

binding, GL–3
attributes, 3–17
automatic, GL–3
client routine summary, A–30
DCE routines for, A–4 to A–6
for multiple objects, example, 6–1
information, GL–4
obtaining, 2–4
manager routine summary, A–38
manual, GL–12
method
automatic, GL–3
explicit, GL–8
implicit, GL–10
NCS 1.5.1, GL–3
server routine summary, A–33
setting, GL–20
states, GL–4

binding handle, 1–5, 2–4 to 2–6, GL–3
client, GL–5
concurrent, 1–7
customized, 3–6, GL–6
importing, 7–8
looking up a set, 7–12
for multiple threads, 2–10
NCS 1.5.1 states, 2–5 to 2–6
primitive, GL–16
resetting, 2–10
routine for creating binding, 2–10
server, GL–19
server routine summary, A–35
unbound, bound, partially bound, 2–5 to 2–6
using partially bound, 2–6 to 2–7
vector, GL–3
See also handle

bindings, managing, 3–17

binopfw example, 5–1 to 5–7
binopfw.idl, 4–2
binopfw_add, 5–2
client.c, 5–1 to 5–5
executing, 5–13
manager.c, 5–11
NCS 1.5.1 version, B–1 to B–8
server.c, 5–6 to 5–11
use_all_protseqs example, 5–14 to 5–15
util.c, 5–11

bitset, 3–13

bitset enum, 3–7

boolean type, 3–3

bound-to-host handle, 2–6, GL–4

bound-to-server handle, 2–5, GL–4

broadcast, 3–14, GL–4
operations, 3–10

broadcast attribute, 2–6, C–3

broadcasting RPCs, in DCE RPC, 2–6

brokers, GL–4

building DCE applications, 5–12

byte type, 3–3

C

call thread, GL–4

cancel, GL–5
pending, 5–20

cancellable function, 5–19

cancellation points, for delivering asynchronous signals, 5–19

CATCH macro, 5–17

CATCH_ALL macro, 5–17

_caux.c extension, 3–21

CDS. *See* Cell Directory Service

cdsd, 7–5

Cell Directory Service, 1–3, 7–5

Cell Directory Service daemon, 7–5

character type, 3–3

clean-up operations, context handles, 3–6

cleanup handlers, GL–5

client, GL–5
application thread, GL–5
binding handle, GL–5
converting to DCE RPC, 5–1 to 5–5
endpoint map service routines, 7–3
name service routines, 7–7 to 7–13
NCS 1.5.1 **binopfw**, B–1 to B–3
NCS 1.5.1 **stacks**, B–10
stacks example, 6–4 to 6–17
stub, controlling size with **nocode**, 3–18
switches, GL–21
using **rpc_ep** routines, 7–3
using **rpc_ns** routines, 7–7 to 7–13

client stubs, GL–5

code attribute, 3–18, C–6

comm_status attribute 3–14, 3–18, C–6

communication failures, handling, 3–18

communication services
API routines, 2–7, A–3 to A–11
RPC, GL–17

communications link, GL–6

compatibility, and version attributes. *See* interoperability

compatible server, GL–6

complex declarators, 3–13

concurrent binding handle, 1–7

Concurrent Programming Support, GL–6

conformant array, 1–5, 3–4, GL–6

const, 3–11

constant declarations, 3–11 to 3–12, 4–10 to 4–11

constants, 1–5

constructed types, DCE IDL, 3–2 to 3–7

context handle, 1–5, 3–6, GL–6
    active, GL–1

context_handle attribute, 3–6, 3–15, C–5

context_handle, type attribute, 3–12

control program, RPC, GL–17

conventions
    interface names, 3–8 to 3–9, 3–21

converting programs
    to DCE, 1–8 to 1–9, 2–21 to 2–22
    using **nidl_to_idl**, 4–1 to 4–8

CPS (Concurrent Programming Support), GL–6

creating, endpoints dynamically, 2–6

_cstub.c extension, 3–21, 4–13

_cswtch.c extension, 3–21

customized handles, 3–6, GL–6

customizing, applications with an ACF, 3–15 to 3–19

D

daemon
    CDS, 7–5
    cdsd, 7–5
    GLB, GL–9
    LLB, GL–11
    llbd, 2–4, 2–6
    RPC, GL–17
    rpcd, 2–4, 2–6, 5–2

data, transferring with pipes, 3–3

data representation, controlling with **represent_as**, 3–19

data types
    compatible network representations, 2–22
    DCE IDL, 3–2 to 3–7

DCE
    API, 1–3
    architecture, 1–1 to 1–2
    components of, 1–2
    converting programs to, 1–8 to 1–9
    IDL features, 3–1 to 3–21
    overview, 1–1 to 1–9
    writing interface definitions, 4–1 to 4–13

DCE ACF, *See* attribute configuration file

DCE client routine summary, A–30

DCE management routine summary, A–38

DCE RPC
    API routines, A–1 to A–40
    bindings and handles, 2–4 to 2–6
    building applications, 5–12
    concepts, 2–1 to 2–22
    handling signals, 5–16 to 5–20
    migrating to, 2–21 to 2–22
    overview of API, 2–6 to 2–9
    replacement for socket address, 2–1 to 2–3
    UUID string representation, 2–20

DCE RPC summary, A–30
    client routines, A–30
    management routines, A–38
    server routines, A–33

DCE server routine summary, A–33

DCE Threads, 1–6
    thread status, 5–17
    See also threads

dce/exc_handling.h, 5–18

DCE/NCS, differences, 1–4 to 1–7

dce_error_inq_text, 2–18, 5–11, A–13

debugging server applications, with **rpc_$set_fault_mode**, 2–12
directory service, component of DCE, 1–3
distributed applications, converting, 5–1 to 5–22
DNS (Domain Name Service), 1–3
documentation, related, iv
$ (dollar sign), ANSI C compliance, 4–4, 6–2
Domain Name Service (DNS), 1–3
dynamic endpoint, 2–6, GL–7

E

+e, C compiler flag, 6–2
enable_allocate attribute, 3–19, C–6
endpoint, 2–2, 3–10, GL–7
  attribute, C–2
  dynamic, 2–6, GL–7
  forwarding example, 5–1 to 5–5
  well-known, 3–10, GL–22
endpoint map, 2–4, GL–11
  adding entries to, 2–6
  DCE routines for, A–11 to A–13
  entries, 7–2
  manager routine summary, A–38
  registering and unregistering with, 7–4
  routines for, 2–15, 7–2
  server routine summary, A–35
  services, API routines for, 2–8
endpoint map routines, compared with location broker calls, 7–3
endpoint map service, 2–8, 7–2 to 7–5, A–11 to A–13
enpoint map, client routine summary, A–31
entry point vectors, GL–7
enum, 3–7
enumerations, in DCE IDL, 3–7
EPVs, GL–7
  generating. *See* manager EPVs
error_$, data types, 2–18
error_$ routines, 2–17 to 2–22
error_$c_get_text, 2–18
error_$c_text, 2–18
error_status_t, 3–3
errors
  client routine summary, A–31
  DCE routine for, A–13
  error service, 2–8
  handling, 3–18
  manager routine summary, A–38
  server routine summary, A–35
example. *See* program example
examples
  binopfw, 5–1 to 5–5
  binopfw, NCS 1.5.1 version, B–1 to B–8
  lookup, **7–18**
  stacks, 6–1 to 6–17
  stacks, NCS 1.5.1 version, B–8 to B–18
  string_conv, **7–9 to 7–13**
  using **nidl_to_idl**, 4–3 to 4–8
exception, 5–16, GL–7
  handling comm errors, 3–18
  scope, 5–18, GL–7
  See also exception handling
EXCEPTION macro, 5–17
exception-returning, DCE package, 1–6, 2–18,, 5–16 to 5–20, GL–7
  program example, 5–2, 5–7
  using, 5–17 to 5–18
executing, **binopfw** program, 5–13

explicit binding method, GL–8

explicit handle, GL–8
  example, 3–17

explicit_handle attribute, 3–17, C–6

export
  an interface, GL–8
  operations, A–18

extensions
  IDL, 3–21
  nidl_to_idl, 4–4, 4–6

F

fault handling, 1–6

fault_status attribute, 3–18, C–6

faults
  handling, 3–18
  handling in DCE, 5–16 to 5–20
  no replacement for **rpc_$set_fault_mode**, 5–20

first_is attribute, C–4

fixed arrays, 3–4 to 3–5

floating-point types, 3–3

forwarding endpoints, 2–6

forwarding RPCs, GL–8

full pointer, 3–4, GL–8

fully bound handle, 2–5, GL–8

function pointers, 3–13

G

GDA (Global Directory Agent), 1–3

GDS (Global Directory Service), 1–3

generating, object UUIDs, 6–2 to 6–17

generating stubs, controlling, 3–18

generic handle, 3–6, GL–8
  analogous to DCE customized handle, 3–6

GLB,
  See Global Location Broker

GLB database, CDS differences, 7–5

Global Directory Agent (GDA), 1–3

Global Directory Service (GDS), 1–3

Global Location Broker, 1–6, 2–5, GL–9
  and NSI, 7–1

group, name service entry, A–19
  See also name service

H

handle, 3–6, GL–9
  bound-to-host, GL–4
  bound-to-server, GL–4
  context, 3–6
  customized, 3–6, GL–6
  explicit, GL–8
  fully bound, GL–8
  generic, GL–8
  implicit, GL–10
  partially bound, GL–15
  primitive, GL–16
  RPC, GL–18
  unbound, GL–22

handle attribute, C–4

handle, type attribute, 3–12

handle_t, 2–19, 3–3, 4–11

heap attribute, 3–19, C–6

host IDs, GL–9
hosts, GL–9

I

–i, uuidgen option, 4–9
–I, idl option, 3–11, 4–12
idempotent attribute, 3–14, C–3
idempotent semantics, GL–9
–idir, NIDL option, 3–11, 4–12
IDL, GL–9
  attribute configuration file, 1–5 to 1–6
  auxiliary file, 3–21
  compiler, GL–9
  compiler options, 4–12 to 4–13
  component of DCE RPC, 1–3
  data types, constructed types, attributes, 3–2 to 3–7
  differences, 1–5 to 1–6
  extensions generated, 3–21, 4–13
  generated files, 3–21, 4–13
  header files, 3–21
  output files, 3–21
  overview of DCE features, 3–1 to 3–21
  running compiler, 4–12 to 4–13
  stub files, 3–21
  warning, 3–9
IDL attributes, C–1, C–2
  broadcast, C–3
  context_handle, C–5
  endpoint, C–2
  first_is, C–4
  for arrays, C–4
  for context handles, C–5
  for customized handles, C–4
  for interface definition headers, C–2
  for NCS compatibility, C–5
  for operations, C–3
  for parameters, C–3
  for pointers, C–4
  for structures, C–3
  for type declarations, C–5
  for unions, C–3
  handle, C–4
  idempotent, C–3
  ignore, C–3
  in, C–3
  last_is, C–4
  length_is, C–4
  local, C–2
  max_is, C–4
  maybe, C–3
  out, C–3
  pointer_default, C–2
  ptr, C–4
  ref, C–4
  size_is, C–4
  string, C–4
  transmit_as, C–5
  uuid, C–2
  v1_array, C–5
  v1_enum, C–5
  v1_string, C–5
  v1_struct, C–5
  version, C–2
  See also attribute names
idl_char, 3–13
ignore attribute, 3–4, C–3
implementing, an interface, GL–9
implicit binding method, GL–10
implicit handle, GL–10
implicit_handle attribute, 3–10, 3–17, C–6
import
  declarations, 3–11, 4–10
  interface attribute, 4–10
  operations, A–18
importing, interfaces, GL–10

importing binding handles, 7–8

in attribute, 3–3, C–3

in_line attribute, 3–19, C–6

include files
  DCE exception returning, 5–18
  for DCE RPC programs, 5–2
  pthreads, 5–18
  required for DCE, 5–7

initializing, server, program example, 5–7

initializing memory, with **enable_allocate**, 3–19

integer types, 3–3

interface, GL–10
  attributes, 3–9 to 3–12, 4–10 to 4–11
  client routine summary, A–30
  DCE routines for, A–7
  implementing, GL–9
  manager routine summary, A–38
  routine for registering, 2–12
  RPC, GL–18
  server routine summary, A–34
  unregistering, 2–12

interface definition
  and attribute configuration file, 1–5 to 1–6
  attributes, 3–9 to 3–10
  constant declarations, 3–11 to 3–12, 4–10 to 4–11
  converting to DCE, 4–1 to 4–8
  differences, 1–5
  example, 3–20, 4–2
  generating skeletal, 4–9
  import declarations, 3–11 to 3–12, 4–10 to 4–11
  interface attributes, 3–9 to 3–12, 4–10 to 4–11
  interoperability with NCS 1.5.1, 2–22
  language, GL–10
  local attribute, 3–10
  naming, 3–8 to 3–12, 4–10 to 4–11
  NCS 1.5.1 stacks, B–8
  operation declarations, 3–14 to 3–16, 4–11
  overview, 3–1 to 3–2
  procedure for writing, 4–9 to 4–11
  replacing $s, 4–4
  skeletal, generating, 4–9 to 4–13
  stacks example, 6–1 to 6–17
  structure of, 3–8
  translating to DCE, 4–1 to 4–8
  type declarations, 3–12, 4–11
  versioning, 3–9
  writing, 4–1 to 4–13

interface definition language. *See* IDL

interface UUIDs, GL–10
  generating, 4–9 to 4–13

Internet Protocol, 2–4

interoperability
  and attribute configuration file, 1–5 to 1–6, 3–16
  controlling with version numbers, 3–9
  guaranteeing with **nidl_to_idl**, 4–4
  See also attributes for compatibility

K

–keep c_source, idl option, 4–12

L

last_is attribute, 3–5, C–4

lb_$, data types, 2–18 to 2–19

lb_$ routines, 2–5, 2–15 to 2–22, 7–2
  and **rpc_ep** routines, 7–2 to 7–18
  and **rpc_ns** routines, 7–6 to 7–7

lb_$entry_t, 2–19

lb_$lookup_handle_t, 2–19

lb_$lookup_interface, 2–16

lb_$lookup_object, 2–15

lb_$lookup_object_local, 2–16

lb_$lookup_range, 2–16

lb_$lookup_type, 2–16

lb_$register, 2–16

lb_$server_flag_t, 2–19

lb_$unregister, 2–16

length_is attribute, 3–5, C–4

LLB, GL–11
routines, 2–15, 7–2

llbd, 2–4, 2–6

llbd daemon, replaced by rpcd daemon, 7–2

local
and function pointers, 3–13
interface attribute, 3–10

local application thread, GL–11

local attribute, C–2

local database.
See endpoint map, Location Broker database

local endpoint map, GL–11

Local Location Broker, GL–11
and endpoint map, 7–1

local management
client routine summary, A–31
manager routine summary, A–38
server routine summary, A–35

local/remote management
client routine summary, A–31
manager routine summary, A–38
server routine summary, A–35

locating a specific server, 7–13

Location Broker, GL–12
Global, 7–1
Local, 7–1
routines, 2–15 to 2–22, 7–2 to 7–18

Location Broker Client Agent, GL–11, GL–12

Location Broker database, 2–4
compared with endpoint map, 7–2

Location Broker routines, compared with name service routines, 7–6

lookup calls, name service, 7–12

lookup example, 7–18

M

–m, NIDL option, 4–12

macros, **TRY/CATCH**, 1–6
See also **CATCH, CATCH_ALL, EXCEPTION, TRY**

major version number, 3–9
See also **version** attribute

Makefile, for DCE RPC programs, 5–12

management
component of DCE, 1–3
DCE routines for, A–14 to A–18
server routine summary, A–36

management services, GL–18
API routines for, 2–8
local, 2–8
remote, 2–8

manager, GL–12
EPV, GL–12
EPVs, 6–2
program example, 6–9
NCS 1.5.1 **binopfw**, B–8
routine for registering, 2–12
using **rpc_ep** routines, 7–5
writing multiple, 6–1

managing, distributed applications with context handles, 3–6

managing bindings, 3–17

manual binding, GL–12

marshalling, 3–21, GL–12
  arrays, 3–5
  code, 3–19

max_is attribute, 3–4, C–4

maybe attribute 3–14, C–3

memory
  allocating from the heap, 3–19
  allocating UUID vectors, 6–9
  initializing, 3–19
  management routines, 3–19

memory leaks, and context handles, 3–6

memory management, in stub support services, 2–9

migrating to DCE, 2–21 to 2–22

migration attributes,
  See attributes for compatibility

minor version number, 3–9
  See also **version** attribute

monitoring clients, with context handles, 3–6

multi-threaded server, GL–12

multiple threads
  support for, 1–6 to 1–9
  See also threads

N

name service, 1–4, A–18 to A–25, GL–18
  client routine summary, A–31
  client routines, 7–7 to 7–13
  component of DCE, 1–3
  database, 2–5
  DCE routines for, A–18 to A–25
  group entries, 7–5, A–19
  inquiries, 7–13
  interface, 1–6, GL–13
  lookup calls, 7–12
  manager routine summary, A–39
  objects, A–18
  profile entries, 7–5, A–19
  routines, 2–15 to 2–16
  server entries, 7–5, A–19
  server routine summary, A–36
  server routines, 7–13

Name Service Access utilities, 7–18

Name Service Interface. *See* name service

names
  IDL extensions, 3–21
  of interface definition files, 3–21
  interface maximum length, 3–9
  nidl_to_idl generated extensions, 4–4

naming the interface, 3–8 to 3–12, 4–10 to 4–11

nbase.idl, 3–13

NCA, GL–13, GL–14
  connection protocol, 2–4
  datagram service, 2–4

ncacn_ip_tcp, 2–4

ncadg_ip_udp, 2–4

NCK, GL–14

NCS, GL–14

NCS 1.5.1
  changing routines to DCE RPC, 2–9 to 2–18
  converting to DCE, 1–8 to 1–9
  data types, 2–18 to 2–21
  program examples, B–1 to B–18
  UUID string representation, 2–20

NDR, GL–14
  See also network data representation

ndr_$char, 3–13

network
  address family, GL–13
  addresses, 2–2, GL–13
  client routine summary, A–30

DCE routines for, A–7
server routine summary, A–34

Network Computing Architecture, GL–14

Network Computing Kernel, GL–14

Network Computing System, GL–14

network data representation, GL–14
associating local representation, 3–19
guaranteeing same, 4–4
See also attributes for compatibility

Network Interface Definition Language. *See* NIDL

NIDL, GL–14
changes to compiler options, 4–12
differences with IDL, 1–5

NIDL Compiler, GL–14

nidl_to_idl, 1–5, 1–8, 2–21, 2–22, 4–1 to 4–8
ACF example, 4–6
converting Pascal example, 4–7
invoking, 4–1 to 4–8
_v2.acf extension, 4–6
_v2.idl extension, 4–4
warnings, 4–2

nocode attribute, 3–18, C–6

–no_cpp, idl option, 4–12

–no_mepv, idl option, 4–12, 6–2

API, GL–2

NSA (Name Service Access), 7–18

NSI, (Name Service Interface). A–18, GL–14
See also name service

null–terminated strings, 3–5

O

object, GL–15
assigning type to, 2–12
type, GL–15
type UUIDs, GL–15

object UUIDs, 2–2, GL–15
creating vector, 6–9
DCE routines, A–8
preserving, 6–2
server routine summary, A–34

obtaining binding information, 2–4

opaque ports, GL–15

open arrays, GL–6, GL–15
See also conformant array

operation, GL–15
declarations, 4–11
changes in IDL, 3–14 to 3–15
IDL example, 3–14
NCS 1.5.1 example, 3–14
parameters, 3–15 to 3–16
returning pointers, 3–10

out attribute, 3–3, C–3

out_of_line attribute, 3–19, 3–21, C–6

P

parameters, 3–15 to 3–16
ACF example, 4–7
IDL example, 4–6, 4–7
status, GL–20

partially bound handle, 2–6, GL–15
program example, 6–4
using, 2–6 to 2–7

partially bound server binding handles, resolving, 7–3

Pascal, interface example, 4–7

pathnames, avoiding listing absolute, 3–11

PFM.
See Process Fault Manager

pfm_$ data types, 2–19

pfm_$ routines, 2–18 to 2–22
replacement for, 1–6

pfm_$cleanup_rec, 2–19

pfm_$init, 5–19
replacement for, 5–2, 5–7, 5–16

pipes, 1–5, 3–13
and attributes, 3–3

pointer, alias, 3–4, GL–1

pointer_default attrbute, 3–10, C–2

pointers, 3–3 to 3–4
alias, 3–4, GL–1
and arrays, 3–4
full, GL–8
function, 3–13
ignore attribute, 3–4
in DCE, 1–5
parameters, 3–4
ptr attribute, 3–4
ref attribute, 3–3, 3–4
reference, GL–17
return value, 3–10
setting default behavior, 3–10

port, 2–2, 3–10, GL–16
numbers, GL–16
opaque, GL–15
well-known, GL–23

presented types, GL–16

primitive binding handle, GL–16

procedure
for building DCE applications, 5–12
for writing interface definitions, 4–9

Process Fault Manager, 5–16, GL–16
routines, 2–18 to 2–22
See also **pfm_$** routines

profile, name service entry, A–19
See also name service

program examples
generating object types and UUIDs, 6–2
multiple managers, 6–14 to 6–16
NCS 1.5.1, B–1 to B–18
See also examples

protocol families, GL–16

protocol sequence, GL–18
checking valid, 2–14
endpoint attribute, 3–10
as part of a string binding, 2–2
registering, 2–11
replaces address family, 2–3
using all, program example, 5–14 to 5–15
vector, GL–17

protocol sequences, client routine summary, A–31

protocols
RPC, GL–18
supported DCE, 1–2
transport, GL–21

pthread_cancel, 5–16, 5–18

pthread_cancel_e, 5–17, 5–18

pthread_cond_wait, 5–19

pthread_signal_to_cancel_p, 5–18

pthreads. *See* threads

ptr attribute, 3–4, 3–10, 3–15, C–4

ptr, type attribute, 3–12

R

ref attribute, 3–3, 3–10, 3–15, C–4

ref, type attribute, 3–12

reference pointer, 3–3, GL–17

registering
with the endpoint map, 7–4, GL–17
an interface, GL–17

the interface and managers, 2–12
multiple managers, 6–2
an object, GL–17

related manuals, iv

remote procedure call, GL–17
binding occurs, 2–4 to 2–6
broadcasting, 2–6
forwarding endpoints, 2–6
See also RPC

represent_as attribute, 3–19, C–6

resolving a partially bound server binding, with **rpc_ep_resolve_binding**, 7–4

RPC
Communication Services, GL–17
component of DCE, 1–3
control program, GL–17
daemon, 2–4, GL–17
handle, GL–18
interface, GL–18
management services, GL–18
name service, GL–18
protocol sequence, GL–18
protocols, GL–18
routines, diffeneces in APIs, 2–6
runtime, defined, GL–18
runtime library, GL–19
threads, GL–19

RPC API, concepts, 2–1

RPC communications, 2–1 to 2–4
setting timeout value, 2–11

RPC routines, API differences, 1–4

RPC runtime
listening for rpcs, 2–12
registering endpoint information, 2–11
registering protocol sequences with, 2–11
stopping server from listening, 2–12

rpc_$, data types, 2–19

rpc_$ routines
changing to DCE RPC, 2–9 to 2–13
client calls, 2–10, 2–12
server calls, 2–11, 2–12

rpc_$alloc_handle, 2–10

rpc_$allow_remote_shutdown, 2–12

rpc_$are_you_there, 2–13

rpc_$bind, 1–7, 2–10

rpc_$clear_binding, 2–10

rpc_$clear_server_binding, 2–10

rpc_$dup_handle, 2–10

rpc_$epv_t, 2–19

rpc_$free_handle, 2–11

rpc_$generic_ept_t, 2–19

rpc_$if_spec_t, 2–19

rpc_$inq_binding, 2–12

rpc_$inq_interfaces, 2–13

rpc_$inq_object, 2–12

rpc_$inq_stats, 2–13

rpc_$listen, 2–12

rpc_$mgr_epv_t, 2–19

rpc_$name_to_sockaddr, 2–12

rpc_$register, 2–12

rpc_$register_mgr, 2–12, 6–9

rpc_$register_object, 2–12

rpc_$set_asyc_ack, 2–11

rpc_$set_binding, 2–10

rpc_$set_fault_mode, 2–12
no replacement for, 5–20

rpc_$set_short_timeout, 2–11

rpc_$shut_check_fn_t, 2–19

rpc_$shutdown, 2–12, 2–13

rpc_$sockaddr_to_name, 2–13

rpc_$unregister, 2–12

rpc_$use_family, 2–11

rpc_$use_family_wk, 2–11

rpc_binding_copy, 2–10, A–5

rpc_binding_free, 2–11, **7–8**, A–4

rpc_binding_from_string_binding, 2–10, 2–12, 2–14, A–4
program example, 5–2

rpc_binding_handle_t, 2–19, 2–20

rpc_binding_inq_auth_client, A–6

rpc_binding_inq_auth_info, A–6

rpc_binding_inq_object, 2–12, A–6

rpc_binding_reset, 2–10, A–5

rpc_binding_server_from_client, A–6

rpc_binding_set_auth_info, A–6

rpc_binding_set_object, A–6

rpc_binding_to_string_binding, 2–12, 2–13, A–5

rpc_binding_vector_free, A–4

rpc_c_protseq_max_reqs_default, 5–7

RPC_DEFAULT_ENTRY, **7–8**

rpc_ep routines
using in a client, 7–3
using in a server, 7–3

rpc_ep_register, 2–16, 7–3, A–12

rpc_ep_register_no_replace, 2–16, 7–3, A–12

rpc_ep_resolve_binding, 2–16, 7–3, A–13

rpc_ep_unregister, 2–16, 7–3, A–13

rpc_if_handle_t, 2–19

rpc_if_id_vector_free, A–7

rpc_if_id_vector_t, 2–20

rpc_if_inq_id, A–7

rpc_mgmt_authorization_fn_t, 2–19

rpc_mgmt_ep routines, using in a manager, 7–5

rpc_mgmt_ep_elt_inq_begin, 2–16, A–17

rpc_mgmt_ep_elt_inq_done, 2–16, A–17

rpc_mgmt_ep_elt_inq_next, 2–16, A–17

rpc_mgmt_ep_unregister, 2–16, A–18
using in a manager, 7–5

rpc_mgmt_inq_com_timeout, A–14

rpc_mgmt_inq_dflt_protect_level, A–14

rpc_mgmt_inq_if_ids, 2–13, A–16

rpc_mgmt_inq_is_server_listening, 2–13, A–17

rpc_mgmt_inq_server_princ_name, A–16

rpc_mgmt_inq_stats, 2–13, A–16

rpc_mgmt_inq_stop_server_listening, A–17

rpc_mgmt_set_authorization_fn, 2–12, A–15

rpc_mgmt_set_cancel_timeout, A–15

rpc_mgmt_set_com_timeout, 2–11, A–15

rpc_mgmt_set_server_stack_size, A–15

rpc_mgmt_stats_vector_free, A–15

rpc_mgmt_stop_server_listening, 2–12, 2–13

rpc_mgr_epv_t, 2–19

rpc_network_inq_protseq, 2–14

rpc_network_inq_protseqs, A–7

rpc_network_is_protseq_valid, 2–14, A–7

rpc_ns routines, compared
with **lb_$** routines, 7–6

rpc_ns_binding_export, 2–16, 7–13

rpc_ns_binding_import_begin, 7–8

rpc_ns_binding_import_done, 7–8
rpc_ns_binding_import_next, 7–8
rpc_ns_binding_lookup_begin, 2–16, 7–12
rpc_ns_binding_lookup_done, 2–16, 7–12
rpc_ns_binding_lookup_next, 2–16, 7–12
rpc_ns_binding_select, 7–12, 7–13
rpc_ns_binding_unexport, 2–16, 7–13
rpc_ns_entry_object_inq_begin, 2–15
rpc_ns_entry_object_inq_done, 2–15
rpc_ns_entry_object_inq_next, 2–15
rpc_ns_group, 7–13
rpc_ns_group_delete, A–23
rpc_ns_group_mbr_add, A–23
rpc_ns_group_mbr_inq_begin, A–24
rpc_ns_group_mbr_inq_done, A–24
rpc_ns_group_mbr_inq_next, A–24
rpc_ns_group_mbr_remove, A–23
rpc_ns_handle_t, 2–18
rpc_ns_mgmt_handle_set_exp_age, A–24
rpc_ns_mgmt_inq_exp_age, A–24
rpc_ns_mgmt_set_exp_age, A–24
rpc_ns_profile, 7–13
rpc_ns_profile_delete, A–25
rpc_ns_profile_elt_add, 7–13, A–25
rpc_ns_profile_elt_inq_begin, A–25
rpc_ns_profile_elt_inq_done, A–25
rpc_ns_profile_elt_inq_next, A–25
rpc_ns_profile_elt_remove, 7–13, A–25
rpc_object_inq_type, A–8
rpc_object_set_inq_fn, A–8
rpc_object_set_type, 2–12, 6–9, A–8
rpc_protseq_vector_free, A–7
rpc_s_ok, 2–18
rpc_server_inq_bindings, A–10
rpc_server_inq_if, A–11
rpc_server_listen, 2–12, A–11
rpc_server_register, 6–9
rpc_server_register_auth_info, A–11
rpc_server_register_if, 2–12, 6–9, A–10
rpc_server_unregister_if, 2–12, A–11
rpc_server_use_all_protseqs, 2–11, 2–16, A–9
  program example, 5–14 to 5–15
rpc_server_use_all_protseqs_if, 2–11, A–9
rpc_server_use_protseq, 2–11, 2–16, A–9
  program example, 5–7
rpc_server_use_protseq_ep, A–9
rpc_server_use_protseq_if, 2–11, A–9
rpc_ss_allocate, A–26
rpc_ss_client_free, A–28
rpc_ss_destroy_client_context, A–28
rpc_ss_disable_allocate, A–27
rpc_ss_enable_allocate, A–27
rpc_ss_free, A–27
rpc_ss_get_thread_handle, A–28
rpc_ss_set_client_alloc_free, A–27
rpc_ss_set_thread_handle, A–28
rpc_ss_swap_client_alloc_free, A–27
rpc_stats_vector_t, 2–20
rpc_string_binding_compose, 2–10, 2–14, A–5
  program example, 5–3

rpc_string_binding_parse, 2–14, A–5

rpc_string_free, A–26

rpccp (RPC control program), 7–5, GL–17

rpcd, 2–4, 2–6, 5–7, A–11 to A–12, GL–17, GL–19

rpcd daemon, 5–2, 7–2

rrpc_$, data types, 2–20

rrpc_$ calls, 2–13 to 2–22

rrpc_$interface_vec_t, 2–20

rrpc_$stat_vec_t, 2–20

rundown procedure, with context handles, 3–6

runtime, GL–18

runtime library, GL–19

S

–s, **uuidgen** option, 6–2

–s, NIDL option, 4–12

sample applications. *See* examples

_saux.c extension, 3–21

security service, 1–3

server, GL–19
- address, GL–19
- application thread, GL–19
- binding handle, GL–19
- binopfw program example, 5–6 to 5–11
- compatible, 2–4, GL–6
- context, GL–20
- DCE routines for, A–8 to A–14
- DCE routines for initializing, A–10
- endpoint map service routines, 7–3
- initializing, 5–6
- locating, 7–13
- name service routines, 7–13
- NCS 1.5.1 binopfw, B–4 to B–7
- NCS 1.5.1 stacks, B–13
- registering with the endpoint map, 7–4
- server routine summary, A–34
- stub, GL–20
- using **rpc_ep** routines, 7–3
- writing multiple managers, 6–1

server routines, name service, 7–13

service, GL–20
- transport, GL–22

set of binding handles, looking up, 7–12

sets, 3–7

short bitset, 3–13

short bitset enum, 3–7

short enum, 3–7, 3–13

signals, GL–20
- asynchronous, 5–16
  - delivery at cancellation points, 5–19
  - handling, 5–18 to 5–22
- handling, 1–6, 5–16 to 5–20
- no replacement for rpc_$set_fault_mode, 5–20
- synchronous, 5–16

signatures, GL–20

sigwait, 5–18

simple declarators, 3–13

simple types. *See* data types

size_is attribute, 3–4, C–4

skeletal interface definitions, generating, 4–9 to 4–13

socket
- address, 2–1 to 2–2, GL–20
  - differences, 1–5
  - replaced by protocol sequence, 2–3 to 2–4
- family ID, 2–2

socket_$, data types, 2–20

socket_$ calls, 2–13 to 2–22
replacing, 6–4
socket_$equal, 2–14
socket_$family_from_name, 2–14
socket_$family_to_name, 2–14
socket_$from_local_rep, 2–15
socket_$inq_broad_addrs, 2–15
socket_$inq_my_netaddr, 2–15
socket_$max_pkt_size, 2–15
socket_$set_wk_port, 2–15
socket_$to_local_rep, 2–15
socket_$to_name, 2–14
socket_$valid_families, 2–14
socket_$valid_family, 2–14
specifying binding information, DCE, 2–13 to 2–22
_sstub.c extension, 3–21, 4–13
stacks example, 6–1 to 6–17
amanager.c, 6–14 to 6–15
client.c, 6–4
lmanager.c, 6–16
NCS 1.5.1 version, B–8 to B–18
server.c, 6–9 to 6–17
stacksdf.h header file, 6–3
util.c, 6–17
state information, maintaining with context handles, 1–5, 3–6
status parameters, GL–20
status_$t, 3–3
data type, 2–18
string, 3–5 to 3–6
attribute, 3–15, C–4
client routine summary, A–31
DCE routine for, A–26
IDL and NIDL examples, 4–4
type attribute, 3–12
string binding, 2–4
comparison of NCS 1.5.1 and DCE RPC, 2–3
creating, A–4 to A–6
definition, 2–2 to 2–3
example of, 2–3
and handle, 2–4 to 2–6
string services, API routines for, 2–8
string_conv example
client.c, 7–9 to 7–13
server.c, 7–14 to 7–17, 7–14 to 7–17
string0, 3–13
NCS data type, 3–5
NIDL example, 4–4
strings
manager routine summary, A–39
server routine summary, A–35
struct, 3–7
structures, in DCE IDL, 3–7
stub, GL–21
client, GL–5
controlling generation of, 3–18
files generaged by IDL, 3–21
server, GL–20
stub memory management, A–26
stub support
client routine summary, A–33
server routine summary, A–37
stub support services, 2–9
switches, GL–21
syntax, transfer, GL–21

T

tasks, GL–21
threads, 1–2, 1–6, 2–11, GL–19, GL–21

call thread, GL–4
cancellation mechanism, 5–16
concurrent binding handles, 1–7
local application, GL–11
manipulating multiple, 2–10
multithreading, GL–13
and signals, 5–16

time service, DCE component, 1–3

timeouts, setting communication, 2–11

transfer syntax, GL–21

transferring large quantities of data, with pipes, 3–3

transmit_as attribute, 3–19 C–5, GL–21

transmit_as, type attribute, 3–12

transmitted types, GL–21

transport
independence, GL–21
protocols, GL–21
service, GL–22

TRY macro, 1–6, 5–17

TRY/CATCH macros, 1–6

type, GL–22
attributes, 3–12
basic, 3–2
data, transferring with pipes, 3–3
declarations, 3–12 to 3–13, 4–11
presented, GL–16
specifiers, 3–12 to 3–13
transmitted, GL–21
UUIDs, GL–22

type UUIDs, generating, 6–2

typedef, type declaration, 3–12

types, declarators, 3–13 to 3–21

typographic conventions, vi

U

UIDs, routines, 2–17 to 2–22

unbound handle, 2–6, 2–10, GL–22

union, 3–7

unions, in DCE IDL, 3–7

Universal Unique Identifiers. *See* UUIDs

UNIX signals. *See* signals

unmarshalling, GL–22

user–defined handles, 3–6

using exceptions, with DCE exception package, 5–17 to 5–18

uuid attribute, 3–9, C–2

uuid_$, data types, 2–20 to 2–21

uuid_$ routines, 2–17 to 2–22

uuid_$create_nil, 2–17

uuid_$decode, 2–17

uuid_$encode, 2–17, 5–21

uuid_$equal, 2–17

uuid_$from_string, 2–17

uuid_$gen, 2–17

uuid_$hash, 2–17

uuid_$nil, 1–7, 2–17, 5–7
program example, 5–2

uuid_$string_t, 1–7, 2–20, 5–22

uuid_$t, 1–7, 2–20
data structure, 5–20

uuid_$to_uid, 2–17

uuid_compare, 2–17, A–29

uuid_create, 2–17, A–29

uuid_create_nil, 1–7, 2–20, A–29
uuid_equal, 2–17, A–29
uuid_from_string, 2–17, A–29
uuid_gen, 2–20
uuid_hash, 2–17, A–29
uuid_is_nil, 2–17, A–29
uuid_t, 1–7, 2–20
  data structure, 5–20
uuid_to_string, 2–17, A–29
uuid_vector_t, initializing, 6–9
uuidgen, 2–17, 2–20, 4–9
  –i option, 4–9
  –s option, 6–2
UUIDs, GL–22
  client routine summary, A–33
  comparison of NCS 1.5.1 and DCE, 5–20 to 5–22
  converting NCS 1.5.1, 5–21
  DCE routines for, A–29
  differences, 1–7 to 1–8
  generating object, 6–2 to 6–17
  as global variable, 1–7
  initializing vector, 6–9
  interface, GL–10
  interface attribute, 3–9, 4–10
  manager routine summary, A–4
  NCS 1.5.1 string representation, 2–20
  object, generating, 6–2
  preserving NCS 1.5.1, 5–21
  routines, 2–17 to 2–22
  server routine summary, A–37
  services, 2–8
  string representation, 5–22
  using DCE, 5–20

V

v1_array attribute, 4–4, C–5
v1_enum attribute, 4–4, C–5
v1_string attribute, 4–4, C–5
v1_struct attribute, 4–4, C–5
_v2.acf extension, 4–6
_v2.idl extension, 4–4
varying arrays, 3–5, GL–22
vectors, creating, 6–9
version attribute, 3–9, 4–10, C–2
  and compatibility, 3–9
versioning interface definitions, 3–9
void type, 3–3
void *, 3–6

W

well-known endpoints, GL–22
well-known ports, GL–23
writing multiple managers, example, 6–1 to 6–17

WIN AN HP CALCULATOR!

Your comments help us determine how well we meet your needs. Returning this card with your name and address enters you in a quarterly drawing for an HP calculator*.

NCS 1.5.1 to DCE RPC Transition Guide
B3193-90002 E0293

What operating system and hardware do you use? (Type: **uname -rvm** and write the displayed information on the following line.)

_____ _____ _____

How much have you used this manual?

_____ Extensively _____ Often _____ Occasionally _____ Not at all

----fold here----------fold here----

Complete this section only if you have used this manual. Use the column labeled "NC" if you have no comment or opinion regarding that topic.

	Agree				Disagree		NC
The manual is well organized.	○	○	○	○	○	○	○
It is easy to find information in the manual.	○	○	○	○	○	○	○
It tells me clearly what I need to know.	○	○	○	○	○	○	○
It is easy to perform step-by-step procedures.	○	○	○	○	○	○	○
Overall, the manual meets my expectations.	○	○	○	○	○	○	○

Please take the time to describe any problems you've had or any suggestions that would improve our product/manual. Use additional pages if needed. The more specific your comments, the more useful they are to us. Thank you.

Comments:

---fold here----------fold here----

○ **Check here if you would like a reply.**

** Offer expires 02/01/95.*

Please tape here **Please print/type the following information** *Please tape here*

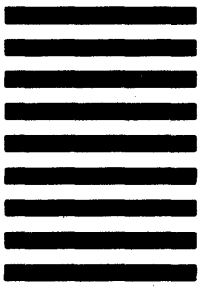
Name: _____ Telephone: _____

Company: _____

Address: _____

City: _____ State: _____ Zip Code/Country: _____

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 37 LOVELAND, COLORADO

POSTAGE WILL BE PAID BY ADDRESSEE

Learning Products HP-UX
Hewlett-Packard Company
3404 East Harmony Road
Fort Collins CO 80525-9988

NCS 1.5.1 to DCE RPC Transition Guide
B3193-90002 E0293

Manual Part No.
B3193-90002

Manufacturing
Part No.
B3193-90002

B3193-90002